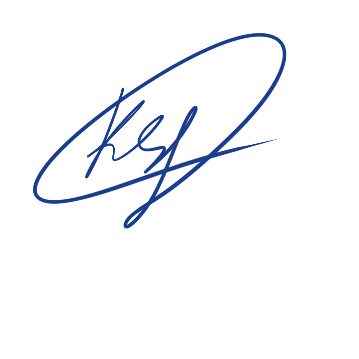
**Федеральное государственное бюджетное образовательное учреждение высшего образования "Белгородский государственный технологический университет им. В.Г. Шухова"**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем.

**Лабораторная работа №3**

Объединение блочных шифров.

Выполнил:

Студент группы КБ-211

Коренев Д.Н.

Принял:

Смакаев А.В.

Оглавление

[Задание 3](#_Toc180808330)

[Реализация программы 4](#_Toc180808331)

[Код программы 5](#_Toc180808332)

[Тестирование производительности 14](#_Toc180808333)

[Вывод 18](#_Toc180808334)

*Цель работы:* ознакомится с принципами объединения блочных шифров и алгоритмом 3DES. Разработать консольное приложение, использующее для шифрования и расшифрования алгоритм 3DES.

Задание

1. Разработать консольное приложение, реализующее шифрование/расшифрование файла по алгоритму 3DES EDE (Encrypt-Decrypt-Encrypt) с использованием следующих режимов сцепления блоков:

* 3DES ECB;
* 3DES Inner CBC;
* 3DES Outer CBC;
* 3DES with pad;
* native 3DES (если есть в вашей криптографической библиотеке -<https://pycryptodome.readthedocs.io/en/latest/src/cipher/des3.html>).

Для шифрования использовать три разных ключа. Ключи записывать в файл в бинарном виде без разделителей.

Инициализирующий вектор размером 8 байт также записывать в бинарном виде в отдельный файл.

**Требования к консольному приложению**

Консольное приложение должно иметь три режима:

* генерация ключа;
* шифрование файла;
* расшифрование файла.

Консольное приложение должно принимать на вход следующие аргументы:

* режим работы;
* режим шифрования;
* путь к файлу для шифрования/расшифрования (если требуется);
* путь к файлу для сохранения зашифрованного/расшифрованного файла (если требуется);
* путь к файлу с инициализирующим вектором (если требуется);
* путь к файлу с ключом.

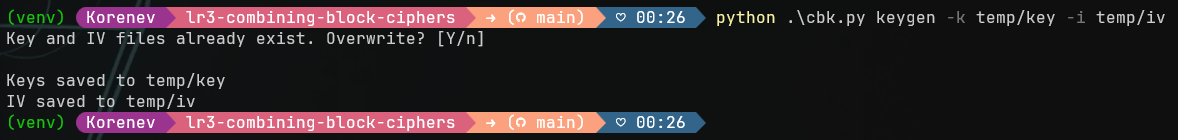
Приветствуется использование аргументов командной строки для передачи вышеописанных параметров. Наличие интерактивного режима не обязательно – при его отсутствии или при запуске с неверными аргументами приложение должно показывать инструкцию по использованию.

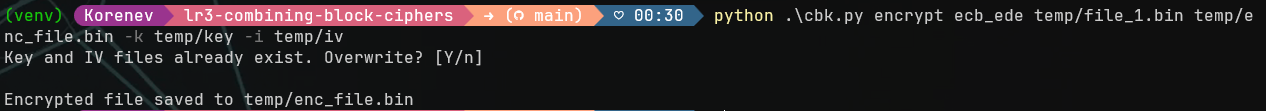
Ключ для 3DES состоит из трех ключей DES. Для хранения ключа использовать бинарный формат, записывать все три ключа подряд без разделителей.

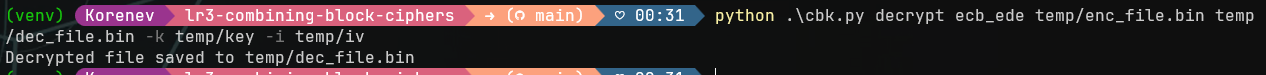
Реализация программы

Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание  
Рисунок 1. Помощь по программе.

  
Рисунок 2. Генерация ключей.

  
Риснуок 3. Шифрование файла.

  
Рисунок 4. Расшифровывание файла.

