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 № 4

по дисциплине «Криптографические методы защиты информации»

Современные симметричные шифры

|  |
| --- |
| Студент гр. \_\_\_\_  \_\_\_\_\_\_\_\_\_\_\_\_\_ В. С. Новиков  «\_\_\_» \_\_\_\_\_\_\_\_\_ 2025 г. |
| Руководитель  Заведующий кафедрой информационной безопасности киберфизических систем  канд. техн. наук, доцент  \_\_\_\_\_\_\_\_\_\_\_\_\_\_О.О. Евсютин  «\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г. |

Москва 2025

**СОДЕРЖАНИЕ**

[1 Задание на практическую работу 3](#_Toc191996594)

[2 Теоретическая часть 4](#_Toc191996595)

[2.1 ГОСТ Р 34.12-2015 4](#_Toc191996596)

[2.2 Шифр «Магма» 4](#_Toc191996597)

[3 Программный код и описание вариант шифра «Магма» 5](#_Toc191996598)

[3.1 Описание 5](#_Toc191996599)

[3.2 Код 5](#_Toc191996600)

[3.3 Запуск 8](#_Toc191996601)

[3.4 Пример работы 8](#_Toc191996602)

[4 ПРИЛОЖЕНИЕ А. 9](#_Toc191996603)

# Задание на практическую работу

Целью данной работы является приобретение навыков программной реализации современных алгоритмов симметричного шифрования.

Написать программную реализацию одного из перечисленных ниже симметричных шифров (по выбору студента). Реализация шифра должны быть выполнена студентом самостоятельно без использования готовых библиотечных функций, напрямую реализующих алгоритм шифрования или его отдельные этапы. Варианты шифров:

- Магма;

- Кузнечик;

- AES;

# Теоретическая часть

## 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:

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

- 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.

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

## Шифр «Магма»

* блочный шифр, 64 бита блок, 256 бит ключ;
* основа: сеть Фейстеля, 32 раунда;
* режим простой замены (ECB);
* используемые операции: сложение, сдвиг, подстановка.

# Программный код и описание варианта шифра «Магма»

## Описание

* язык: Python;
* функции генерация ключей, шифрование/расшифрование блока, обработка файла;
* особенности реализация без библиотек, работа с файлами поблочно;
* SBOX - фиксированная таблица замен из ГОСТ Р 34.12-2015. (ГОСТ Р 34.12-2015, пункт Приложение А (обязательное);
* generate\_round\_keys - генерирует 32 раундовых ключа из 256-битного ключа;
* G: функция преобразования — сложение по модулю 232232, циклический сдвиг на 11 бит и подстановка по таблице;
* encrypt\_block / decrypt\_block - шифрование и расшифрование одного блока (64 бита) с использованием сети Фейстеля;
* process\_file - читает входной файл поблочно, шифрует/расшифрует и записывает результат.

## Код

Ссылка на код: <https://github.com/vit81g/Cybersecurity_HSE/blob/main/HomeWorks/Cryptographic/practic/PW4v3.py>

import os # Импортируем модуль os для работы с файловой системой  
from typing import List # Импортируем List для аннотаций типов  
  
# Фиксированная таблица замен (S-box) из ГОСТ Р 34.12-2015, Приложение А  
# Это 8 строк по 16 значений (от 0 до 15), используется для подстановки 4-битовых кусков  
SBOX: List[List[int]] = [  
 [12, 4, 6, 2, 10, 5, 11, 9, 14, 8, 13, 7, 0, 3, 15, 1], # \(\pi\_0\): первая строка таблицы  
 [6, 8, 2, 3, 9, 10, 5, 12, 1, 11, 7, 13, 0, 4, 15, 14], # \(\pi\_1\): вторая строка таблицы  
 [7, 11, 5, 8, 12, 4, 2, 0, 14, 1, 3, 10, 9, 15, 6, 13], # \(\pi\_2\): третья строка таблицы  
 [13, 1, 7, 4, 11, 5, 0, 15, 3, 12, 14, 6, 9, 10, 2, 8], # \(\pi\_3\): четвёртая строка таблицы  
 [5, 10, 15, 12, 1, 13, 14, 11, 8, 3, 6, 0, 4, 7, 9, 2], # \(\pi\_4\): пятая строка таблицы  
 [14, 5, 0, 15, 13, 11, 3, 6, 9, 2, 12, 7, 1, 8, 10, 4], # \(\pi\_5\): шестая строка таблицы  
 [11, 13, 12, 3, 7, 14, 10, 5, 0, 9, 4, 15, 2, 8, 1, 6], # \(\pi\_6\): седьмая строка таблицы  
 [15, 12, 9, 7, 3, 0, 11, 4, 1, 14, 2, 13, 6, 10, 8, 5] # \(\pi\_7\): восьмая строка таблицы  
]  
  
