Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет   
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы Защиты Информации

ОТЧЁТ

к лабораторной работе №4

на тему

**Асимметричная криптография.** **Алгоритм Мак-Элиса**

Выполнил: студент гр. 253505

Сенько Н. С.

Проверил: ассистент кафедры

информатики

Герчик А. В.

Минск 2025

**СОДЕРЖАНИЕ**

[1 Цель работы 3](#_Toc207722280)

[2 Ход работы 4](#_Toc207722281)

[Заключение 5](#_Toc207722282)

[Приложение А (обязательное) исходный код программного продукта 6](#_Toc207722283)

# 1 ЦЕЛЬ РАБОТЫ

Целью данной лабораторной работы является изучение принципов асимметричная шифрования на примере алгоритма Мак-Элиса.

В рамках работы требуется разработать программные средство для шифрования и дешифрования данных, закрепить навыки работы с асимметричными алгоритмами шифрования, освоить процесс преобразования данных с использованием открытых и закрытых ключей.

Результатом выполнения работы должен быть скрипт, который позволяет:

– зашифровать входные данные используя алгоритм Мак-Элиса.

– расшифровать зашифрованные данные обратно.

– продемонстрировать работоспособность алгоритма с различными текстами.

# 2 ХОД РАБОТЫ

Алгоритм был реализован на языке Python. Для выполнения задачи был создан объект, содержащий все методы, необходимые для шифрования и дешифрования данных. Алгоритм применяет алгоритм Мак-Элиса шифрования для входных данных и возвращает его. Далее скрипт выполняет дешифровку данных используя закрытый ключ.

Результат полного выполнения скрипта можно увидеть на рисунке 1 и рисунке 2.

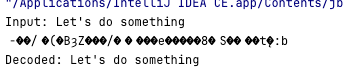
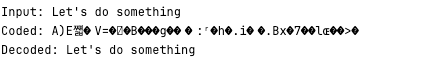


Рисунок 1 – Вывод работы программы



Как видно на рисунке 1 и рисунке 2 разный закодированный сообщения при одинаковых ключах. Это связано с внесение слуйчаной ошибки, которые исправляются при дешифровке. Скрипт успешно справился с шифровкой и дешифровкой информации. Одно из полей вывода содержит оригинальное значение.

# ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были изучены теоретические основы асимметричных шифров на примере алгоритм Мак-Элиса. Также на Kotlin был реализована его реализация с возможностью шифрование и расшифровки данных, используя набор открытого и закрытых ключей. Таким образом, поставленные цели были достигнуты: алгоритм был реализован, проведены эксперименты с его использованием, а также закреплены навыки программной реализации асимметричных блочных шифров.

# ПРИЛОЖЕНИЕ А

# (ОБЯЗАТЕЛЬНОЕ)

# ИСХОДНЫЙ КОД ПРОГРАММНОГО ПРОДУКТА

import numpy as np

import logging

# Настройка логгера

log = logging.getLogger("simple\_mceliece")

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(levelname)s - %(message)s')

# --- Вспомогательные функции для GF(2) на базе NumPy ---

def gf2\_matmul(A, B):

"""Умножение матриц над GF(2) (A @ B) % 2"""

return (np.array(A, dtype=int) @ np.array(B, dtype=int)) % 2

def gf2\_inv(A):

"""Обращение матрицы над GF(2) методом Гаусса"""

n = A.shape[0]

M = np.hstack((A, np.eye(n, dtype=int)))

for i in range(n):

pivot = i

while pivot < n and M[pivot, i] == 0:

pivot += 1

if pivot == n:

raise ValueError("Матрица необратима")

M[[i, pivot]] = M[[pivot, i]] # Перестановка строк

for j in range(n):

if i != j and M[j, i] == 1:

M[j] = (M[j] + M[i]) % 2 # Сложение строк (XOR)

return M[:, n:]

def generate\_hamming\_matrices(r):

"""

Генерирует систематические матрицы G и H для кода Хэмминга.

Параметры: r (число проверочных бит).

n = 2^r - 1, k = n - r. Исправляет t=1 ошибку.

"""

n = 2\*\*r - 1

k = n - r

# Создаем матрицу H = [P\_matrix | I\_r]

# P\_matrix состоит из столбцов, являющихся двоичным представлением

# чисел от 1 до 2^r-1, которые НЕ являются степенями двойки (имеют вес > 1).

P\_cols = []

Ir\_cols = []

identity\_indices = {2\*\*i for i in range(r)}

for i in range(1, n + 1):

# Двоичное представление (младший бит первый для удобства построения)

bin\_col = np.array([int(b) for b in format(i, f'0{r}b')[::-1]])

if i in identity\_indices:

# Это будет часть единичной матрицы Ir (нужно отсортировать)

pass

else:

P\_cols.append(bin\_col)

P\_matrix = np.array(P\_cols).T # r x k

Ir = np.eye(r, dtype=int) # r x r

Ik = np.eye(k, dtype=int) # k x k

# Систематическая форма: H = [P\_matrix | Ir]

H = np.hstack((P\_matrix, Ir))

# Соответствующая G = [Ik | P\_matrix.T]

G = np.hstack((Ik, P\_matrix.T))

return G, H, k, n

# --- Основной класс ---

class SimpleMcElieceCipher:

def \_\_init\_\_(self, r\_param=3):

"""

Инициализация с параметром r для кода Хэмминга.