Код программы

Python 3.12

import os

import argparse

from hashlib import sha256

from Crypto.Cipher import DES3

from Crypto.Cipher import DES

from Crypto.Random import get\_random\_bytes

from Crypto.Util.Padding import pad, unpad

import colorama

# ✨ 3DES ECB EDE ---------------------------------------------------------------

class DES3\_ECB\_EDE:

    def \_\_init\_\_(self, key1, key2, key3):

        self.key1 = key1

        self.key2 = key2

        self.key3 = key3

    def encrypt(self, data):

        '''

        Encrypt the data

        Args:

            data (bytes): The data to encrypt

        Returns:

            bytes: The encrypted data

        '''

        data = pad(data, DES.block\_size)

        cipher = DES.new(self.key1, DES.MODE\_ECB)

        data = cipher.encrypt(data)

        cipher = DES.new(self.key2, DES.MODE\_ECB)

        data = cipher.decrypt(data)

        cipher = DES.new(self.key3, DES.MODE\_ECB)

        data = cipher.encrypt(data)

        return data

    def decrypt(self, data):

        '''

        Decrypt the data

        Args:

            data (bytes): The data to decrypt

        Returns:

            bytes: The decrypted data

        '''

        cipher = DES.new(self.key3, DES.MODE\_ECB)

        data = cipher.decrypt(data)

        cipher = DES.new(self.key2, DES.MODE\_ECB)

        data = cipher.encrypt(data)

        cipher = DES.new(self.key1, DES.MODE\_ECB)

        data = cipher.decrypt(data)

        data = unpad(data, DES.block\_size)

        return data

# ✨ native 3DES ----------------------------------------------------------------

class DES3\_NATIVE\_EDE:

    def \_\_init\_\_(self, key1, key2, key3):

        self.key = key1 + key2 + key3

        assert len(self.key) == 24, f"Key must be 24 bytes long, got {

            len(self.key)}"

    def encrypt(self, data):

        '''

        Encrypt the data

        Args:

            data (bytes): The data to encrypt

        Returns:

            bytes: The encrypted data

        '''

        data = pad(data, DES3.block\_size)

        cipher = DES3.new(self.key, DES3.MODE\_ECB)

        data = cipher.encrypt(data)

        return data

    def decrypt(self, data):

        '''

        Decrypt the data

        Args:

            data (bytes): The data to decrypt

        Returns:

            bytes: The decrypted data

        '''

        cipher = DES3.new(self.key, DES3.MODE\_ECB)

        data = cipher.decrypt(data)

        data = unpad(data, DES3.block\_size)

        return data

# ✨ 3DES Inner CBC -------------------------------------------------------------

# во внутреннем CBC сцепление блоков происходит на каждом из трех этапов

# шифрования

class DES3\_INNER\_CBC\_EDE:

    def \_\_init\_\_(self, key1, key2, key3, iv):

        self.key1 = key1

        self.key2 = key2

        self.key3 = key3

        self.iv = iv

    def encrypt(self, data):

        '''

        Encrypt the data

        Args:

            data (bytes): The data to encrypt

        Returns:

            bytes: The encrypted data

        '''

        data = pad(data, DES.block\_size)

        cipher = DES.new(self.key1, DES.MODE\_CBC, self.iv)

        data = cipher.encrypt(data)

        cipher = DES.new(self.key2, DES.MODE\_CBC, self.iv)

        data = cipher.decrypt(data)

        cipher = DES.new(self.key3, DES.MODE\_CBC, self.iv)

        data = cipher.encrypt(data)

        return data

    def decrypt(self, data):

        '''

        Decrypt the data

        Args:

            data (bytes): The data to decrypt

        Returns:

            bytes: The decrypted data

        '''

        cipher = DES.new(self.key3, DES.MODE\_CBC, self.iv)

        data = cipher.decrypt(data)

        cipher = DES.new(self.key2, DES.MODE\_CBC, self.iv)

        data = cipher.encrypt(data)

        cipher = DES.new(self.key1, DES.MODE\_CBC, self.iv)

        data = cipher.decrypt(data)

        data = unpad(data, DES.block\_size)

        return data

# ✨ 3DES Outer CBC -------------------------------------------------------------

# во внешнем CBC сцепление работает так, как будто все три этапа шифрования

# являются одним

class DES3\_OUTER\_CBC\_EDE:

    def \_\_init\_\_(self, key1, key2, key3, iv):

        self.key = key1 + key2 + key3

        assert len(self.key) == 24, f"Key must be 24 bytes long, got {

            len(self.key)}"

        self.iv = iv

    def encrypt(self, data):

        '''

