Правительство Российской Федерации
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»
(НИУ ВШЭ)

Московский институт электроники и математики им. А.Н. Тихонова

ОТЧЕТ
О ПРАКТИЧЕСКОЙ РАБОТЕ № 2
по дисциплине «Криптографические методы защиты информации»
ТЕМА РАБОТЫ
Современные симметричные шифры

Студент гр. МКБ231
А.Р. Касимов
«24» марта 2024 г.

Руководитель
Заведующий кафедрой информационной безопасности киберфизических систем
канд. техн. наук, доцент
_____О.О. Евсютин
«___» _____ 2024 г.

Москва 2024

СОДЕРЖАНИЕ

1 Задание на практическую работу.

Целью работы является...

В рамках практической работы необходимо выполнить следующее:

1) написать программную реализацию одного из следующих симметричных шифров (по выбору студента):
   - Магма;
   - ***Кузнечик;***
   - AES;
2) подготовить отчет о выполнении работы.

Программа должна обладать следующей функциональностью:

1) принимать на вход файл, содержащий открытый текст, подлежащий зашифрованию, или шифртекст, подлежащий расшифрованию;
2) принимать на вход секретный ключ;
3) давать пользователю возможность выбирать режим работы блочного шифра*;
4) осуществлять зашифрование или расшифрование выбранного файла по выбору пользователя и сохранять результат в новом файле.

2 Теоретическая часть.

Основное отличие между современными и историческими (классическими) шифрами, к последним из которых относятся все алгоритмы шифрования докомпьютерной эпохи, заключается в способе представления данных. Современные шифры обрабатывают данные в цифровом виде как битовые последовательности, природа которых не имеет значения, в то время как исторические шифры могли работать лишь с текстами, что накладывало определенные ограничения на использовавшийся математический аппарат.

При этом основные криптографические преобразования остались прежними, лишь усложнился способ их реализации (здесь следует отметить, что речь идет исключительно о симметричных шифрах).

Основным методом криптоанализа исторических шифров является статистический криптоанализ, когда криптоаналитик изучает статистику встречаемости символов в шифртексте и, используя полученную информацию, может осуществить дешифрование зашифрованных данных.

До появления компьютерной криптографии данные, которые необходимо было защищать с помощью шифрования, практически всегда представляли собой текст на естественном языке, обладающем высокой избыточностью. Именно высокая избыточность открытых текстов является причиной успешного применения различных статистических атак для взлома классических шифров. В связи с этим К. Шенноном были предложены два основных криптографических метода сокрытия избыточности открытого текста: перемешивание и рассеивание.

Перемешивание устраняет зависимость между открытым текстом и шифртекстом, затрудняет попытки найти в шифртексте избыточность и статистические закономерности. Перемешивание осуществляется с помощью подстановок (замен).

Рассеивание перераспределяет избыточность открытого текста, распространяя ее на весь шифртекст. Каждый бит открытого текста при шифровании должен повлиять на максимальное число других бит открытого текста. Простейшим способом создания рассеивания является перестановка.

По отдельности ни перемешивание, ни рассеивание не могут обеспечить надежного шифрования, поэтому классические подстановочные и перестановочные шифры неприменимы в современном мире, однако криптосистемы, в которых реализованы оба этих метода, обладают высокой криптостойкостью.

Можно выделить следующие основные элементарные операции над данными, объединение и многократное повторение которых положено в основу всех современных симметричных алгоритмов шифрования:

1 замена элементов данных (короткие битовые последовательности) с помощью специальных таблиц;

2 перестановка элементов данных;

3 битовый сдвиг;

4 операции сложения/вычитания по модулю;

5 поразрядное сложение по модулю 2.

Ниже перечислены основные критерии оценки симметричных алгоритмов шифрования (в частности, данные критерии использовались при выборе стандарта шифрования AES).

- Реальная защищенность от криптоаналитических атак. При этом основными методами криптоанализа являются:

6 дифференциальный криптоанализ;

7 расширения для дифференциального криптоанализа;

8 поиск наилучшей дифференциальной характеристики;

9 линейный криптоанализ;

10 интерполяционное вторжение;

11 вторжение с частичным угадыванием ключа;

12 вторжение с использованием связанного ключа;

13 вторжение на основе обработки сбоев;

14 поиск лазеек.

- Статистическая безопасность криптографического алгоритма.
- Надежность математической базы криптографического алгоритма.
- Расчетная сложность криптографического алгоритма для программной и аппаратной реализаций (может оцениваться скоростью преобразований).
- Требования к памяти при программной и аппаратной реализациях. При аппаратной реализации оценивается числом логических элементов, при программной – количеством необходимой оперативной и постоянной памяти, в том числе для различных платформ и сред.
- Гибкость алгоритма, то есть:

15 возможность работы с иными длинами начальных ключей и информационных блоков;

16 безопасность реализации в широком диапазоне различных платформ и приложений, включая 8-битовые процессоры;

17 возможность использования криптографического алгоритма в качества поточного шифра или генератора псевдослучайных чисел, алгоритма хеширования, для обеспечения подлинности сообщений (выработка кодов аутентичности сообщений) и т.п.;

18 одинаковая сложность как аппаратной, так и программной реализации, а также программно-аппаратной реализации.

Современная криптография выделяет два типа симметричных криптосистем: блочные и поточные.

Поточные шифры преобразуют открытый текст в шифртекст последовательно по одному биту путем сложения битов открытого текста с битами гаммы по модулю 2. Гамма необходимой длины формируется из секретного ключа шифрования фиксированной длины. При этом поточный шифр может быть построен на основе блочного шифра при использовании того в специальном режиме.

Блочные шифры обрабатывают открытый текст, разбивая его на блоки равного размера, причем все современные блочные шифры являются раундовыми. Это означает, что зашифрование одного блока открытого текста (расшифрование одного блока шифртекста) заключается в многократном применении к нему некоторой последовательности элементарных криптографических операций. Соответственно, однократное применение такой последовательности к блоку данных называется раундом.

Кратко опишем современные стандарты симметричного шифрования.

2.2 ГОСТ 28147-89

Первый российский стандарт симметричного шифрования описан в нормативном документе «ГОСТ 28147-89. Системы обработки информации. Защита криптографическая. Алгоритм криптографического преобразования». В настоящее время данный стандарт является устаревшим, он утратил силу 31 декабря 2015 г.

ГОСТ 28147-89 является блочным шифром, построенным на основе сети Фейстеля и работающим с блоками длиной 64 бита. Ключ шифрования составляет 256 бит.

ГОСТ 28147-89 предназначен для работы в следующих режимах:

19 режим простой замены;

20 режим гаммирования;

21 режим гаммирования с обратной связью;

22 режим выработки имитовставки.

Базовым режимом работы ГОСТ 28147-89 является режим простой замены. В этом режиме блоки открытого текста последовательно и независимо друг от друга преобразуются в блоки шифртекста. Зашифрование одного блока данных происходит в течение 32-х раундов основного криптографического преобразования.

В режиме гаммирования открытый текст, разбитый на 64-битовые блоки, зашифровывается путем поразрядного сложения по модулю 2 с гаммой шифра, которая вырабатывается блоками по 64 бита. Выработка блоков гаммы осуществляется следующим образом. С помощью простого рекуррентного соотношения для каждого блока открытого текста вычисляется некоторое значение той же длины, что и блок. Данное значение зашифровывается в режиме простой замены и образует очередной блок гаммы. Для инициализации данного процесса необходимо задать 64-битовое начальное значение, называемое синхропосылкой.

Режим гаммирования с обратной связью очень похож на режим гаммирования и отличается от него только способом выработки гаммы. Сначала, как и в режиме гаммирования, необходимо задать 64-битовую синхропосылку, которая затем зашифровывается в режиме простой замены. Полученное значение является первым блоком гаммы, который накладывается на первый блок открытого текста путем побитного сложения по модулю 2.

