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. \_\_\_\_  Е.В. Шараев  «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 3](#_Toc155608317)

[2 Краткая теоретическая часть 4](#_Toc155608318)

3[Описание программной реализации 4](#_Toc155608319)

3.1 Описание блочного шифра Кузнечик………………………………………….………4

3.2 Описание режимов работы…………………………………………………….……….6

3.2.1 Режим простой замены ……………………………………………………………... 6

3.2.2 Режим простой замены с зацеплением ……………………………………….…..... 6

3.2.3 Режим гаммирования ………………………………………………………….…..... 8

3.2.4 Режим гаммирования с обратной связью по выходу ………………….…….…..... 8

3.2.5 Режим гаммирования с обратной связью по шифртексту ……………….……….. 9

4 Демонстрация работы программы……………………………………………..…………...10

5[Выводы о проделанной работе](#_Toc155608320) 10

6[Список использованных источников](#_Toc155608321) 21

# 1 Задание на практическую работу

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

В рамках практической работы необходимо выполнить следующее:

1. написать программную реализацию одного из следующих симметричных шифров (по выбору студента):

- Магма;

**- Кузнечик; (выбранный вариант)**

- AES;

1. подготовить отчет о выполнении работы.

Программа должна обладать следующей функциональностью:

* принимать на вход файл, содержащий открытый текст, подлежащий зашифрованию, или шифртекст, подлежащий расшифрованию;
* принимать на вход секретный ключ;
* [***дополнительная опция, не являющаяся обязательной***] давать пользователю возможность выбирать режим работы блочного шифра;
* осуществлять зашифрование или расшифрование выбранного файла по выбору пользователя и сохранять результат в новом файле.

Отчет должен содержать следующие составные части:

* раздел с заданием;
* раздел с краткой теоретической частью;
* раздел с описанием программной реализации с учетом особенностей выбранной среды разработки и языка программирования;
* раздел с результатами работы программы;
* раздел с выводами о проделанной работе.

# **2 Краткая теоретическая часть**

«Кузнечик» (англ. Kuznyechik[1] или англ. Kuznechik[2][3]) — симметричный алгоритм блочного шифрования с размером блока 128 бит и длиной ключа 256 бит, использующий для генерации раундовых ключей SP-сеть.

**Описание алгоритма**. Для шифрования, расшифрования и генерации ключа используются следующие функции:

1. XOR с раундовым ключом
2. Нелинейное биективное преобразование **S** (подстановка по таблице соотвестствия)
3. Линейное преобразование **L**, где происходит сдвиг элементов блока на 1 блок, а утраченный после сдвига блок восполняется путем свертки всех блоков в один в результате линейного преобразования

При зашифровке операции XSL производятся 9 раз (раундов), а 10-й раунд включает только операцию наложения раундового ключа. Расшифрование представляет собой последовательное применение обратных процедур. Подборнее алогитм описывается в Части 3

**Режим работы.**

Режимы работы гаммирования (реализованный мною помимо простой подстановки блоков) подразумевает не прямое шифрование блоков открытого текста , а сгенерированного из синхропосылки счетчика (получение *гаммы*), затем на каждом блоке открытого текста применяется операция XOR с поученной *гаммой,* результат чего и попадает в ширф-текст.

Режим

#### **3. Описание программной реализации**

#### **3.1 Описание блочного шифра Кузнечик**

Программа реализована в классе *Cricket* в файле *cricket.py* в данном репозитории.

\* Статические параметры *pi* и pi\_*inv*: Целочисленные массивы, где индекс каждого элемента соответствует значению исходного при биективном нелинейном отображении (X) и обратной опреации (X-1).

\* Метод *\_\_init\_\_(self)* Запускается при инициализации класса. Принимает на вход главный ключ (256 bit), запускает генерацию раундовых ключей и сохраняет их в виде упорядоченного массива в параметре объекта *self.round\_keys*

\* Статический метод *\_\_generate\_round\_keys(key)* Принимает на вход главный 256 битный ключ и генерирует из него 10 раундовых 128 битных ключей.

а) На первом метод разбивает глвный ключ на две равные части, и сохраняет их в массив *self.round\_keys* в качестве первых двух ключей.

б) Остальные 8 (4 пары) ключей вырабатываются в цикле for i in range(4):, где на каждом шаге цикла к предыдущей паре ключей 8-кратно применяются преобразования сети Фейстеля.

\* Метод *encrypt(self, x)*: Принимая на вход блок длинной 128 бит, затем в цикле for rnd in range(9): выполняет 9 раундов зашифрования.

а) Запускает сначала операцию **X** (XOR) с райндовым ключом

*x ^ self.round\_keys[rnd]*

б) Затем нелинейное биективное преобразование **S**

*Cricket.\_\_s\_transformation(x ^ self.round\_keys[rnd])*

в) И наконец линейное преобразование **L**

x = Cricket.\_\_l\_transformation(Cricket.\_\_s\_transformation(x ^ self.round\_keys[rnd]))

где каждый байт 16 раз помножается на соответствующий ему элемент поля Галуа. В программе этот шаг реализован в методе *\_\_linear\_function(x)*

г) Последний, 10 раунд зишифрования является не полным и состоит только из наложения последнего раундового ключа

return x ^ self.round\_keys[-1]

\* Расшифрование *def decrypt(self, x)* устроено противоположным образом. Метод принимает на вход зашифрованный блок длинной 128 бит.

а) Сначала строится развернутый массив ключей

keys = self.round\_keys[::-1]

б) Затем 9 раз применяются обратные функции X-1 S-1L-1

x = Cricket.\_\_s\_inv\_transformation(Cricket.\_\_l\_inv\_transformation(x ^ keys[rnd]))

в) Последний раунд также является не полным и заключается лишь в применении последнего раундового ключа

return x ^ self.round\_keys[-1]

**3.2 Описание режимов работы**

Режимы реализованы в классе *EncryptionMode*.