        Encrypt the data

        Args:

            data (bytes): The data to encrypt

        Returns:

            bytes: The encrypted data

        '''

        data = pad(data, DES.block\_size)

        cipher\_encrypt = DES3.new(self.key, DES3.MODE\_CBC, self.iv)

        data = cipher\_encrypt.encrypt(data)

        return data

    def decrypt(self, data):

        '''

        Decrypt the data

        Args:

            data (bytes): The data to decrypt

        Returns:

            bytes: The decrypted data

        '''

        cipher\_decrypt = DES3.new(self.key, DES3.MODE\_CBC, self.iv)

        data = cipher\_decrypt.decrypt(data)

        data = unpad(data, DES.block\_size)

        return data

# ✨ 3DES with pad --------------------------------------------------------------

# В режиме 3DES with pad между первым и вторым, а также между вторым и третьим

# шифрованиями текст дополняется строкой случайных битов длинной полблока.

# Таким образом обеспечивается перекрытие блоков шифрования

class DES3\_ECB\_PAD\_EDE:

    def \_\_init\_\_(self, key1, key2, key3):

        self.key1 = key1

        self.key2 = key2

        self.key3 = key3

    def \_\_add\_random\_bits(self, data, block\_size):

        random\_bits = get\_random\_bytes(block\_size // 2)

        return random\_bits + data

    def \_\_remove\_random\_bits(self, data, block\_size):

        return data[block\_size // 2:]

    def encrypt(self, data):

        '''

        Encrypt the data

        Args:

            data (bytes): The data to encrypt

        Returns:

            bytes: The encrypted data

        '''

        # Padding the data to be a multiple of the block size

        data = pad(data, DES.block\_size)

        # Encrypt with the first key

        cipher1 = DES.new(self.key1, DES.MODE\_ECB)

        data = cipher1.encrypt(data)

        # Add random bits between first and second encryption

        data = self.\_\_add\_random\_bits(data, DES.block\_size)

        # Padding the data to be a multiple of the block size

        data = pad(data, DES.block\_size)

        # Encrypt with the second key

        cipher2 = DES.new(self.key2, DES.MODE\_ECB)

        data = cipher2.decrypt(data)

        # Add random bits between second and third encryption

        data = self.\_\_add\_random\_bits(data, DES.block\_size)

        # Padd the data to be a multiple of the block size

        data = pad(data, DES.block\_size)

        # Encrypt with the third key

        cipher3 = DES.new(self.key3, DES.MODE\_ECB)

        data = cipher3.encrypt(data)

        return data

    def decrypt(self, data):

        '''

        Decrypt the data

        Args:

            data (bytes): The data to decrypt

        Returns:

            bytes: The decrypted data

        '''

        # Decrypt with the third key

        cipher3 = DES.new(self.key3, DES.MODE\_ECB)

        data = cipher3.decrypt(data)

        # Unpad the decrypted data

        data = unpad(data, DES.block\_size)

        # Remove random bits between second and third encryption from the start

        # of the data

        data = self.\_\_remove\_random\_bits(

            data, DES.block\_size)

        # Decrypt with the second key

        cipher2 = DES.new(self.key2, DES.MODE\_ECB)

        data = cipher2.encrypt(data)

        # Unpad the decrypted data

        data = unpad(data, DES.block\_size)

        # Remove random bits between first and second encryption

        data = self.\_\_remove\_random\_bits(

            data, DES.block\_size)

        # Decrypt with the first key

        cipher1 = DES.new(self.key1, DES.MODE\_ECB)

        data = cipher1.decrypt(data)

        # Unpad the decrypted data

        data = unpad(data, DES.block\_size)

        return data

def test():

    '''

    Test all the implemented 3DES modes

    Raises:

        Exception: If any of the tests fail

    '''

    data = get\_random\_bytes(1533)

    key = get\_random\_bytes(24)

    d3\_ecb\_ede = DES3\_ECB\_EDE(key[0:8], key[8:16], key[16:24])

    ciphertext = d3\_ecb\_ede.encrypt(data)

    plaintext = d3\_ecb\_ede.decrypt(ciphertext)

    if sha256(data).digest() != sha256(plaintext).digest():

        raise Exception("3DES ECB EDE failed")

    else:

        print(f"{colorama.Fore.GREEN}3DES ECB EDE passed{

            colorama.Style.RESET\_ALL}")

    data = get\_random\_bytes(1533)

    key = get\_random\_bytes(24)

    d3\_ecb\_ede = DES3\_NATIVE\_EDE(key[0:8], key[8:16], key[16:24])

    ciphertext = d3\_ecb\_ede.encrypt(data)

    plaintext = d3\_ecb\_ede.decrypt(ciphertext)

    if sha256(data).digest() != sha256(plaintext).digest():

        raise Exception("3DES Native EDE failed")

    else:

        print(f"{colorama.Fore.GREEN}3DES Native EDE passed{

            colorama.Style.RESET\_ALL}")