Полученный в результате первый блок шифртекста зашифровывается в режиме простой замены, и результат шифрования используется в качестве второго блока гаммы, то есть побитно складывается по модулю 2 со вторым блоком открытого текста.

Таким образом, каждый блок гаммы, за исключением первого, получается путем шифрования в режиме простой замены предыдущего блока шифртекста.

Рассмотренные режимы работы стандарта шифрования ГОСТ 28147-89 обеспечивают непосредственно шифрование данных, то есть служат для обеспечения конфиденциальности. Кроме того, в ГОСТ 28147-89 предусмотрен режим, предназначенный для решения задачи обеспечения контроля целостности информации, – режим выработки имитовставки

Имитовставка – это контрольная комбинация, зависящая от открытого текста и секретного ключа, использующаяся для обнаружения всех случайных или преднамеренных изменений в открытом тексте.

Выработка имитовставки осуществляется следующим образом: открытый текст разбивается на блоки длиной 64 бита, первый блок зашифровывается 16-ю циклами режима простой замены, полученное значение побитно складывается по модулю 2 со вторым блоком открытого текста, результат зашифровывается 16 циклами режима простой замены и побитно складывается по модулю 2 с третьим блоком открытого текста. Эти действия повторяются для всех блоков открытого текста, и в итоге мы получим 64-битовое значение. В качестве имитовставки можно взять любое количество бит этого значения.

В режиме выработки имитовставки используется то же основное криптографическое преобразование, что и в режиме простой замены, но течении 16-ти циклов, а не 32-х. Длина ключа так же составляет 256 бит.

Имитовставка добавляется отправителем к защищаемым данным, которые могут передаваться и в открытом виде, если задача обеспечения конфиденциальности не ставится. Получатель, знающий секретный ключ, вычисляет имитовставку и сравнивает вычисленное значение с полученным. Если обнаруживается несовпадение, полученные данные отвергаются как ложные.

Таким образом, для потенциального злоумышленника практически неразрешимы следующие задачи: вычисление имитовставки для заданного открытого текста, и подбор открытого текста для заданного значения имитовставки. Имитовставка обеспечивает невозможность имитации и подмены данных.

2.3 ГОСТ Р 34.12-2015

Действующий российский стандарт симметричного шифрования описан в документе «ГОСТ Р 34.12-2015. Информационная технология. Криптографическая защита информации. Блочные шифры». Данный стандарт был разработан на смену ГОСТ 28147-89 и введен 1 января 2016 г.

ГОСТ Р 34.12-2015 описывает два симметричных блочных шифра:

23 шифр «Магма» с длиной блока 64 бита и длиной ключа шифрования 256 бит;

24 шифр «Кузнечик» с длиной блока 128 бит и длиной ключа шифрования 256 бит.

Шифр «Магма» представляет собой тот же самый шифр, который был описан в стандарте ГОСТ 28147-89. Отличие заключается лишь в том, что в шифре «Магма» определена фиксированная таблица замен (один из этапов основного криптографического преобразования в данном шифре заключается в замене 4-битовых

подблоков блока данных с помощью специальной таблицы), в то время как в ГОСТ 28147-89 данная таблица не определена и ее выбор оставлен на усмотрение разработчиков средств криптографической защиты информации, реализующих соответствующий алгоритм шифрования.

Шифр «Кузнечик» является вновь разработанным шифром. Его создание было обусловлено потребностью в криптографическом алгоритме, работающем с большей длиной блока, нежели 64 бита. В отличие от шифра «Магма» шифр «Кузнечик» основан не на сети Фейстеля, а на SP-сети (подстановочно-перестановочной сети). Количество раундов работы данного шифра равно 10.

Важным отличием ГОСТ Р 34.12-2015 от ГОСТ 28147-89 является то, что в новом стандарте определены лишь базовые блочные шифры и не определены режимы их работы. Здесь под базовым блочным шифром понимается шифр, реализующий при каждом фиксированном значении ключа одно обратимое отображение множества блоков открытого текста фиксированной длины в блоки шифртекста такой же длины. Поэтому одновременно со стандартом ГОСТ Р 34.12-2015 был введен дополняющий его стандарт, определяющий режимы работы блочных шифров.

2.4 ГОСТ Р 34.13-2015

Данный стандарт описан в документе «ГОСТ Р 34.13-2015. Информационная технология. Криптографическая защита информации. Режимы работы блочных шифров» и определяет следующие режимы работы алгоритмов блочного шифрования:

25 режим простой замены (Electronic Codebook, ECB);

26 режим гаммирования (Counter, CTR);

27 режим гаммирования с обратной связью по выходу (Output Feedback, OFB);

28 режим простой замены с зацеплением (Cipher Block Chaining, CBC);

29 режим гаммирования с обратной связью по шифртексту (Cipher Feedback, CFB);

30 режим выработки имитовставки (Message Authentication Code algorithm).

В режиме простой замены блоки открытого текста последовательно и независимо друг от друга преобразуются в блоки шифртекста. Другими словами, открытый текст разбивается на блоки, и к каждому блоку применяется базовый блочный шифр.

Режим гаммирования, определенный в ГОСТ Р 34.13-2015, очень похож на режим гаммирования, определенный в ГОСТ 28147-89. Зашифрование в режиме

гаммирования заключается в покомпонентном сложении открытого текста с гаммой шифра, которая вырабатывается путем зашифрования последовательности значений счетчика базовым алгоритмом блочного шифрования с последующим усечением. Начальное значение счетчика зависит от синхропосылки (в терминологии нового стандарта – вектора инициализации), каждое последующее значение увеличивается на единицу. Таким образом, в режиме гаммирования значение очередного блока гаммы зависит от номера соответствующего блока открытого текста.

В режиме гаммирования с обратной связью по выходу используется регистр сдвига, начальным заполнением которого является значение синхропосылки. В ходе выработки гаммы часть разрядов регистра сдвига зашифровывается с помощью базового блочного шифра и полученное значение после усечения используется в качестве блока гаммы. При этом заполнение регистра сдвига сдвигается на количество использованных разрядов, а на освободившиеся позиции записывается выходное значение базового блочного шифра. Таким образом, в режиме гаммирования с обратной связью по выходу в формировании очередного блока гаммы участвуют предыдущие блоки гаммы.

В режиме простой замены с зацеплением также используется регистр сдвига. В данном режиме очередной блок шифртекста получается путем зашифрования результата покомпонентного сложения значения очередного блока открытого текста со значением n разрядов регистра сдвига с большими номерами. Затем регистр сдвигается на один блок в сторону разрядов с большими номерами. В разряды с меньшими номерами записывается значение блока шифртекста. Другими словами, каждый блок открытого текста перед зашифрованием сцепляется (с помощью поразрядного сложения по модулю 2) с блоком, сформированным из предшествующего шифртекста. Само зашифрование блока открытого текста осуществляется с помощью базового блочного шифра.

Режим гаммирования с обратной связью по шифртексту похож на режим гаммирования с обратной связью по выходу. Отличие заключается в том, что в регистр сдвига записываются блоки шифртекста вместо блоков гаммы. В результате в режиме гаммирования с обратной связью по шифртексту в формировании очередного блока гаммы участвуют предыдущие блоки шифртекста.

Режим выработки имитовставки мало отличается от аналогичного режима, определенного в ГОСТ 28147-89, он имеет то же самое предназначение и схожее устройство.

3) Программная реализация.

Краткие выводы о проделанной работе.

