Министерство образования Республики Беларусь

Учреждение образования

БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Методы защиты информации

ОТЧЕТ

к лабораторной работе №4

на тему

**Асимметричная криптография.**

**Алгоритм Мак-Элиса.**

Выполнил Д.С. Шевцова

Проверил

Минск 2024

СОДЕРЖАНИЕ

[Введение 3](#_Toc158758843)

[1 Краткие теоретические сведения 4](#_Toc158758844)

[2 Результаты выполнения лабораторной работы](#_Toc158758845) 6

[Заключение](#_Toc158758846) 8

[Приложение А](#_Toc158758848) [(обязательное)](#_Toc158758849) [Листинг кода](#_Toc158758850) 9

## ВВЕДЕНИЕ

В данной лабораторной работе нужно реализовать программные средства шифрования и дешифрования текстовых файлов при помощи алгоритма Мак-Элиса для криптостойких размеров порождающей матрицы. Реализовать алгоритм для генерации криптостойких порождающих матриц. Обосновать выбор размера и характеристик матрицы.

## 1 КРАТКИЕ ТЕОРЕТИЧЕСКИЕ СВЕДЕНИЯ

Алгоритм Мак-Элиса — это асимметричная криптосистема, основанная на сложности задачи декодирования случайных линейных кодов. Это один из первых предложенных алгоритмов для шифрования с открытым ключом, и его безопасность обусловлена сложностью нахождения матриц, которые могут декодировать случайные линейные коды.

Как и любая асимметричная криптосистема, система Мак-Элиса использует два ключа: открытый ключ для шифрования сообщений, который может быть доступен любому, и закрытый ключ для расшифровки, известный только получателю сообщений.

Алгоритм начинается с выбора линейного кода, представленного в виде матрицы G, которая служит для кодирования сообщений. Однако матрица должна быть изменена таким образом, чтобы она выглядела случайно, сохраняя при этом возможность расшифровки для владельца закрытого ключа.

Выбираются две матрицы: матрица перестановки P, которая перемешивает строки матрицы, и матрица ошибок S, которая добавляет определённое количество ошибок.

Открытый ключ состоит из матрицы G′, которая является результатом перемножения исходной матрицы G, матрицы ошибок S, и перестановочной матрицы P: G′=S⋅G⋅P.

Закрытый ключ включает исходные матрицы S, G, и P, которые позволяют расшифровывать сообщения.

Сообщение, представленное в виде бинарного вектора, умножается на открытую матрицу G′. Это создаёт зашифрованный текст, который выглядит случайным из-за перемешивания и ошибок, добавленных через S и P.

На этом этапе сообщение становится зашифрованным, и его нельзя легко расшифровать без знания оригинальных матриц G, S, и P, которые присутствуют только в закрытом ключе.

Для расшифровки получатель сначала использует матрицу P, чтобы восстановить исходный порядок строк в зашифрованном сообщении.

Затем применяется матрица ошибок S, чтобы устранить добавленные ошибки, тем самым возвращая сообщение в его первоначальный вид.

Используя оригинальную матрицу G, сообщение может быть декодировано и преобразовано обратно в исходный текст. На рисунке 1.1 представлена блок схема алгоритма.

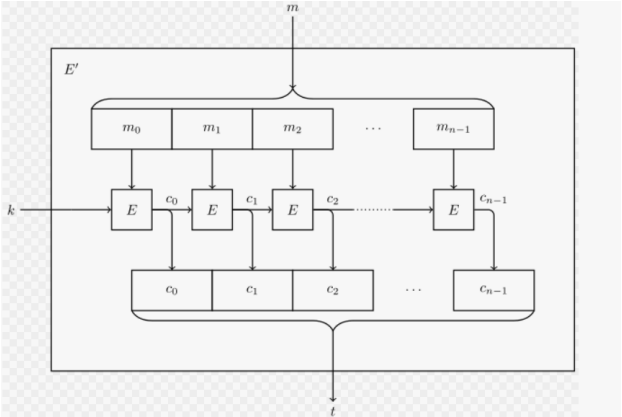


Рисунок 1.1 — Блок схема алгоритма

# 2 РЕЗУЛЬТАТЫ ВЫПОЛНЕНИЯ ЛАБОРАТОРНОЙ РАБОТЫ

Для шифрования и дешифрования файлом используется интерфейс командной строки, где указывается режим программы, а именно шифрования и дешифрования. На рисунке 2.1 указана команды для шифрования/дешифрования, а также зашифрованные данные. На рисунках 2.2 и 2.3 указан текст, с которым мы работаем и расшифрованные данные.