    # Data to encrypt

    data = get\_random\_bytes(1533)

    # Keys and IVs for each DES operation

    key = get\_random\_bytes(24)

    iv = get\_random\_bytes(8)  # Single IV for inner CBC

    d3\_inner\_cbc = DES3\_INNER\_CBC\_EDE(key[0:8], key[8:16], key[16:24], iv)

    ciphertext = d3\_inner\_cbc.encrypt(data)

    plaintext = d3\_inner\_cbc.decrypt(ciphertext)

    if sha256(data).digest() != sha256(plaintext).digest():

        raise Exception("3DES Inner CBC EDE failed")

    else:

        print(f"{colorama.Fore.GREEN}3DES Inner CBC EDE passed{

            colorama.Style.RESET\_ALL}")

    # Data to be encrypted

    data = get\_random\_bytes(1533)

    # Key must be either 24 bytes long

    key = get\_random\_bytes(24)

    # Generate a random IV

    iv = get\_random\_bytes(DES3.block\_size)

    d3\_outer\_cbc = DES3\_OUTER\_CBC\_EDE(key[0:8], key[8:16], key[16:24], iv)

    ciphertext = d3\_outer\_cbc.encrypt(data)

    plaintext = d3\_outer\_cbc.decrypt(ciphertext)

    if sha256(data).digest() != sha256(plaintext).digest():

        raise Exception("3DES Outer CBC EDE failed")

    else:

        print(f"{colorama.Fore.GREEN}3DES Outer CBC EDE passed{

            colorama.Style.RESET\_ALL}")

    # Data to be encrypted

    data = get\_random\_bytes(1533)

    # Keys must be 8 bytes long each

    key = get\_random\_bytes(24)

    d3\_ecb\_pad\_ede = DES3\_ECB\_PAD\_EDE(key[0:8], key[8:16], key[16:24])

    ciphertext = d3\_ecb\_pad\_ede.encrypt(data)

    plaintext = d3\_ecb\_pad\_ede.decrypt(ciphertext)

    if sha256(data).digest() != sha256(plaintext).digest():

        raise Exception("3DES with pad failed")

    else:

        print(f"{colorama.Fore.GREEN}3DES with pad passed{

            colorama.Style.RESET\_ALL}")

def save\_keys(key1, key2, key3, filename):

    '''

    Save the 3 keys to a file

    Args:

        key1 (bytes): The first key

        key2 (bytes): The second key

        key3 (bytes): The third key

        filename (str): The name of the file to save the keys to

    '''

    with open(filename, "wb") as f:

        f.write(key1)

        f.write(key2)

        f.write(key3)

def read\_keys(filename):

    '''

    Read the 3 keys from a file

    Args:

        filename (str): The name of the file to read the keys from

    Returns:

        tuple: The 3 keys

    '''

    with open(filename, "rb") as f:

        key1 = f.read(8)

        key2 = f.read(8)

        key3 = f.read(8)

    return key1, key2, key3

def generate\_keys():

    '''

    Generate 3 random keys

    Returns:

        tuple: The 3 keys

    '''

    key1 = get\_random\_bytes(8)

    key2 = get\_random\_bytes(8)

    key3 = get\_random\_bytes(8)

    return key1, key2, key3

def save\_iv(iv, filename):

    '''

    Save the IV to a file

    Args:

        iv (bytes): The IV

        filename (str): The name of the file to save the IV to

    '''

    with open(filename, "wb") as f:

        f.write(iv)

def read\_iv(filename):

    '''

    Read the IV from a file

    Args:

        filename (str): The name of the file to read the IV from

    Returns:

        bytes: The IV

    '''

    with open(filename, "rb") as f:

        iv = f.read()

    return iv

def generate\_iv():

    '''

    Generate a random IV

    Returns:

        bytes: The IV

    '''

    return get\_random\_bytes(DES3.block\_size)

def encrypt\_file(input\_file, output\_file, mode, key1, key2, key3, iv=None):

    '''