I. Данный код написан на языке программирования Python и реализует все режимы шифрования режиме с использованием блочного шифра Кузнечик. Исходный код полученной программы размещен на Github [6]

A. Реализация режим шифрования CBC:

CBC (Cipher Block Chaining) - это режим блочного шифрования, используемый для шифрования сообщений переменной длины. В этом режиме каждый блок открытого текста комбинируется с предыдущим зашифрованным блоком перед шифрованием. Это создает зависимость между блоками, что делает шифрование более надежным и устойчивым к атакам. Однако для начала шифрования требуется инициализационный вектор (IV), который должен быть случайным и уникальным для каждого сообщения. Режим CBC обеспечивает конфиденциальность данных, но не обеспечивает аутентификацию, поэтому обычно используется совместно с методами аутентификации, такими как HMAC (Hash-based Message Authentication Code), для обеспечения целостности и подлинности данных.

1) **Функция «xor_bytes(a, b)»:**

- Эта функция выполняет операцию XOR над двумя байтовыми строками.
- Принимает два параметра a и b, которые представляют две байтовые строки.
- Итерируется по байтам в каждой строке, выполняет операцию XOR над соответствующими байтами и добавляет результат в результирующую строку.

```
def xor_bytes(a, b):
    # Функция для выполнения операции XOR над двумя байтовыми строками
    result = b''
    for i in range(min(len(a), len(b))):
        result += bytes([a[i] ^ b[i]])
```

```
return result
```

2) **Функция «pad_message(message, block_size)» :**

- Функция добавляет отступы к сообщению до размера блока.
- Принимает сообщение message и размер блока block_size.
- Вычисляет длину отступа, чтобы дополнить сообщение до размера блока, создает отступ соответствующей длины и добавляет его к сообщению.

```
def pad_message(message, block_size):
    # Добавление отступов к сообщению до размера блока
    padding_length = block_size - (len(message) % block_size)
    padding = bytes([padding_length]) * padding_length
    return message + padding
```

3) **Функция «unpad_message(padded_message)»**:

- Эта функция удаляет добавленные отступы из сообщения.
- Принимает зашифрованное сообщение с добавленными отступами padded_message.
- Извлекает длину отступа из последнего байта сообщения и удаляет этот отступ из сообщения.

```
def unpad_message(padded_message):
    # Удаление добавленных отступов из сообщения
    padding_length = padded_message[-1]
    return padded_message[:-padding_length]
```

4) **Функция «encrypt_cbc(input_file, output_file, key, iv)» :**

- Шифрует текст из файла в режиме CBC.
- Принимает путь к входному файлу input_file, путь к выходному файлу output_file, ключ key и вектор инициализации iv.
- Считывает текст из входного файла, добавляет к нему отступы, разбивает на блоки и применяет операцию XOR между текущим блоком и предыдущим зашифрованным блоком. Затем каждый блок шифруется с использованием Кузнечика и добавляется к выходной строке.

```
def encrypt_cbc(input_file, output_file, key, iv):
    # Шифрование текста из файла в режиме input_file
    block_size = len(key)
    with open(input_file, 'rb') as file:
        plaintext = file.read()
    plaintext = pad_message(plaintext, block_size)
    ciphertext = b''
    previous_block = iv
    for i in range(0, len(plaintext), block_size):
        block = plaintext[i:i+block_size]
        block = xor_bytes(block, previous_block)
        encrypted_block = int.to_bytes(kuznyechik_encrypt(int.from_bytes(block, byteorder='big'), int.from_bytes(key, byteorder='big')), 16, byteorder='big')
        ciphertext += encrypted_block
        previous_block = encrypted_block
    with open(output_file, 'wb') as file:
        file.write(ciphertext)
```

5) **Функция «decrypt_cbc(input_file, output_file, key, iv)» :**

- Расшифровывает текст из зашифрованного файла, зашифрованного в режиме CBC.

- Принимает путь к зашифрованному файлу input_file, путь к файлу для расшифрованного текста output_file, ключ key и вектор инициализации iv.
- Считывает зашифрованный текст из файла, разбивает его на блоки и расшифровывает каждый блок с использованием Кузнечика. Затем применяется операция XOR между текущим расшифрованным блоком и предыдущим зашифрованным блоком. После расшифровки удаляет добавленные отступы из результата и записывает его в файл.

```
def decrypt_cbc(input_file, output_file, key, iv):
    # Расшифрование текста из файла, зашифрованного в режиме input_file
    block_size = len(key)
    with open(input_file, 'rb') as file:
        ciphertext = file.read()
    plaintext = b''
    previous_block = iv
    for i in range(0, len(ciphertext), block_size):
        block = ciphertext[i:i+block_size]
        decrypted_block = int.to_bytes(kuznyechik_decrypt(int.from_bytes(block, byteorder='big'), int.from_bytes(key, byteorder='big')), 16, byteorder='big')
        decrypted_block = xor_bytes(decrypted_block, previous_block)
        plaintext += decrypted_block
        previous_block = block
    plaintext = unpad_message(plaintext)
    with open(output_file, 'wb') as file:
        file.write(plaintext)
```

B. Реализация режим шифрования CFB:

CFB (Cipher Feedback) - это режим блочного шифрования, который позволяет использовать блочные шифры для шифрования потока данных переменной длины. В этом режиме каждый блок шифруется отдельно и используется для шифрования

следующего блока открытого текста. После шифрования блока результат комбинируется с открытым текстом следующего блока перед шифрованием. Это позволяет преобразовать блочный шифр в поточный шифр, что полезно для работы с данными переменной длины. Режим CFB обеспечивает конфиденциальность данных и сопротивление некоторым атакам, но, как и CBC, не обеспечивает аутентификацию, поэтому часто используется с методами аутентификации для обеспечения целостности и подлинности данных.

1) **Функция «xor_bytes(a, b)» :**

- Эта функция выполняет операцию XOR над двумя байтовыми строками.
- Принимает два параметра a и b, которые представляют две байтовые строки.
- Итерируется по байтам в каждой строке, выполняет операцию XOR над соответствующими байтами и добавляет результат в результирующую строку.

```
def xor_bytes(a, b):
    # Функция для выполнения операции XOR над двумя байтовыми строками
    result = b''
    for i in range(min(len(a), len(b))):
        result += bytes([a[i] ^ b[i]])
    return result
```

2) **Функция «pad_message(message, block_size)» :**

- Эта функция добавляет отступы к сообщению до размера блока.
- Принимает сообщение message и размер блока block_size.
- Вычисляет длину отступа, чтобы дополнить сообщение до размера блока, создает отступ соответствующей длины и добавляет его к сообщению.

```
def pad_message(message, block_size):
    # Добавление отступов к сообщению до размера блока
    padding_length = block_size - (len(message) % block_size)
```

```
    padding = bytes([padding_length]) * padding_length
    return message + padding
```

3) **Функция «unpad_message(padded_message)» :**

- Эта функция удаляет добавленные отступы из сообщения.
- Получает в качестве входного параметра зашифрованное сообщение с отступами padded_message.
- Определяет длину добавленного отступа, читая последний байт padded_message, и затем удаляет соответствующее количество байтов от конца сообщения.

```
def unpad_message(padded_message):
    # Удаление добавленных отступов из сообщения
    padding_length = padded_message[-1]
    return padded_message[:-padding_length]
```