  
def generate\_round\_keys(key: bytes) -> List[int]:  
 *"""  
 Генерирует 32 раундовых ключа из 256-битного исходного ключа для шифра Магма.  
 Аргументы: key (bytes): Исходный ключ длиной 256 бит (32 байта).  
 Возвращает: List[int]: Список из 32 раундовых ключей, каждый по 32 бита.  
 Обработка ошибок: ValueError: Если длина ключа не равна 32 байтам.  
 """* # Проверяем, что длина ключа ровно 256 бит (32 байта)  
 if len(key) != 32:  
 raise ValueError("Ключ должен быть длиной 256 бит (32 байта)")  
  
 round\_keys: List[int] = [] # Создаём пустой список для хранения 8 базовых ключей  
  
 # Делим исходный ключ на 8 частей по 4 байта (32 бита)  
 for i in range(0, 32, 4):  
 # Берем срез байтов от i до i+4 (например, 0-3, 4-7 и т.д.)  
 chunk: bytes = key[i:i + 4]  
 # Преобразуем 4 байта в 32-битовое целое число (big-endian — старшие байты слева)  
 k: int = int.from\_bytes(chunk, 'big')  
 # Добавляем ключ в список  
 round\_keys.append(k)  
  
 # Формируем расписание ключей согласно ГОСТ Р 34.12-2015:  
 # Первые 24 раунда — ключи 0-7 повторяются трижды (8 × 3 = 24)  
 # Последние 8 раундов — ключи 7-0 в обратном порядке  
 schedule: List[int] = round\_keys \* 3 + round\_keys[::-1]  
  
 # Возвращаем полный список из 32 ключей  
 return schedule  
  
  
def G(a: int, k: int) -> int:  
 *"""  
 Функция преобразования G в сети Фейстеля для шифра Магма.  
 Аргументы: a (int): Правая часть блока (32 бита); k (int): Раундовый ключ (32 бита).  
 Возвращает: int: Результат преобразования (32 бита).  
 """* # Складываем входное значение a и ключ k по модулю 2^32 (максимум 32-битного числа)  
 t: int = (a + k) % (2 \*\* 32)  
  
 # Выполняем циклический сдвиг влево на 11 бит:  
 # Сдвигаем t влево на 11 позиций  
 left\_shift: int = t << 11  
 # Сдвигаем t вправо на 21 позицию (32 - 11 = 21), чтобы взять старшие биты  
 right\_shift: int = t >> 21  
 # Объединяем результаты сдвигов побитовым ИЛИ и ограничиваем 32 битами  
 t = (left\_shift | right\_shift) & 0xFFFFFFFF  
  
 # Инициализируем переменную для результата подстановки  
 result: int = 0  
  
 # Применяем таблицу замен SBOX к каждому 4-битовому куску (нибблу)  
 for i in range(8):  
 # Извлекаем i-й 4-битовый кусок из t:  
 # Сдвигаем вправо на 4\*i позиций и берём младшие 4 бита (маска 0xF)  
 nibble: int = (t >> (4 \* i)) & 0xF  
 # Выполняем подстановку: выбираем значение из i-й строки SBOX по индексу nibble  
 subst: int = SBOX[i][nibble]  
 # Сдвигаем подставленное значение на нужную позицию и добавляем к результату  
 result |= (subst << (4 \* i))  
  