    Encrypt a file using 3DES

    Args:

        input\_file (str): The name of the file to encrypt

        output\_file (str): The name of the file to save the encrypted data to

        mode (str): The mode of operation to use

        key1 (bytes): The first key

        key2 (bytes): The second key

        key3 (bytes): The third key

        iv (bytes): The IV

    '''

    with open(input\_file, "rb") as f:

        data = f.read()

    if mode == "ecb\_ede":

        cipher = DES3\_ECB\_EDE(key1, key2, key3)

    if mode == "native\_ede":

        cipher = DES3\_NATIVE\_EDE(key1, key2, key3)

    elif mode == "inner\_cbc\_ede":

        cipher = DES3\_INNER\_CBC\_EDE(key1, key2, key3, iv)

    elif mode == "outer\_cbc\_ede":

        cipher = DES3\_OUTER\_CBC\_EDE(key1, key2, key3, iv)

    elif mode == "ecb\_pad\_ede":

        cipher = DES3\_ECB\_PAD\_EDE(key1, key2, key3)

    else:

        raise ValueError(f"Invalid mode: {mode}")

    ciphertext = cipher.encrypt(data)

    with open(output\_file, "wb") as f:

        f.write(ciphertext)

def decrypt\_file(input\_file, output\_file, mode, key1, key2, key3, iv=None):

    '''

    Decrypt a file using 3DES

    Args:

        input\_file (str): The name of the file to decrypt

        output\_file (str): The name of the file to save the decrypted data to

        mode (str): The mode of operation to use

        key1 (bytes): The first key

        key2 (bytes): The second key

        key3 (bytes): The third key

        iv (bytes): The IV

    '''

    with open(input\_file, "rb") as f:

        data = f.read()

    if mode == "ecb\_ede":

        cipher = DES3\_ECB\_EDE(key1, key2, key3)

    if mode == "native\_ede":

        cipher = DES3\_NATIVE\_EDE(key1, key2, key3)

    elif mode == "inner\_cbc\_ede":

        cipher = DES3\_INNER\_CBC\_EDE(key1, key2, key3, iv)

    elif mode == "outer\_cbc\_ede":

        cipher = DES3\_OUTER\_CBC\_EDE(key1, key2, key3, iv)

    elif mode == "ecb\_pad\_ede":

        cipher = DES3\_ECB\_PAD\_EDE(key1, key2, key3)

    else:

        raise ValueError(f"Invalid mode: {mode}")

    plaintext = cipher.decrypt(data)

    with open(output\_file, "wb") as f:

        f.write(plaintext)

def main():

    parser = argparse.ArgumentParser(

        description=f"{colorama.Fore.CYAN}3DES encryption with defferent "

        + f"modes of operation{colorama.Style.RESET\_ALL}")

    parser.add\_argument(

        "-t", "--test", action="store\_true", help="Run the tests")

    parser.add\_argument(

        "-y", "--yes", action="store\_true", help="Skip the confirmation")

    # mode [key generation / encrypting / decrypting]

    parser.add\_argument(

        "mode", nargs='?',

        help="The mode of operation [keygen / encrypt / decrypt]")

    # encryption method [ecb\_ede / inner\_cbc\_ede / outer\_cbc\_ede / ecb\_pad\_ede]

    parser.add\_argument(

        "method", nargs='?',

        help=f"The encryption method [ecb\_ede / native\_ede / inner\_cbc\_ede /"

        + " outer\_cbc\_ede / ecb\_pad\_ede]")

    # input file

    parser.add\_argument("input", nargs='?', help="The input file")

    # output file

    parser.add\_argument("output", nargs='?', help="The output file")

    # key file

    parser.add\_argument("-k", "--key", help="The key file")

    # iv file

    parser.add\_argument("-i", "--iv", help="The IV file")

    args = parser.parse\_args()

    if args.test:

        test()

        exit(0)

    if args.mode and args.mode == "keygen":

        if not args.key or not args.iv:

            print("Key and IV files are required for key generation")

            exit(1)

        if os.path.exists(args.key) or os.path.exists(args.iv):

            if not args.yes:

                print("Key and IV files already exist. Overwrite? [Y/n]")