4) **Функция «encrypt_cfb(input_file, output_file, key, iv)» :**

- Эта функция шифрует текст из файла в режиме CFB (Cipher Feedback).
- Считывает данные из входного файла input_file.
- Дополняет сообщение отступами до размера блока.
- Инициализирует пустую строку для хранения зашифрованного текста.
- Использует предыдущий блок данных (iv в начале) и ключ для шифрования каждого блока сообщения.
- Выполняет операцию XOR между текущим блоком данных и зашифрованным предыдущим блоком.
- Записывает зашифрованные данные в выходной файл output_file.

```
def encrypt_cfb(input_file, output_file, key, iv):
    # Шифрование текста из файла в режиме CFB
    block_size = len(key)
    with open(input_file, 'rb') as file:
        plaintext = file.read()
    plaintext = pad_message(plaintext, block_size)
    ciphertext = b''
    previous_block = iv
    for i in range(0, len(plaintext), block_size):
        encrypted_block = kuznyechik_encrypt(int.from_bytes(previous_block, byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование предыдущего блока
        block = plaintext[i:i+block_size]
        encrypted_block = xor_bytes(block, int.to_bytes(encrypted_block, block_size, byteorder='big'))  # XOR с зашифрованным предыдущим блоком
        ciphertext += encrypted_block
        previous_block = encrypted_block[-block_size:]  # Последние block_size байтов зашифрованного блока
    with open(output_file, 'wb') as file:
        file.write(ciphertext)
```

5) **Функция «decrypt_cfb(input_file, output_file, key, iv)» :**

- Эта функция расшифровывает текст из файла, зашифрованного в режиме CFB.
- Считывает зашифрованные данные из входного файла input_file.
- Инициализирует пустую строку для хранения расшифрованного текста.
- Использует предыдущий блок данных (iv в начале) и ключ для расшифрования каждого блока сообщения.
- Выполняет операцию XOR между текущим блоком данных и зашифрованным предыдущим блоком.
- Записывает расшифрованные данные в выходной файл output_file.

```
def decrypt_cfb(input_file, output_file, key, iv):
    # Расшифрование текста из файла, зашифрованного в режиме CFB
    block_size = len(key)
    with open(input_file, 'rb') as file:
        ciphertext = file.read()
    plaintext = b''
    previous_block = iv
    for i in range(0, len(ciphertext), block_size):
        encrypted_block = kuznyechik_encrypt(int.from_bytes(previous_block, byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование предыдущего блока
        block = ciphertext[i:i+block_size]
        decrypted_block = xor_bytes(block, int.to_bytes(encrypted_block, block_size, byteorder='big'))  # XOR с зашифрованным предыдущим блоком
        plaintext += decrypted_block
        previous_block = block  # Зашифрованный текст становится предыдущим блоком для следующей итерации
    plaintext = unpad_message(plaintext)
    with open(output_file, 'wb') as file:
        file.write(plaintext)
```

C. Реализация режим шифрования ECB:

ECB (Electronic Codebook) - это один из простейших режимов блочного шифрования, где каждый блок открытого текста шифруется независимо друг от друга с использованием одного и того же ключа. Таким образом, один и тот же блок открытого текста всегда будет преобразовываться в один и тот же шифротекст. Это может привести к различным атакам, таким как анализ частотности и повторное использование шифротекста, что делает его менее безопасным в сравнении с другими режимами. ECB обычно не рекомендуется для шифрования длинных данных или критически важной информации.

1) **Функция «pad_message(message, block_size)» :**

- Эта функция добавляет отступы к сообщению до размера блока.
- Принимает сообщение message и размер блока block_size.
- Вычисляет длину отступа, чтобы дополнить сообщение до размера блока, создает отступ соответствующей длины и добавляет его к сообщению.

```
def pad_message(message, block_size):
    # Добавление отступов к сообщению до размера блока
    padding_length = block_size - (len(message) % block_size)
    padding = bytes([padding_length]) * padding_length
    return message + padding
```

2) **Функция «unpad_message(padded_message)» :**

- Эта функция удаляет добавленные отступы из сообщения.
- Получает в качестве входного параметра зашифрованное сообщение с отступами padded_message.
- Определяет длину добавленного отступа, читая последний байт padded_message, и затем удаляет соответствующее количество байтов от конца сообщения.

```
def unpad_message(padded_message):
    # Удаление добавленных отступов из сообщения
    padding_length = padded_message[-1]
    return padded_message[:-padding_length]
```

3) **Функция «encrypt_ecb(input_file, output_file, key)» :**

- Эта функция шифрует текст из файла в режиме ECB (Electronic Codebook).
- Считывает данные из входного файла input_file.
- Дополняет сообщение отступами до размера блока.
- Инициализирует пустую строку для хранения зашифрованного текста.
- Шифрует каждый блок данных сообщения с использованием ключа.
- Записывает зашифрованные данные в выходной файл output_file.

```
def encrypt_ecb(input_file, output_file, key):
```

```
    # Шифрование текста из файла в режиме ECB
    block_size = len(key)
    with open(input_file, 'rb') as file:
        plaintext = file.read()
    plaintext = pad_message(plaintext, block_size)
    ciphertext = b''
    for i in range(0, len(plaintext), block_size):
        block = plaintext[i:i+block_size]
        encrypted_block = int.to_bytes(kuznyechik_encrypt(int.from_bytes(block,
byteorder='big'), int.from_bytes(key, byteorder='big')), 16, byteorder='big')
        ciphertext += encrypted_block
    with open(output_file, 'wb') as file:
        file.write(ciphertext)
```

4) Функция «decrypt_ecb(input_file, output_file, key)» :

- Эта функция расшифровывает текст из файла, зашифрованного в режиме ECB.
- Считывает зашифрованные данные из входного файла input_file.
- Инициализирует пустую строку для хранения расшифрованного текста.
- Расшифровывает каждый блок данных сообщения с использованием ключа.
- Удаляет добавленные отступы из расшифрованного текста.
- Записывает расшифрованные данные в выходной файл output_file.

```
def decrypt_ecb(input_file, output_file, key):
    # Расшифрование текста из файла, зашифрованного в режиме ECB
    block_size = len(key)
    with open(input_file, 'rb') as file:
        ciphertext = file.read()
    plaintext = b''
    for i in range(0, len(ciphertext), block_size):
        block = ciphertext[i:i+block_size]
        decrypted_block = int.to_bytes(kuznyechik_decrypt(int.from_bytes(block,
byteorder='big'), int.from_bytes(key, byteorder='big')), 16, byteorder='big')
```

```
        plaintext += decrypted_block
    plaintext = unpad_message(plaintext)
    with open(output_file, 'wb') as file:
        file.write(plaintext)
```

D. Реализация режим шифрования OFB:

OFB (Output Feedback) - это режим блочного шифрования, который преобразует блочный шифр в поточный шифр. В этом режиме блок шифрования используется для генерации псевдослучайной последовательности (потока), которая затем комбинируется с открытым текстом для получения шифротекста. Этот шифротекст используется также для генерации следующей порции псевдослучайной последовательности. Поскольку каждый блок зависит только от предыдущего состояния, режим OFB поддерживает операцию в режиме с обратной связью, что обеспечивает хорошую диффузию. Кроме того, ошибки в передаче не распространяются на последующие блоки. Однако, подобно другим режимам без аутентификации, OFB не обеспечивает аутентификацию, поэтому может быть подвержен атакам внедрения данных.