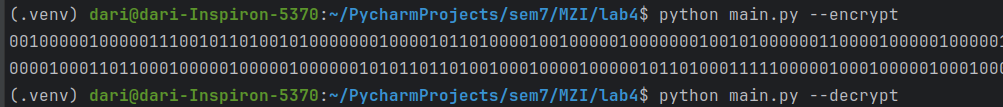


Рисунок 2.1 – Команда для шифрования и начальные данные

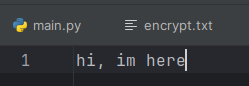


Рисунок 2.2 – Команда для дешифрования

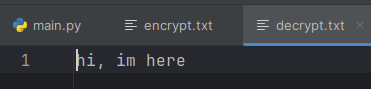


Рисунок 2.3 – Расшифрованный текст

Обоснование выбора размера и характеристик матрицы для алгоритма Мак-Элиса можно построить на основе следующих критериев: криптостойкость, производительность и допустимый уровень защиты для предполагаемой области применения.

Алгоритм Мак-Элиса основан на использовании кодов исправления ошибок, кодов Хэмминга, которые защищают сообщение от ошибок при шифровании и обеспечивают дополнительную криптостойкость. При увеличении размера матрицы растёт вычислительная нагрузка на систему. Поэтому выбор размера матрицы должен основываться на необходимости обеспечить приемлемую производительность при достаточной защите.

Если выбрать слишком маленький kgen (например, kgen=3, что даёт код (7,4)), это приводит к маленькому размеру ключа и низкой криптостойкости, что может сделать систему уязвимой для атак.

1 Если выбрать слишком большой kgen (например, kgen=12, что даёт код длиной 4095 бит), это приведёт к большому объёму вычислений и снижению производительности.

2 Оптимальный выбор — kgen от 10 до 11, что даёт длину кодового слова от 1023 до 2047 бит. Такие размеры достаточно велики для защиты от современных атак, но при этом обеспечивают разумную производительность при шифровании и дешифровании.

Выбор размера матрицы также должен зависеть от требуемого уровня защиты для задачи, которую решает система. В реальных системах криптостойкость должна соответствовать типу данных, которые нужно защитить.

1 Для обычных данных: Если вы работаете с данными средней важности, можно выбрать порождающую матрицу с размером kgen=10, что обеспечивает достаточную защиту при умеренных вычислительных затратах.

2 Для более критичных данных: Если данные требуют высокой криптостойкости (например, при передаче конфиденциальной информации), можно увеличить kgen до 11 или выше, что даст длину кодового слова до 2047 бит. Это значительно повысит уровень защиты, хотя и увеличит вычислительные затраты.

## ЗАКЛЮЧЕНИЕ

В ходе выполнения данной лабораторной работы была написана реализация программных средств шифрования и дешифрования текстовых файлов при помощи алгоритма Мак-Элиса для криптостойких размеров порождающей матрицы

Реализован алгоритм для генерации криптостойких порождающих матриц. Обоснованы выбор размера и характеристик матрицы.

# **ПРИЛОЖЕНИЕ А**

## (обязательное)

## Листинг кода

utilits.py

import numpy as np

import math

class hamming\_keygen:

"Generates a public key for a message of size 2^kgen-1. Returns G^, the public key, as well as S G and P (Private Key Components)"

def \_\_init\_\_(self, kgen):

self.kgen = kgen

self.k = 2\*\*self.kgen - self.kgen -1

self.n = 2\*\*self.kgen -1

self.G = self.genHammingMatrix()

self.S = self.genInvertibleMatrix()

self.P = self.genPermuteMatrix()

self.Gcarat = np.matmul(np.matmul(self.S, self.G), self.P) % 2

def genHammingMatrix(self):

"""Generates Hamming Matrix given k and n"""

identity = np.identity(self.kgen)

identityk = np.identity(self.k)

left = np.zeros((self.kgen, 2\*\*self.kgen - 1 - self.kgen)).T

rowcount = 0

for i in range(2\*\*self.kgen):

if i + 1 != 1:

if (i + 1) & i != 0:

binarystring = np.binary\_repr(i+1)

column = np.zeros((len(binarystring), 1))

for i in range(len(binarystring)):

column[-i - 1] = binarystring[i]

column = np.pad(column, (0, self.kgen - len(binarystring)), 'constant')

left[rowcount] = column.T[0]

rowcount += 1

left = left.T

self.paritycheck = np.block([left, identity])

self.generator = np.block([identityk, np.transpose(left)])

return self.generator

def genInvertibleMatrix(self):

"Generates S ,an invertible matrix of size K\*l"

S = np.random.randint(0,2,(self.k, self.k), dtype=np.uint)

while np.linalg.det(S) == 0:

S = np.random.randint(0,2,(self.k, self.k), dtype=np.uint)

return S

def genPermuteMatrix(self):

"Generates P, a random perumutation of the identity matrix"

P = np.identity(self.n, dtype=np.uint)

return P[np.random.permutation(self.n)]

class Encoder:

'"Encodes a given message, m, using public key g\_prime, by permuting and then adding errors to the message."'

def \_\_init\_\_(self, m, g\_prime, t=1):

self.g\_prime = g\_prime

self.message = m

(k, n) = g\_prime.shape

self.k = k

self.n = n

self.t = t

self.z = self.generate\_errors()

self.encoded = self.encode()

def generate\_errors(self):

"""Generates 1 random errors in bitstring of length n"""

self.z = np.zeros(self.n)

idx\_list = np.random.choice(self.n, self.t, replace=False)

for idx in idx\_list:

self.z[idx] = 1

return self.z

def encode(self):

"""Encode message by multiplying by G^, and add random error."""

self.c\_prime = np.matmul(self.message, self.g\_prime) % 2

c = (self.c\_prime + self.z) % 2

return c

def get\_message(self):

return self.message

def get\_encrypted(self):

return self.encoded

class decoder:

def \_\_init\_\_(self, c, S, P, H, m):

self.c = c

self.S = S

self.P = P

self.H = H

self.m = m

self.decrypted = self.decrypt()

self.correct = (self.m == self.decrypted)

def decrypt(self):

"Decrypts a given message given the private key (S, G, and P)"

P\_inv = np.linalg.inv(self.P)

S\_inv = np.linalg.inv(self.S)

c\_prime = np.matmul(self.c, P\_inv)

m\_prime = self.error\_correction(c\_prime)

decrypted = np.matmul(m\_prime, S\_inv) % 2

return decrypted

def error\_correction(self, c\_prime):

"Finds the error based on the calculated parity matrix, and flips that bit."

parity = np.matmul(c\_prime, np.transpose(self.H)) % 2

parity\_bits = np.ma.size(parity, 0)

parity\_total = 0

#Calculate Syndrome (From binary representation to integer)

for i in range(parity\_bits):

parity\_total += 2\*\*i \* parity[i]

#if no errors, return the message

if (int((parity\_total - 1)) & int(parity\_total)) == 0:

return c\_prime[0:(c\_prime.size - parity\_bits)]

#otherwise, flip the bit with an error in it. Note math.ceil(np.log2(parity\_total)-1) is to switch from standard form.

else:

error\_message = c\_prime

error\_bit = int(parity\_total - math.ceil(np.log2(parity\_total)) - 1)

if error\_message[error\_bit] == 1:

error\_message[error\_bit] = 0

return error\_message[0:(c\_prime.size - parity\_bits)]

elif error\_message[error\_bit] == 0:

error\_message[error\_bit] = 1

return error\_message[0:(c\_prime.size - parity\_bits)]

else:

...

def is\_correct(self):

return self.correct8

main.py

import sys

import numpy as np

import utilits as ut

def split\_string\_into\_blocks(input\_string):

current\_block = ""

blocks = []

for char in input\_string:

char\_binary = bin(int(char, 16))[2:].zfill(16)

current\_block += char\_binary

block = []

for i in range(len(current\_block)):

if len(block) == 4:

blocks.append(block)

block = []

block.append(int(current\_block[i]))

blocks.append(block)

return blocks

def blocks\_into\_string(blocks):

proto\_string = ''

string = ''

for i in range(len(blocks)):

for j in range(4):

proto\_string += str(int(blocks[i][j]))

for i in range(0, len(proto\_string), 16):

delta\_string = proto\_string[i:i + 16]

string += chr(int(hex(int(delta\_string, 2)), 16))

return string

def main():

with open('text.txt', 'r', encoding='utf8') as file:

source\_text = file.read()

bin\_text = [hex(ord(elem)) for elem in source\_text]

blocks = split\_string\_into\_blocks(bin\_text)

key\_info = ut.hamming\_keygen(3)

g\_prime = key\_info.Gcarat

enc = []

check = []

for i in range(len(blocks)):

encoded = ut.Encoder(np.array(blocks[i]), g\_prime)

message = encoded.get\_message()

encrypted = encoded.get\_encrypted()

enc.append(encrypted)

check.append(message)

dec = []

for i in range(len(enc)):

decoded = ut.decoder(enc[i], key\_info.S, key\_info.P, key\_info.paritycheck, check[i])

dec.append(decoded.decrypted)

dec\_string = blocks\_into\_string(dec)

enc\_string = ''.join([''.join(map(str, map(int, sublist))) for sublist in enc])

if sys.argv[1] == '--encrypt':

with open('encrypt.txt', 'w', encoding='utf8') as file:

# file.write(dec\_string)

file.write(enc\_string)

print(enc\_string)

if sys.argv[1] == '--decrypt':

with open('decrypt.txt', 'w', encoding='utf8') as file:

file.write(dec\_string)

# else:

# print(sys.argv[1])

return 0

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

main()