                \_input = input().lower()

                if \_input != "y" and \_input != "":

                    exit(0)

        key1, key2, key3 = generate\_keys()

        save\_keys(key1, key2, key3, args.key)

        iv = generate\_iv()

        save\_iv(iv, args.iv)

        if not args.yes:

            print(f"Keys saved to {args.key}")

            print(f"IV saved to {args.iv}")

        exit(0)

    if args.mode \

        and (args.mode.lower() == "encrypt"

             or args.mode.lower() == "e"):

        if not (args.key and args.iv):

            print("Key and IV files are required for encryption")

            exit(1)

        if not (os.path.exists(args.key) and os.path.exists(args.iv)):

            print("Key and IV files do not exist")

            exit(1)

        if not (args.input and args.output):

            print("Input and output files are required for encryption")

            exit(1)

        if not (args.method == "ecb\_ede"

                or args.method == "inner\_cbc\_ede"

                or args.method == "outer\_cbc\_ede"

                or args.method == "ecb\_pad\_ede"

                or args.method == "native\_ede"):

            print("Invalid encryption method")

            exit(1)

        if os.path.exists(args.output):

            if not args.yes:

                print("Key and IV files already exist. Overwrite? [Y/n]")

                \_input = input().lower()

                if \_input != "y" and \_input != "":

                    exit(0)

        key1, key2, key3 = read\_keys(args.key)

        iv = read\_iv(args.iv)

        encrypt\_file(args.input, args.output,

                     args.method, key1, key2, key3, iv)

        if not args.yes:

            print(f"Encrypted file saved to {args.output}")

        exit(0)

    if args.mode \

            and (args.mode.lower() == "decrypt"

                 or args.mode.lower() == "d"):

        if not (args.key and args.iv):

            print("Key and IV files are required for encryption")

            exit(1)

        if not (os.path.exists(args.key) and os.path.exists(args.iv)):

            print("Key and IV files do not exist")

            exit(1)

        if not (args.input and args.output):

            print("Input and output files are required for encryption")

            exit(1)

        if not (args.method == "ecb\_ede"

                or args.method == "inner\_cbc\_ede"

                or args.method == "outer\_cbc\_ede"

                or args.method == "ecb\_pad\_ede"

                or args.method == "native\_ede"):

            print("Invalid encryption method")

            exit(1)

        if os.path.exists(args.output):

            if not args.yes:

                print("Key and IV files already exist. Overwrite? [Y/n]")

                \_input = input().lower()

                if \_input != "y" and \_input != "":

                    exit(0)

        key1, key2, key3 = read\_keys(args.key)

        iv = read\_iv(args.iv)

        decrypt\_file(args.input, args.output,

                     args.method, key1, key2, key3, iv)

        if not args.yes:

            print(f"Decrypted file saved to {args.output}")

        exit(0)

    parser.print\_help()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

2. Сравнить производительность алгоритма с каждым режимом сцепления блоков, произведя замеры времени шифрования и расшифрования файлов размером 1МБ, 5МБ, 10МБ, 50МБ, 100МБ. Визуализировать результаты в виде графиков или столбчатых диаграмм.

Тестирование производительности

Результаты тестирования:

Изображение выглядит как текст, снимок экрана, линия, диаграмма

Автоматически созданное описание  
Рисунок 5. Результаты исследования.

Для тестирования производетельности был написан скрипт:

Python 3.12

from hashlib import sha256

import time

import os

from Crypto.Random import get\_random\_bytes

import colorama

import cbk

def generate\_fake\_file(filename, size):

    with open(filename, 'wb') as f:

        f.write(get\_random\_bytes(size))

if \_\_name\_\_ == '\_\_main\_\_':

    if not os.path.exists('temp'):

        os.makedirs('temp')

        print('Directory temp created')

    size\_range = range(1, 102, 10)

    test\_iterations = 3

    step = (max(size\_range) - min(size\_range))/(len(size\_range)-1)

    print(f'Size range: {list(size\_range)}')

    print(f'Step: {step}')

    # generate files from 1 MB to 100 MB with 1 MB step

    for i in size\_range:

        if not os.path.exists(f'temp/file\_{i}.bin'):

            generate\_fake\_file(f'temp/file\_{i}.bin', i \* 1024 \* 1024)

            print(

                f'{colorama.Fore.GREEN}File temp/file\_{i}.bin '

                + f'created{colorama.Style.RESET\_ALL}')

    # with every method from cbk.py, encrypt and decrypt every file, measure

    # time and write it to a csv file. Compate file's hash.

    # Run all test 3 times.

    # generate keys

    key1, key2, key3 = cbk.generate\_keys()