1) **Функция «xor_bytes(a, b)» :**

- Эта функция выполняет операцию XOR над двумя байтовыми строками.
- Принимает два параметра a и b, которые представляют две байтовые строки.
- Итерируется по байтам в каждой строке, выполняет операцию XOR над соответствующими байтами и добавляет результат в результирующую строку.

```
def xor_bytes(a, b):
    # Функция для выполнения операции XOR над двумя байтовыми строками
    result = b''
    for i in range(min(len(a), len(b))):
        result += bytes([a[i] ^ b[i]])
    return result
```

2) **Функция «pad_message(message, block_size)» :**

- Эта функция добавляет отступы к сообщению до размера блока.
- Принимает сообщение message и размер блока block_size.
- Вычисляет длину отступа, чтобы дополнить сообщение до размера блока, создает отступ соответствующей длины и добавляет его к сообщению.

```
def pad_message(message, block_size):
    # Добавление отступов к сообщению до размера блока
    padding_length = block_size - (len(message) % block_size)
    padding = bytes([padding_length]) * padding_length
    return message + padding
```

3) **Функция «unpad_message(padded_message)» :**

- Эта функция удаляет добавленные отступы из сообщения.
- Получает в качестве входного параметра зашифрованное сообщение с отступами padded_message.
- Определяет длину добавленного отступа, читая последний байт padded_message, и затем удаляет соответствующее количество байтов от конца сообщения.

```
def unpad_message(padded_message):
    # Удаление добавленных отступов из сообщения
    padding_length = padded_message[-1]
    return padded_message[:-padding_length]
```

4) **Функция «encrypt_ofb(input_file, output_file, key, iv)» :**

- Эта функция шифрует текст из файла в режиме OFB (Output Feedback).
- Считывает данные из входного файла input_file.
- Дополняет сообщение отступами до размера блока.
- Инициализирует пустую строку для хранения зашифрованного текста.
- Использует предыдущий блок данных (iv в начале) и ключ для шифрования каждого блока сообщения.
- Выполняет операцию XOR между текущим блоком данных и зашифрованным предыдущим блоком.
- Записывает зашифрованные данные в выходной файл output_file.

```
def encrypt_ofb(input_file, output_file, key, iv):
    # Шифрование текста из файла в режиме OFB
    block_size = len(key)
    with open(input_file, 'rb') as file:
        plaintext = file.read()
    plaintext = pad_message(plaintext, block_size)
    ciphertext = b''
    previous_block = iv
    for i in range(0, len(plaintext), block_size):
        encrypted_block = kuznyechik_encrypt(int.from_bytes(previous_block, byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование предыдущего блока
        block = plaintext[i:i+block_size]
        encrypted_block = xor_bytes(block, int.to_bytes(encrypted_block, block_size, byteorder='big'))  # XOR с зашифрованным предыдущим блоком
        ciphertext += encrypted_block
        previous_block = encrypted_block  # Предыдущий блок для следующей итерации
    with open(output_file, 'wb') as file:
        file.write(ciphertext)
```

5) Функция «decrypt_ofb(input_file, output_file, key, iv)» :

- Эта функция расшифровывает текст из файла, зашифрованного в режиме OFB.
- Считывает зашифрованные данные из входного файла input_file.
- Инициализирует пустую строку для хранения расшифрованного текста.
- Использует предыдущий блок данных (iv в начале) и ключ для расшифрования каждого блока сообщения.
- Выполняет операцию XOR между текущим блоком данных и зашифрованным предыдущим блоком.
- Записывает расшифрованные данные в выходной файл output_file.

```
def decrypt_ofb(input_file, output_file, key, iv):
    # Расшифрование текста из файла, зашифрованного в режиме OFB
    block_size = len(key)
```

```
    with open(input_file, 'rb') as file:
        ciphertext = file.read()
    plaintext = b''
    previous_block = iv
    for i in range(0, len(ciphertext), block_size):
        encrypted_block = kuznyechik_encrypt(int.from_bytes(previous_block,
byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование предыдущего
блока
        block = ciphertext[i:i+block_size]
        decrypted_block = xor_bytes(block, int.to_bytes(encrypted_block, block_size,
byteorder='big'))  # XOR с зашифрованным предыдущим блоком
        plaintext += decrypted_block
        previous_block = encrypted_block  # Предыдущий блок для следующей итерации
    plaintext = unpad_message(plaintext)
    with open(output_file, 'wb') as file:
        file.write(plaintext)
```

E. Реализация режим шифрования CTR:

Режим Counter (CTR) - это режим блочного шифрования, который превращает блочный шифр в поточный шифр. Он работает путем преобразования ключа и инициализационного вектора в последовательность счётчиков, которые затем комбинируются с открытым текстом для получения шифротекста. Каждый блок открытого текста обрабатывается с помощью шифра с использованием уникального счётчика, что делает возможным параллельное шифрование. CTR обладает высокой производительностью и хорошей распределенной архитектурой, что позволяет ему быть эффективным в многопоточных или распределенных средах. Он также обеспечивает детерминированную криптографию и отсутствие распространения ошибок, однако не предоставляет аутентификацию данных и подвержен атакам внедрения данных.

Nonce (Number used once) — это случайное или уникальное значение, которое используется только один раз в определенном контексте или операции. В

криптографии попсе часто используется вместе с ключом для генерации уникальных шифрованных данных, особенно при использовании режимов шифрования, которые требуют инициализации вектора.

1) **Функция «xor_bytes(a, b)» :**

- Эта функция выполняет операцию XOR над двумя байтовыми строками.
- Принимает два параметра a и b, которые представляют две байтовые строки.
- Итерируется по байтам в каждой строке, выполняет операцию XOR над соответствующими байтами и добавляет результат в результирующую строку.

```
def xor_bytes(a, b):
    # Функция для выполнения операции XOR над двумя байтовыми строками
    result = b''
    for i in range(min(len(a), len(b))):
        result += bytes([a[i] ^ b[i]])
    return result
```

2) **Функция «pad_message(message, block_size)» :**

- Эта функция добавляет отступы к сообщению до размера блока.
- Принимает сообщение message и размер блока block_size.
- Вычисляет длину отступа, чтобы дополнить сообщение до размера блока, создает отступ соответствующей длины и добавляет его к сообщению.

```
def pad_message(message, block_size):
    # Добавление отступов к сообщению до размера блока
    padding_length = block_size - (len(message) % block_size)
    padding = bytes([padding_length]) * padding_length
    return message + padding
```

3) **Функция «unpad_message(padded_message)» :**

- Эта функция удаляет добавленные отступы из сообщения.

- Получает в качестве входного параметра зашифрованное сообщение с отступами padded_message.
- Определяет длину добавленного отступа, читая последний байт padded_message, и затем удаляет соответствующее количество байтов от конца сообщения.

```
def unpad_message(padded_message):
    # Удаление добавленных отступов из сообщения
    padding_length = padded_message[-1]
    return padded_message[:-padding_length]
```

4) Функция «encrypt_ctr(input_file, output_file, key, nonce)» :

- Эта функция шифрует текст из файла в режиме CTR (Counter).
- Считывает данные из входного файла input_file.
- Дополняет сообщение отступами до размера блока.
- Инициализирует пустую строку для хранения зашифрованного текста.
- Генерирует счетчик, объединяя значение Nonce и текущее значение счетчика, и шифрует его.
- Использует зашифрованный счетчик для генерации псевдослучайного потока ключей.
- Производит операцию XOR между каждым блоком данных сообщения и соответствующими байтами из псевдослучайного потока ключей.
- Записывает зашифрованные данные в выходной файл output_file.

```
def encrypt_ctr(input_file, output_file, key, nonce):
    # Шифрование текста из файла в режиме CTR
    block_size = len(key)
    with open(input_file, 'rb') as file:
        plaintext = file.read()
    plaintext = pad_message(plaintext, block_size)
    ciphertext = b''
    counter = 0
    for i in range(0, len(plaintext), block_size):
        counter_block = nonce + counter.to_bytes(block_size // 2, byteorder='big')  #
Генерация счетчика
```

```
        encrypted_counter = kuznyechik_encrypt(int.from_bytes(counter_block,
byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование счетчика
        block = plaintext[i:i+block_size]
        encrypted_block = xor_bytes(block, int.to_bytes(encrypted_counter, block_size,
byteorder='big'))  # XOR с результатом шифрования счетчика
        ciphertext += encrypted_block
        counter += 1
    with open(output_file, 'wb') as file:
        file.write(ciphertext)
```