 # Возвращаем преобразованное 32-битовое значение  
 return result  
  
  
def encrypt\_block(block: bytes, round\_keys: List[int]) -> bytes:  
 *"""  
 Шифрует один 64-битовый блок данных с использованием шифра Магма.  
 Аргументы: block (bytes): Входной блок длиной 64 бита (8 байт); round\_keys (List[int]): Список из 32 раундовых ключей.  
 Возвращает: bytes: Зашифрованный блок (8 байт).  
 Обработка ошибок: ValueError: Если длина блока не равна 8 байтам.  
 """* # Проверяем, что длина блока точно 64 бита (8 байт)  
 if len(block) != 8:  
 raise ValueError("Блок должен быть длиной 64 бита (8 байт)")  
  
 # Разделяем блок на две части по 32 бита:  
 # Левую часть (L) — первые 4 байта  
 L: int = int.from\_bytes(block[:4], 'big')  
 # Правую часть (R) — последние 4 байта  
 R: int = int.from\_bytes(block[4:], 'big')  
  
 # Выполняем 32 раунда сети Фейстеля  
 for i in range(32):  
 # Сохраняем текущую правую часть, так как она станет новой левой  
 old\_R: int = R  
 # Новая правая часть: старая левая XOR результат функции G от правой и ключа  
 R = L ^ G(R, round\_keys[i])  
 # Новая левая часть: старая правая  
 L = old\_R  
  
 # Собираем зашифрованный блок: сначала R, затем L (поменяли местами после раундов)  
 encrypted: bytes = R.to\_bytes(4, 'big') + L.to\_bytes(4, 'big')  
  
 # Возвращаем 8 байт зашифрованного блока  
 return encrypted  
  
  
def decrypt\_block(block: bytes, round\_keys: List[int]) -> bytes:  
 *"""  
 Расшифровывает один 64-битовый блок данных с использованием шифра Магма.  
 Аргументы: block (bytes): Входной блок длиной 64 бита (8 байт); round\_keys (List[int]): Список из 32 раундовых ключей.  
 Возвращает: bytes: Расшифрованный блок (8 байт).  
 Обработка ошибок: ValueError: Если длина блока не равна 8 байтам.  
 """* # Проверяем, что длина блока точно 64 бита (8 байт)  
 if len(block) != 8:  
 raise ValueError("Блок должен быть длиной 64 бита (8 байт)")  
  
 # Разделяем блок на две части по 32 бита:  
 # Левую часть (L) — первые 4 байта  
 L: int = int.from\_bytes(block[:4], 'big')  
 # Правую часть (R) — последние 4 байта  
 R: int = int.from\_bytes(block[4:], 'big')  
  
 # Выполняем 32 раунда в обратном порядке ключей  
 for i in range(31, -1, -1):  
 # Сохраняем текущую правую часть  
 old\_R: int = R  
 # Новая правая часть: старая левая XOR результат функции G от правой и ключа  
 R = L ^ G(R, round\_keys[i])  
 # Новая левая часть: старая правая  
 L = old\_R  
  
 # Собираем расшифрованный блок: сначала R, затем L  
 decrypted: bytes = R.to\_bytes(4, 'big') + L.to\_bytes(4, 'big')  
  
 # Возвращаем 8 байт расшифрованного блока  
 return decrypted  
  
  
def process\_file(input\_file: str, output\_file: str, key: bytes, mode: str = "encrypt") -> None:  
 *"""  
 Обрабатывает файл: шифрует или расшифровывает его поблочно.  
 Аргументы:  
 input\_file (str): Путь к входному файлу.  
 output\_file (str): Путь к выходному файлу.  
 key (bytes): Ключ шифрования (256 бит).  
 mode (str): Режим работы ('encrypt' или 'decrypt').  
 Обработка ошибок: FileNotFoundError: Если входной файл не найден.  
 """* # Генерируем раундовые ключи из исходного ключа  
 round\_keys: List[int] = generate\_round\_keys(key)  
  