    # generate iv

    iv = cbk.generate\_iv()

    modes = ["ecb\_ede",

             "native\_ede",

             "inner\_cbc\_ede",

             "outer\_cbc\_ede",

             "ecb\_pad\_ede"]

    mod\_count = len(modes)

    total\_iterations = mod\_count \* \

        (len(size\_range)) \* test\_iterations

    total\_iterations = int(total\_iterations)

    total\_digits = len(str(int(total\_iterations)))

    with open(f'temp/test.csv', 'w') as f:

        f.write('method,iteration,file\_size,time\n')

        for mode in modes:

            for j in size\_range:

                input\_file = f'temp/file\_{j}.bin'

                with open(input\_file, "rb") as fd:

                    data = fd.read()

                for i in range(0, test\_iterations):

                    current\_iteration = modes.index(mode) \

                        \* (len(size\_range)) \

                        \* test\_iterations + (j/step) \

                        \* test\_iterations + i + 1

                    current\_iteration = int(current\_iteration)

                    print(

                        f'{colorama.Fore.YELLOW}'

                        + f'{current\_iteration:{total\_digits}d}/'

                        + f'{total\_iterations:{total\_digits}d} '

                        + f'Method: {mode}, '

                        + f'File size: {j} MB, '

                        + f'Test {i+1}{colorama.Style.RESET\_ALL}'

                    )

                    start = time.time()

                    if mode == "ecb\_ede":

                        cipher = cbk.DES3\_ECB\_EDE(key1, key2, key3)

                    elif mode == "native\_ede":

                        cipher = cbk.DES3\_NATIVE\_EDE(key1, key2, key3)

                    elif mode == "inner\_cbc\_ede":

                        cipher = cbk.DES3\_INNER\_CBC\_EDE(key1, key2, key3, iv)

                    elif mode == "outer\_cbc\_ede":

                        cipher = cbk.DES3\_OUTER\_CBC\_EDE(key1, key2, key3, iv)

                    elif mode == "ecb\_pad\_ede":

                        cipher = cbk.DES3\_ECB\_PAD\_EDE(key1, key2, key3)

                    else:

                        raise ValueError(f"Invalid mode: {mode}")

                    ciphertext = cipher.encrypt(data)

                    end = time.time()

                    start = time.time()

                    if mode == "ecb\_ede":

                        cipher = cbk.DES3\_ECB\_EDE(key1, key2, key3)

                    elif mode == "native\_ede":

                        cipher = cbk.DES3\_NATIVE\_EDE(key1, key2, key3)

                    elif mode == "inner\_cbc\_ede":

                        cipher = cbk.DES3\_INNER\_CBC\_EDE(key1, key2, key3, iv)

                    elif mode == "outer\_cbc\_ede":

                        cipher = cbk.DES3\_OUTER\_CBC\_EDE(key1, key2, key3, iv)

                    elif mode == "ecb\_pad\_ede":

                        cipher = cbk.DES3\_ECB\_PAD\_EDE(key1, key2, key3)

                    else:

                        raise ValueError(f"Invalid mode: {mode}")

                    plaintext = cipher.decrypt(ciphertext)

                    end = time.time()

                    f.write(f'{mode},{i+1},{j},{end-start}\n')

                    print(

                        f'{colorama.Fore.GREEN}File encrypted and decrypted'

                        + f'{colorama.Style.RESET\_ALL}. '

                        + f'{colorama.Fore.GREEN}Time: '

                        + f'{end-start}{colorama.Style.RESET\_ALL}')

Для построения графика использщовался отдельный скрипт:

Python 3.12

import matplotlib.pyplot as plt

import pandas as pd

import dsmltf

if \_\_name\_\_ == '\_\_main\_\_':

    # read data from csv

    # method,iteration,file\_size,time

    df = pd.read\_csv('temp/test.csv')

    # convert time from s to ms

    df['time'] = df['time'] \* 1000

    # calculate average time for every method for every size by iteration

    means = df.groupby(['method', 'file\_size']).mean()

    means = means.drop(columns='iteration')

    # calculate yerr for every method

    mins = df.groupby(['method', 'file\_size']).min()

    mins = mins.drop(columns='iteration')

    maxs = df.groupby(['method', 'file\_size']).max()

    maxs = maxs.drop(columns='iteration')

    # combine data

    means = means.reset\_index()

    mins = mins.reset\_index()

    maxs = maxs.reset\_index()

    # print(means)

    # print(mins)

    # print(maxs)

    data = pd.merge(means, mins, on=[

                    'method', 'file\_size'], suffixes=('\_mean', '\_min'))

    data = pd.merge(data, maxs, on=['method', 'file\_size'])

    data = data.rename(columns={'time': 'time\_max'})