5) **Функция «decrypt_ctr(input_file, output_file, key, nonce)» :**

- Эта функция расшифровывает текст из файла, зашифрованного в режиме CTR.
- Считывает зашифрованные данные из входного файла input_file.
- Инициализирует пустую строку для хранения расшифрованного текста.
- Генерирует счетчик, объединяя значение Nonce и текущее значение счетчика, и шифрует его.
- Использует зашифрованный счетчик для генерации псевдослучайного потока ключей.
- Производит операцию XOR между каждым блоком зашифрованных данных и соответствующими байтами из псевдослучайного потока ключей.
- Удаляет добавленные отступы из расшифрованного текста.
- Записывает расшифрованные данные в выходной файл output_file.

```
def decrypt_ctr(input_file, output_file, key, nonce):
    # Расшифрование текста из файла, зашифрованного в режиме CTR
    block_size = len(key)
    with open(input_file, 'rb') as file:
        ciphertext = file.read()
    plaintext = b''
    counter = 0
    for i in range(0, len(ciphertext), block_size):
        counter_block = nonce + counter.to_bytes(block_size // 2, byteorder='big')  #
Генерация счетчика
```

```
        encrypted_counter = kuznyechik_encrypt(int.from_bytes(counter_block,
byteorder='big'), int.from_bytes(key, byteorder='big'))  # Шифрование счетчика
        block = ciphertext[i:i+block_size]
        decrypted_block = xor_bytes(block, int.to_bytes(encrypted_counter, block_size,
byteorder='big'))  # XOR с результатом шифрования счетчика
        plaintext += decrypted_block
        counter += 1
    plaintext = unpad_message(plaintext)
    with open(output_file, 'wb') as file:
        file.write(plaintext)
```

II. Реализация ГОСТ Р 34.10-2012 блочного шифра «Кузнечик»:

1) Функции «S(x) и S_inv(x)» :

- Эти функции представляют S-блок, который применяется к каждому байту входного значения x.
- S(x) применяет S-блок к x, а S_inv(x) выполняет обратное преобразование S-блока.

```
# Функция S: входные данные x и выходные данные y имеют размер 128 бит
def S(x):
    y = 0
    for i in reversed(range(16)):
        y <<= 8
        y ^= pi[(x >> (8 * i)) & 0xff]  # Применяем S-блок к каждому байту входного
значения x
    return y


# Обратная функция S: входные данные x и выходные данные y имеют размер 128 бит
def S_inv(x):
    y = 0
```

```
    for i in reversed(range(16)):
        y <<= 8
        y ^= pi_inv[(x >> (8 * i)) & 0xff]  # Обратное преобразование S-блока
    return y
```

2) Функции «multiply_ints_as_polynomials(x, y) и mod_int_as_polynomial(x, m)» :

- Эти функции используются для умножения двух чисел как полиномов и нахождения остатка от деления.
- multiply_ints_as_polynomials(x, y) умножает числа x и y как полиномы, используя метод сложения.
- mod_int_as_polynomial(x, m) находит остаток от деления x на m как полиномы.

```
# Умножение двух чисел как полиномов
def multiply_ints_as_polynomials(x, y):
    if x == 0 or y == 0:
        return 0
    z = 0
    while x != 0:
        if x & 1 == 1:
            z ^= y  # Умножение полиномов методом сложения
        y <<= 1  # Сдвиг влево на 1 бит
        x >>= 1  # Сдвиг вправо на 1 бит
    return z


# Функция для определения количества бит, необходимых для хранения числа x
def number_bits(x):
    nb = 0
```

```
    while x != 0:
        nb += 1
        x >>= 1  # Сдвиг вправо на 1 бит
    return nb


# Функция для нахождения остатка от деления x на m
def mod_int_as_polynomial(x, m):
    nbm = number_bits(m)
    while True:
        nbx = number_bits(x)
        if nbx < nbm:
            return x
        mshift = m << (nbx - nbm)  # Сдвиг m на (nbx - nbm) бит влево
        x ^= mshift  # Выполняем операцию XOR для нахождения остатка
```

3) Функции «kuznyechik_multiplication(x, y)» :

- Эта функция представляет операцию умножения в алгоритме Kuznyechik.
- Она использует функции multiply_ints_as_polynomials и mod_int_as_polynomial для выполнения умножения и возвращения результата с учетом полиномиального модуля.

```
# Умножение двух чисел, входные данные x и y имеют размер 8 бит, выходное значение 8 бит
def kuznyechik_multiplication(x, y):
    z = multiply_ints_as_polynomials(x, y)
    m = int('111000011', 2)  # Полиномиальный модуль для операции умножения
    return mod_int_as_polynomial(z, m)
```

4) **Функция «kuznyechik_linear_functional(x)» :**

- Эта функция представляет линейно-функциональное преобразование, используемое в алгоритме Kuznyechik.
- Применяет линейное преобразование к каждому байту входного значения x.

```
# Линейно-функциональная функция

def kuznyechik_linear_functional(x):

    C = [148, 32, 133, 16, 194, 192, 1, 251, 1, 192, 194, 16, 133, 32, 148, 1]

    y = 0

    while x != 0:

        y ^= kuznyechik_multiplication(x & 0xff, C.pop())  # Применяем линейное
преобразование

        x >>= 8  # Сдвигаем входное значение на 8 бит вправо

    return y
```

5) **Функции «R(x) и R_inv(x)» :**

- Эти функции представляют операции циклического сдвига и XOR, используемые в алгоритме Kuznyechik.
- R(x) выполняет сдвиг и XOR, а R_inv(x) выполняет обратные операции.

```
# Функция R

def R(x):

    a = kuznyechik_linear_functional(x)

    return (a << 8 * 15) ^ (x >> 8)  # Выполняем циклический сдвиг и XOR

# Обратная функция R

def R_inv(x):

    a = x >> 15 * 8

    x = (x << 8) & (2 ** 128 - 1)

    b = kuznyechik_linear_functional(x ^ a)
```

```
    return x ^ b
```

6) **Функции «L(x) и L_inv(x)» :**

- Эти функции представляют композицию операций циклического сдвига и линейно-функционального преобразования.
- L(x) применяет операции R(x) несколько раз, а L_inv(x) выполняет обратные операции.

```
# Функция L
def L(x):
    for _ in range(16):
        x = R(x)
    return x
# Обратная функция L
def L_inv(x):
    for _ in range(16):
        x = R_inv(x)
    return x
```

7) **Функция «kuznyechik_key_schedule(k)» :**

- Эта функция генерирует раундовые ключи на основе основного ключа k, используемого в алгоритме Kuznyechik.

```
# Генерация ключей
def kuznyechik_key_schedule(k):
    keys = []
    a = k >> 128
    b = k & (2 ** 128 - 1)
    keys.append(a)
```

```
    keys.append(b)
    for i in range(4):
        for j in range(8):
            c = L(8 * i + j + 1)
            (a, b) = (L(S(a ^ c)) ^ b, a)
        keys.append(a)
        keys.append(b)
    return keys
```

8) Функции «kuznyechik_encrypt(x, k) и kuznyechik_decrypt(x, k)» :

- Эти функции представляют процессы шифрования и расшифрования в алгоритме Kuznyechik.

```
# Шифрование
def kuznyechik_encrypt(x, k):
    keys = kuznyechik_key_schedule(k)
    for round in range(9):
        x = L(S(x ^ keys[round]))  # Применяем операции L и S
    return x ^ keys[-1]  # Выполняем последний XOR с ключом

# Расшифрование
def kuznyechik_decrypt(x, k):
    keys = kuznyechik_key_schedule(k)
    keys.reverse()
    for round in range(9):
        x = S_inv(L_inv(x ^ keys[round]))  # Обратные операции L и S
    return x ^ keys[-1]  # Выполняем последний XOR с ключом
```

9) **Функции «read_input(file_name) и write_output(file_name, PT, CT, DT, result)» :**

- Эти функции используются для чтения входных данных из файла и записи результатов в файл.

```
# Чтение входных данных из файла
def read_input(file_name):
    with open(file_name, 'r') as file:
        lines = file.readlines()
        PT = int(lines[0].strip(), 16)  # Оригинальный открытый текст
        k = int(lines[1].strip(), 16)  # Ключ
    return PT, k


# Запись результата в файл
def write_output(file_name, PT, CT, DT, result):
    with open(file_name, 'w') as file:
        file.write("Исходный текст (PT): {}\n".format(hex(PT)))
        file.write("Зашифрованный текст (CT): {}\n".format(hex(CT)))
        file.write("Расшифрованный текст (DT): {}\n".format(hex(DT)))
        file.write(result)
```

4) Результат работы программы.