**3.2.1 Режим простой замены**

В режиме *ecb\_mode* блоки открытого текста напрямую шифруются шифром Кузнечик.

Размер блока всегда 16 байт (128 бит) *block\_size = 16*

Статический метод *\_\_padding\_bytes* сначал добавляет 1 (b'\x01'), а затем нулями добивает количество байт до кратного 16-ти

Шифрования блоков происходит последовательно и независимо друг от друга, поэтому в шифртекст переносятся статистические характеристики исходного текста, а значит этот шифр является не надежным

**3.2.2 Режим простой замены с зацеплением**

Реализован в методе *cbc\_mode*

Сначала генерируется синхропосылка

# Генерируем синхропосылку  
 init\_val = EncryptionMode.\_\_get\_initializing\_value(init\_val\_size)

Размер синхропосылки в данном режиме всегда равен 16 байтам.

Далее при зашифровании синхропосылка добавляется в массив зашифрованных данных

# Добавляем синхропосылку в результат работ  
 result\_bytes.extend(init\_val)

Добавляем паддинги методом добавления 1-цы, а затем заполнения 0-ми до кратной длины блока

При расшифровании определяем массив данных и синхропосылку просто отделяя их из массива

if operator == "decrypt":  
 # Или получаем синхропосылку и зашифрованный текст из зашифрованных данных  
 init\_val, byte\_text = byte\_text[:init\_val\_size], byte\_text[init\_val\_size:]

Затем объявляем объект шифровального класса

# Инициализируем объект класса Cricket  
 cricket = Cricket(key)

И наконец в цикле зашифровываем или расшифровываем каждый блок после чего переопределяем сдвиговый регистр как конкатенацию предыдущего массива за вычетом использованных первых n байтов и шифр-текста

# Переопределяем сдвиговый регистр  
 init\_val = init\_val[block\_size:] + encrypted\_block

На каждом шаге полученные зашифрованные / расшифрованные данные я записываю в заранее объявленный массив

result\_bytes.extend(encrypted\_block)

**3.2.3 Режим гаммирования**

В режиме Гаммирования шифрование происходит не напрямую. Вместо открытого текста алгоритм блочного шифра шифрует счетчик, состоящий из синхропосылки и нулей (на первом этапе), а затем усекается на заданное количество байт. В моей реализации я по умолчанию задаю размер блока 13 байт *block\_size: int = 13*, но оно может быть изменено при вызове функции.

Гамма считается как результат шифрования счетчика, а затьем усечается на заданное количество байт

gamma = cricket.encrypt(counter) >> right\_shift

Затем я увеличиваю счетчик. Инкремент счетчика был вынесен мною в отдельный метод.

counter = EncryptionMode.\_\_increment\_counter(counter)

И накладываю полученную гамму на блоки открытого текста

encrypted\_block = gamma ^ block\_int

В конце конкатенирую полученные данные к коллекции *result\_bytes.extend(encrypted\_block)*

Метод *\_\_get\_counter* генерирует счетчик (размером 128 бит) из синхнопосылки и нулей

Метод *\_\_increment\_counter* очевидно служит для увеличения значения счетчика по мере шифрования.

Расшифрование происходит точно также, за исключением: того что на первом шаге нужно отделить синхропосылку от массива зашифрованных данных, а также удалить нулевые байты в конце до единичного байта (включительно)

**3.2.4 Режим гаммирования с обратной связью по выходу**

Отличие данного режима от обычного гаммирования что каждый последующий блок XOR-ится не со счетчиком а с сдвиговым регистром, который на каждом шаге избавляется от первых n-байт и конкатенируется с gamma.

Я оставил название переменной `counter` как из предыдущего режима, хотя на самом дела счетчиком эта переменная не является. Отделяем первые n байт для дальнейшего зашифрования

counter = init\_val >> (m\_value - 16) \* 8

Вторую часть в байтовом виде сохранит отдельно для составления сдвигового регистра

counter\_rest = int.to\_bytes(init\_val, m\_value, byteorder="big", signed=False)[16:]

Затем переопределяем переменную `init\_val` как сумму конкатенирую массивы байтов `counter\_rest` и `gamma`. Прочие шаги — аналогичны методу гаммирования.

**3.2.5 Режим гаммирования с обратной связью по шифртексту**

Реализация аналогична предыдущему методу, но в конце каждого блока сдивговый регистр переопределяется как конкатенация байт из остатка предыдущего регистра + полученного блока шифртекста

init\_val = int.from\_bytes(  
 counter\_rest +  
 encrypted\_block,  
 byteorder='big',  
 signed=False  
 )

Также при расшифровании в в сдвиговый регистр добавляется не расшифрованный блок гаммы а блок полученный на вход

if operator == "decrypt":  
 # Если расшифровываем, то добавлять в регистр блок шифртекста  
 encrypted\_block = block

**4. Демонстрация работы программы**

Подробная демонстрация работы программы приведена в Приложении А.

**5** [Error: Reference source not found](#_Toc155608320)

Получил навыки реализации алгоритмов шифрования и их режимов работы

# ПРИЛОЖЕНИЕ Б. Список использованных источников

1. ГОСТ 34-12-2015 Информационная технология. Криптографическая защита информации. Блочные шифры. – М: Стандартинформ, 2015 . – 25 с.

2. ГОСТ 34-13-2015 Информационная технология. Криптографическая защита информации. Режимы работы блочных шифров – М: Стандартинформ, 2015 . – 42 с.