 # Открываем входной файл в бинарном режиме для чтения  
 with open(input\_file, 'rb') as f\_in:  
 # Открываем выходной файл в бинарном режиме для записи  
 with open(output\_file, 'wb') as f\_out:  
 # Бесконечный цикл для чтения файла поблочно  
 while True:  
 # Читаем очередной блок размером 64 бита (8 байт)  
 block: bytes = f\_in.read(8)  
 # Если блок пустой (конец файла), выходим из цикла  
 if not block:  
 break  
 # Если блок меньше 8 байт (последний блок), дополняем нулями  
 if len(block) < 8:  
 block += b'\x00' \* (8 - len(block))  
 # Выбираем режим обработки  
 if mode == "encrypt":  
 # Шифруем блок  
 result: bytes = encrypt\_block(block, round\_keys)  
 else:  
 # Расшифровываем блок  
 result: bytes = decrypt\_block(block, round\_keys)  
 # Записываем результат в выходной файл  
 f\_out.write(result)  
  
  
def main() -> None:  
 *"""  
 Основная функция программы: запрашивает у пользователя параметры и запускает обработку файла.  
 """* # Выводим приветственное сообщение  
 print("Программа шифрования/расшифрования с использованием шифра Магма")  
 print("-----------------------------------------------------------")  
  
 # Запрашиваем режим работы у пользователя  
 while True:  
 # Получаем ввод и убираем лишние пробелы, приводим к нижнему регистру  
 mode: str = input("Выберите режим (encrypt/decrypt): ").strip().lower()  
 # Проверяем, что введён корректный режим  
 if mode in ["encrypt", "decrypt"]:  
 break  
 # Если режим неверный, выводим сообщение об ошибке  
 print("Ошибка: введите 'encrypt' или 'decrypt'")  
  
 # Запрашиваем путь к входному файлу  
 while True:  
 # Получаем путь к файлу и убираем лишние пробелы  
 input\_file: str = input("Введите путь к входному файлу: ").strip()  
 # Проверяем, существует ли файл  
 if os.path.exists(input\_file):  
 break  
 # Если файла нет, выводим ошибку  
 print("Ошибка: файл не найден")  
  
 # Запрашиваем путь к выходному файлу  
 output\_file: str = input("Введите путь к выходному файлу: ").strip()  
  
 # Запрашиваем ключ шифрования  
 while True:  
 # Получаем ключ в виде hex-строки  
 print("Пример ключа: 000102030405060708090a0b0c0d0e0f101112131415161718191a1b1c1d1e1f")  
 key\_hex: str = input("Введите ключ (256 бит в hex, 64 символа): ").strip()  
 try:  
 # Преобразуем hex-строку в байты  
 key: bytes = bytes.fromhex(key\_hex)  
 # Проверяем, что длина ключа ровно 32 байта (256 бит)  
 if len(key) == 32:  
 break  
 # Если длина неверная, выводим ошибку  
 print("Ошибка: ключ должен быть длиной 256 бит (64 символа в hex)")  
 except ValueError:  
 # Если строка не в формате hex, выводим ошибку  
 print("Ошибка: неверный формат hex-строки")  
  
 # Выполняем обработку файла  
 try:  
 # Запускаем шифрование или расшифрование  
 process\_file(input\_file, output\_file, key, mode)  
 # Сообщаем об успешном завершении  
 print(f"Операция завершена! Результат сохранён в {output\_file}")  
 except Exception as e:  
 # Если произошла ошибка (например, файл не открылся), выводим её  
 print(f"Произошла ошибка: {e}")  
  
  
# запуск  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Вызываем основную функцию при запуске скрипта  
 main()

## Запуск

Сохранить код в файл, например <script>.py.

Запустить в терминале: python <script>.py.

Дополнительные инструкции:

* выбрать encrypt или decrypt;
* указать путь к файлу (например, plaintext.txt);
* указать имя выходного файла (например, encrypted.bin);
* предоставить ключ, например: ffeeddccbbaa99887766554433221100f0f1f2f3f4f5f6f7f8f9fafbfcfdfeff.