На вход программы подается файл input.txt, содержание которого представлено на рисунке 1.

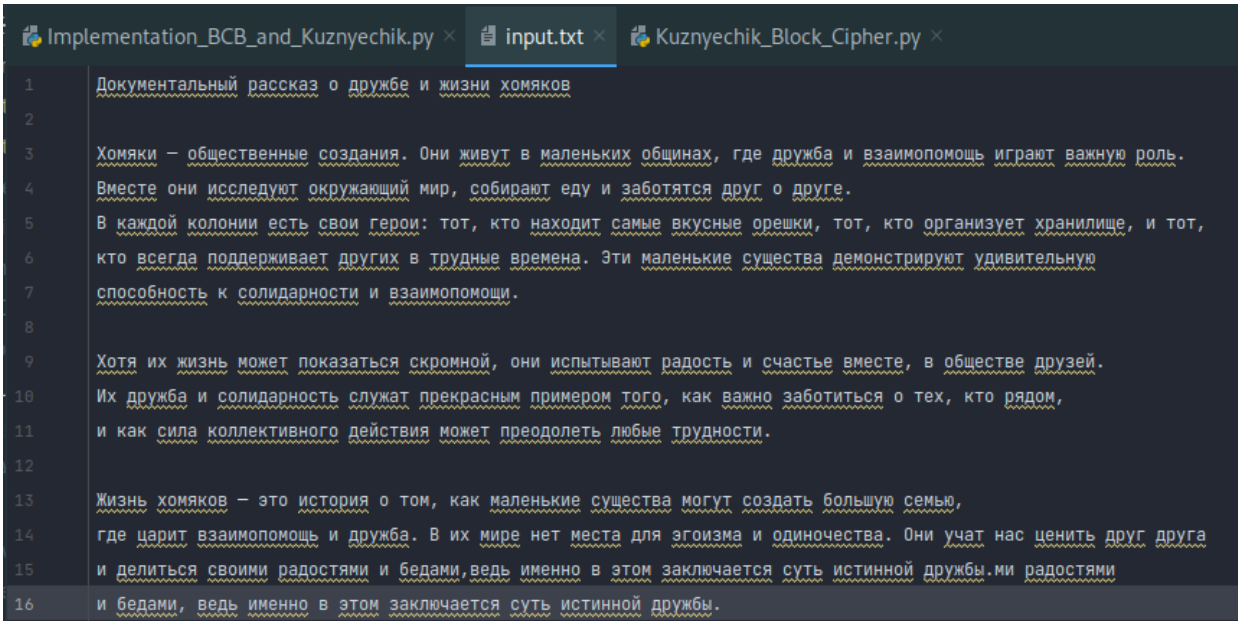

Рисунок 1

После того как код закончил шифрование, он создает зашифрованный файл encrypted.txt, содержимое которого представлено на рисунке 2.

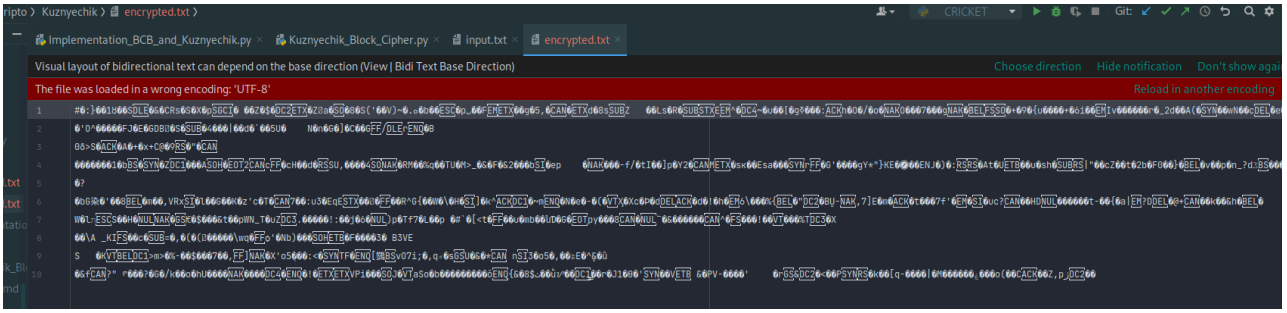

Рисунок 2

А по окончании расшифрования код создает расшифрованный файл decrypted.txt, содержимое которого представлена на рисунке 3.

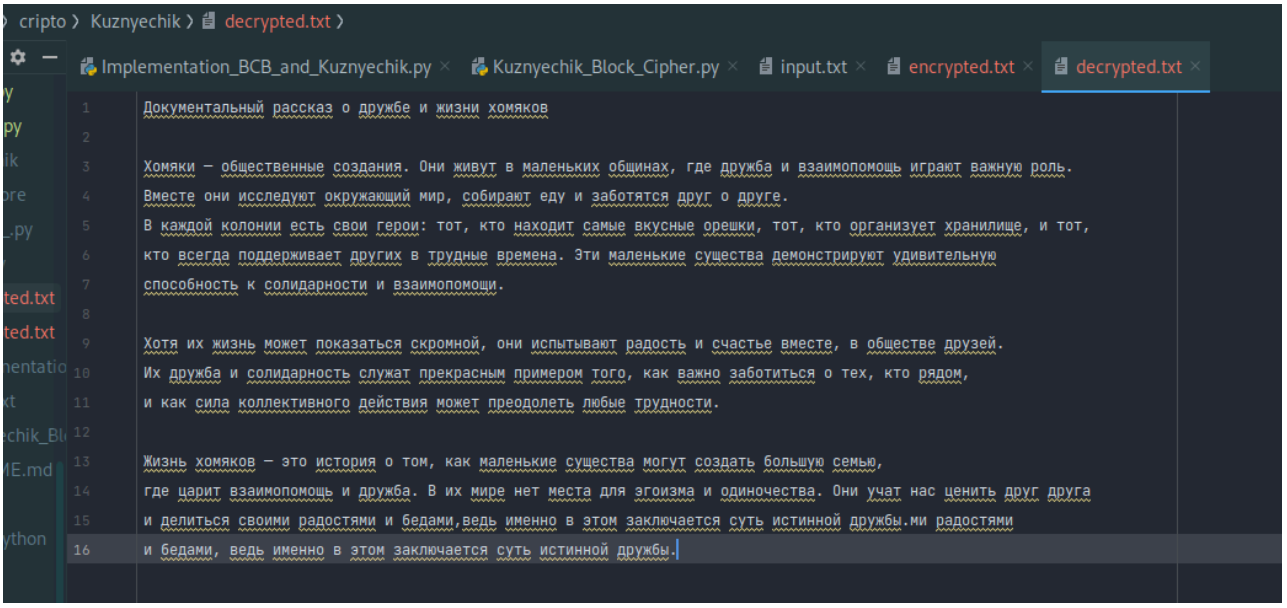

Рисунок 3

Используя специализированный сайт по поиску сравнению документов между собой compare.embedika.ru [7], сравним два файла decrypted.txt и input.txt, что убедиться о неизвестности данных до шифрования и после (Рисунок 4).