    # calculate linear regression

    # for every method

    def gen\_poly\_data(x, P):

        return [sum([P[i] \* x \*\* i for i in range(len(P))]) for x in x]

    def gen\_poly\_str(P):

        terms = []

        for i in range(len(P)-1, 0, -1):

            if i == len(P)-1:

                if i == 1:

                    terms.append(f'{P[i]:.1f}x' if P[i] != 0 else '')

                else:

                    terms.append(f'{P[i]:.1f}x^{i}' if P[i] != 0 else '')

            else:

                if i == 1:

                    terms.append(f'{P[i]:+.1f}x' if P[i] != 0 else '')

                else:

                    terms.append(f'{P[i]:+.1f}x^{i}' if P[i] != 0 else '')

        return ''.join(filter(None, terms)) \

            + (f'{P[0]:+.1f}' if P[0] != 0 else '')

    for method in data['method'].unique():

        method\_data = data[data['method'] == method]

        x = method\_data['file\_size']

        y = method\_data['time\_mean']

        m = dsmltf.approx\_poly(y.tolist(), x.tolist(), 1)

        data.loc[data['method'] == method, 'poly'] \

            = gen\_poly\_str(m)

        data.loc[data['method'] == method, 'time\_poly'] \

            = gen\_poly\_data(x, m)

    color = ['#e41a1c', '#377eb8', '#f781bf', '#dede00', '#4daf4a']

    CB\_color\_cycle = ['#377eb8', '#ff7f00', '#4daf4a',

                      '#f781bf', '#a65628', '#984ea3',

                      '#999999', '#e41a1c', '#dede00']

    print(data)

    percent\_to\_plot = 100

    plot\_lim = int(100 / percent\_to\_plot)

    fig, ax = plt.subplots(figsize=(10, 6))

    for method in data['method'].unique():

        method\_data = data[data['method'] == method]

        ax.errorbar(

            method\_data['file\_size'][::plot\_lim],

            method\_data['time\_mean'][::plot\_lim],

            yerr=[(method\_data['time\_mean']

                   - method\_data['time\_min'])[::plot\_lim],

                  (method\_data['time\_max']

                   - method\_data['time\_mean'])[::plot\_lim]],

            label=method,

            fmt='-o',

            color=color[data['method'].unique().tolist().index(method)]

        )

        method\_data = data[data['method'] == method]

        ax.plot(

            method\_data['file\_size'][::plot\_lim],

            method\_data['time\_poly'][::plot\_lim],

            label=f'{method} poly {method\_data["poly"].iloc[0]}',

            linestyle='--',

            color=color[data['method'].unique().tolist().index(method)]

        )

    ax.set\_xlabel('File size (MB)')

    ax.set\_ylabel('Time (ms)')

    ax.legend()

    # add title

    plt.title('Time of processing for different 3DES methods')

    plt.tight\_layout()

    # plt.show()

    plt.savefig('temp/plot.png')

3. Сделать вывод по полученным данным, соотнести его с информацией о криптостойкости каждого из решений, выбрать оптимальный метод сцепления блоков.

Вывод

В ходе лабораторной работы мы ознакомились с принципами объединения блочных шифров и изучили алгоритм 3DES. Мы разработали консольное приложение, реализующее шифрование и расшифрование файлов по алгоритму 3DES EDE с использованием различных режимов сцепления блоков, таких как ECB, Inner и Outer CBC, а также режим 3DES with pad. В процессе работы были разработаны три режима работы приложения: генерация ключа, шифрование файла и расшифрование файла, при этом программа корректно обрабатывала входные параметры через командную строку.

Произведенное тестирование производительности на различных объемах данных (1 МБ, 5 МБ, 10 МБ, 50 МБ и 100 МБ) позволило нам оценить эффективность каждого из режимов сцепления блоков. Результаты показали, что режимы ECB и Native 3DES продемонстрировали наименьшее время выполнения операций, однако режимы с CBC (как Inner, так и Outer) обеспечивали более высокую криптостойкость за счет дополнительной защиты от анализа данных. Визуализация результатов на графиках позволила четко проследить зависимость времени выполнения от объема файла, а также увидеть различия между режимами.

На основании анализа криптостойкости и времени выполнения операций можно сделать вывод, что наибольший баланс между производительностью и защитой данных демонстрируют режимы Inner и Outer CBC. Несмотря на более высокое время шифрования по сравнению с ECB, данные режимы обеспечивают лучшую защиту от атак, таких как анализ шаблонов в зашифрованных данных. Таким образом, оптимальным выбором для использования в системе, где важен баланс между скоростью и криптостойкостью, является использование 3DES в режиме Inner или Outer CBC.