Например, r=3 дает код (7,4), r=4 дает код (15,11).

Всегда исправляет t=1 ошибку.

"""

self.r = r\_param

self.t = 1 # Код Хэмминга исправляет 1 ошибку

# n и k будут определены при генерации ключей

self.n = None

self.k = None

log.info(f"SimpleMcEliece(Hamming r={self.r}) initiated. Target t={self.t}")

# Закрытые ключи

self.G = None # Порождающая матрица (k x n)

self.H = None # Проверочная матрица (r x n)

self.P = None # Матрица перестановки (n x n)

self.P\_inv = None

self.S = None # Скремблирующая матрица (k x k)

self.S\_inv = None

# Открытый ключ

self.Gp = None # G' = S \* G \* P

def generate\_random\_keys(self):

log.info("Generating keys...")

# 1. Генерация базового кода (Хэмминга)

self.G, self.H, self.k, self.n = generate\_hamming\_matrices(self.r)

log.debug(f"Hamming Code: k={self.k}, n={self.n}")

# 2. Генерация случайной матрицы перестановки P (n x n)

perm = np.random.permutation(self.n)

self.P = np.eye(self.n, dtype=int)[perm]

self.P\_inv = self.P.T # Для матриц перестановки обратная = транспонированной

# 3. Генерация обратимой скремблирующей матрицы S (k x k)

while True:

try:

S\_cand = np.random.randint(0, 2, (self.k, self.k))

self.S\_inv = gf2\_inv(S\_cand)

self.S = S\_cand

break

except ValueError:

# Матрица необратима, пробуем снова

continue

# 4. Вычисление открытого ключа Gp = (S \* G \* P) mod 2

self.Gp = gf2\_matmul(self.S, gf2\_matmul(self.G, self.P))

log.info(f"Keys generated. Public Key Gp shape: {self.Gp.shape}. Needs {self.k} message bits.")

def encrypt(self, msg\_arr):

msg\_arr = np.array(msg\_arr, dtype=int)

if len(msg\_arr) != self.k:

raise Exception(f"Wrong message length. Should be {self.k} bits.")

log.debug(f"Original msg: {msg\_arr}")

# c' = m \* Gp

c\_prime = gf2\_matmul(msg\_arr, self.Gp)

log.debug(f"Clean C': {c\_prime}")

# Внесение ровно ОДНОЙ ошибки (t=1)

error\_pos = np.random.randint(0, self.n)

log.info(f"Adding 1 error at position: {error\_pos}")

c\_prime[error\_pos] = (c\_prime[error\_pos] + 1) % 2

log.debug(f"Corrupted C': {c\_prime}")

return c\_prime

def \_decode\_hamming\_systematic(self, codeword):

"""Внутренняя функция декодирования (исправляет 1 ошибку и извлекает m)"""

# 1. Вычисление синдрома: s = c \* H^T

syndrome = gf2\_matmul(codeword, self.H.T)

if not np.all(syndrome == 0):

log.debug(f"Non-zero syndrome found: {syndrome}")

# Ищем столбец в H, совпадающий с синдромом

for i in range(self.n):

if np.array\_equal(self.H[:, i], syndrome):

log.info(f"REPAIRED ERROR ON {i}th POSITION")

codeword[i] = (codeword[i] + 1) % 2

break

else:

log.debug("Syndrome is zero. No errors.")

# 2. Извлечение сообщения.

# Т.к. G систематическая [Ik | P\_matrix.T], сообщение - это первые k бит.

return codeword[:self.k]

def decrypt(self, c\_prime\_arr):

c\_prime = np.array(c\_prime\_arr, dtype=int)

if len(c\_prime) != self.n:

raise Exception(f"Wrong ciphertext length. Should be {self.n} bits.")

log.debug(f"Received C': {c\_prime}")

# 1. Убираем перестановку: c\_hat = c' \* P\_inv

c\_hat = gf2\_matmul(c\_prime, self.P\_inv)

# 2. Декодируем код Хэмминга (исправляем ошибку и извлекаем m\_hat)

# m\_hat = m \* S

m\_hat = self.\_decode\_hamming\_systematic(c\_hat)

log.debug(f"Decoded m\_hat (m\*S): {m\_hat}")

# 3. Убираем скремблирование: m = m\_hat \* S\_inv

m = gf2\_matmul(m\_hat, self.S\_inv)

log.info(f"Decrypted msg: {m}")

return m

# --- Пример использования ---

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

# Параметр r=3 создает код Хэмминга (7,4). Длина сообщения k=4.

cipher = SimpleMcElieceCipher(r\_param=4)

cipher.generate\_random\_keys()

# Сообщение длиной k=4 бита

message = np.array([1, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1])

print("\n--- Encrypting ---")

ciphertext = cipher.encrypt(message)

print(f"Ciphertext: {ciphertext}")

print("\n--- Decrypting ---")

decrypted\_message = cipher.decrypt(ciphertext)

print(f"\nOriginal: {message}")

print(f"Decrypted: {decrypted\_message}")

assert np.array\_equal(message, decrypted\_message), "Decryption failed!"

print("\nSUCCESS: Message decrypted correctly after correcting 1 error.")