## Пример работы

Пример вывода результата работы программы «шифр Магма»:

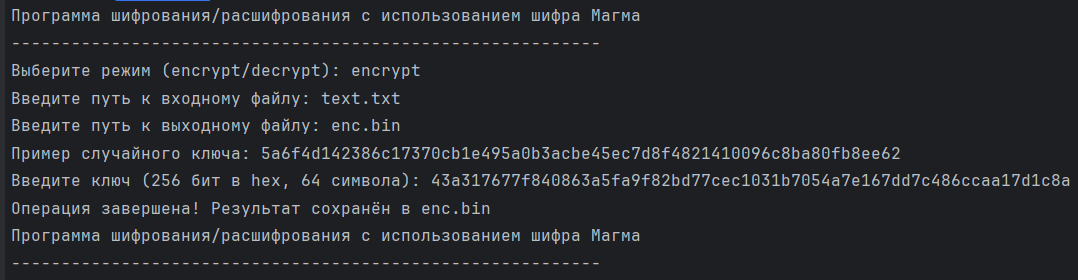


Рисунок 1 – Пример работы программы «шифр Магма»

Пример работы программы «шифр Магма» с параметром помощи <scrypt>.py –h

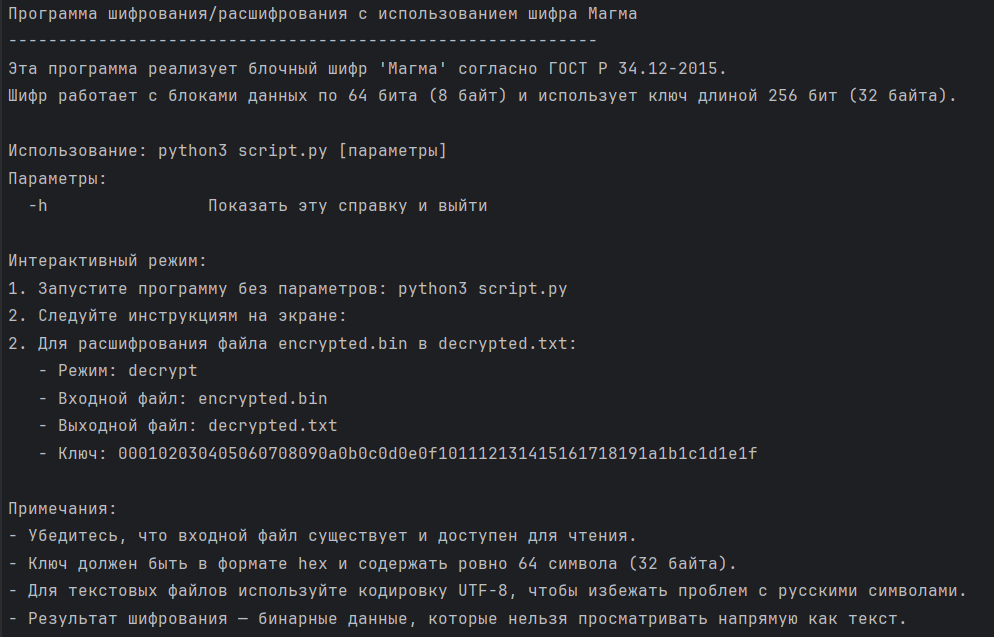


Рисунок 2 – Пример работы программы «шифр Магма» вывод помощи

# ПРИЛОЖЕНИЕ А.

**Пример списка использованных источников**

* 1. ГОСТ Р 34.12-2015. Информационная технология. Криптографическая защита информации. Блочные шифры. – Введ. 2016-01-01. – Москва: Стандартинформ, 2015. – 25 с.
  2. Бабаш А.В., Баранова Е.К. Анализ криптографической стойкости алгоритма Магма // Информационная безопасность. – 2018. – № 2. – С. 15-22.
  3. Молдовян А.А. Современная криптография: теория и практика. – Санкт-Петербург: БХВ-Петербург, 2012. – 512 с.
  4. Смирнов А.А., Петров И.В. Особенности программной реализации блочного шифра Магма // Сборник трудов конференции «RusCrypto’2017». – Москва, 2017. – С. 102-108.
  5. Мартин Р. Чистый код: создание, анализ и рефакторинг // «Библиотека программиста (Питер)», 2024г.
  6. Шоу Зед. Легкий способ выучить Python 3 // «Бомбора», 2021г.