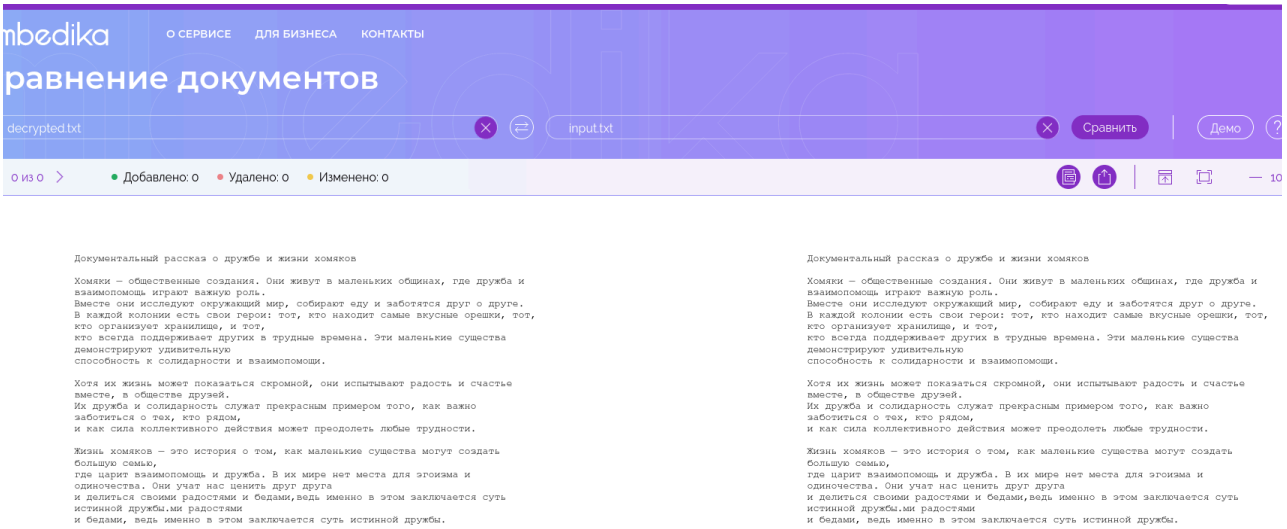

Рисунок 4

Также была реализована программная побитовая проверка сходства файла с исходным текстом и текстом после расшифрования (Рисунок 5).

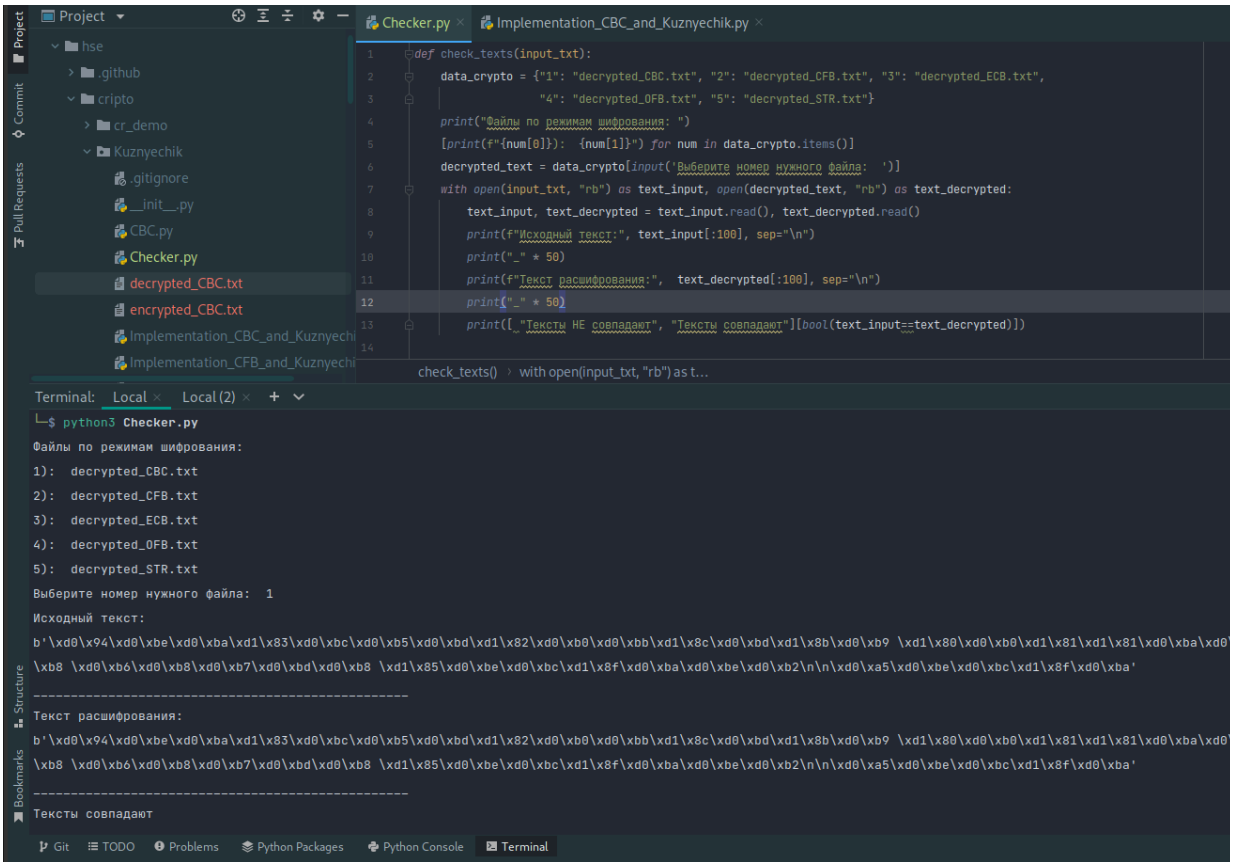

Рисунок 5

5 Выводы о проделанной работе.

При выполнении задания было реализовано шифрование и расшифрование во всех режимах с использованием блочного шифра Кузнечик. Полученный опыт и практическая работа помогли приобрести новые навыки и понимание в области криптографии и программирования.

Во-первых, было более глубоко изучено симметричное блочное шифрование и его применение на практике, включая основные концепции работы всех режимов шифрования.

Во-вторых, разработаны и отлажены сложные алгоритмы шифрования и дешифрования на Python с применением блочного шифра Кузнечик, что позволило эффективно манипулировать байтовыми данными и обеспечить правильную обработку данных на различных этапах шифрования и расшифрования.

Третьим важным аспектом было управление файлами в Python: чтение данных из файлов, обработка и запись результатов в другие файлы, что окажется полезным при работе с файловой системой в различных приложениях и проектах.

6 Список использованных источников.

1. GOST R 34.12-2015: Block Cipher "Kuznyechik. – URL: GOST R 34.12-2015: Block Cipher "Kuznyechik.

2. ГОСТ Р 34.12 2015 . – URL: https://tc26.ru/standard/gost/GOST_R_3412-2015.pdf?ysclid=lu5xn659fg361666475.

3. JacksonInfoSec. The Kuznyechik Block Cipher. – URL: https://github.com/jacksoninfosec/kuznyechik/blob/main/kuznyechik.py

4. JacksonInfoSec. The Rijndael S-Box. – URL: https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf

5. Лекторий МФТИ. Защита информации. Блочные шифры.– URL: https://mipt.lectoriy.ru/lecture/CompTech-InforSecur-L03-Vladim-130921.01?ysclid=lulkp6g6az855487084

6. AES Documents – URL: https://csrc.nist.gov/files/pubs/fips/197/final/docs/fips-197.pdf

7. Исходный код разработанной программы. – URL: https://github.com/Kasimov-Aleksey/Kuznyechik

8. compare.embedika.ru. – URL: https://compare.embedika.ru/?ysclid=lu612a04zw598013503