ГУАП

КАФЕДРА 33

ОТЧЕТ   
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| доцент |  |  |  | А.В.Окатов |
| должность, уч. степень, звание |  | подпись, дата |  | инициалы, фамилия |

|  |
| --- |
| ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №3  ШИФР С УПРАВЛЯЕМЫМИ ПЕРЕСТАНОВКАМИ |
|  |
| по курсу: ЗАЩИТА ИНФОРМАЦИИ |
|  |
|  |

РАБОТУ ВЫПОЛНИЛ

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| СТУДЕНТ ГР. № | 4941 |  |  |  | Н. С. Горбунов |
|  |  |  | подпись, дата |  | инициалы, фамилия |

Санкт-Петербург 2022

**Цель работы:** изучить принципы работы метода шифрования с управляемыми перестановками, а также оценить количество раундов шифрования, необходимых для качественного сокрытия изображения.

**Ход работы**

Для обработки входного изображения был выбран блок , который предоставит уникальных вариантов перестановки битов, что приближено к теоретическому значению

Схема перестановок представлена на рисунке 1.

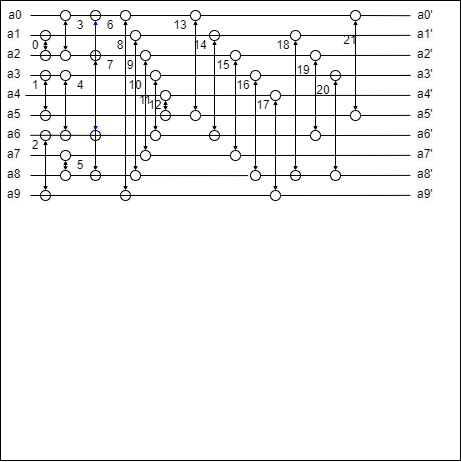


Рисунок 1. Схема перестановок

Для применения такой схемы перестановок необходимо будет размножить биты управляющего блока. Это будет выполнено с помощью операции XOR с определенными операндами.

**Тестирование алгоритма**

На рисунке 2 представлено исходное изображение



Рисунок 2. Исходное изображение

Гистограмма исходного изображения приведена на рисунке 3. Тест на решетчатость приведен на рисунке 4.

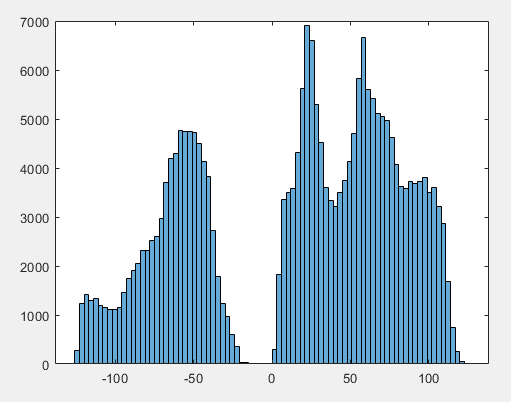


Рисунок 3.

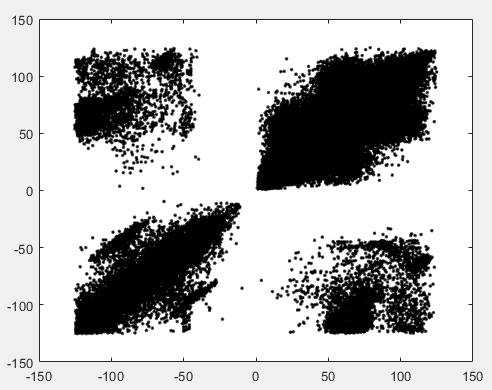


Рисунок 4.

Исходя из визуального анализа статистических тестов рисунков 3 и 4 можно сделать вывод, о том, что в приведенном массиве данных существуют закономерности. На графике теста на решетчатость видно неравномерность распределения значений пикселей.

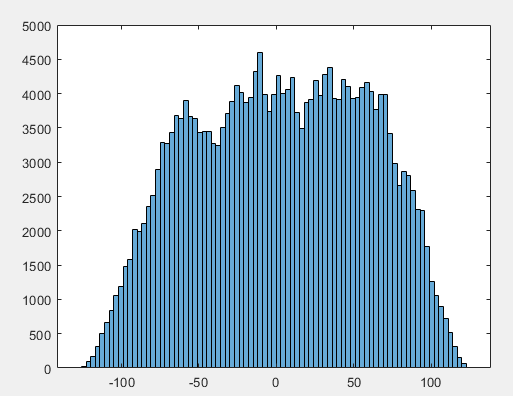
**Шифрование изображения**

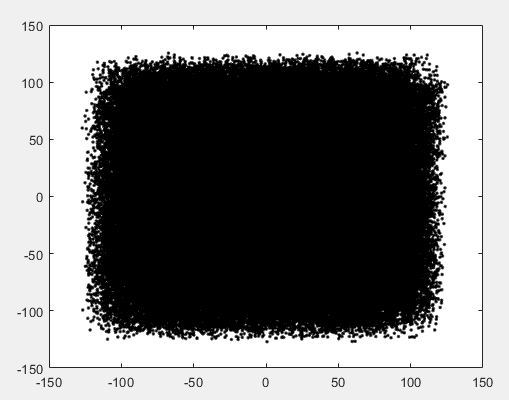
***После первого раунда шифрования изображение принимает вид:***



Рисунок 5. Первый раунд шифрования

Статистические тесты:





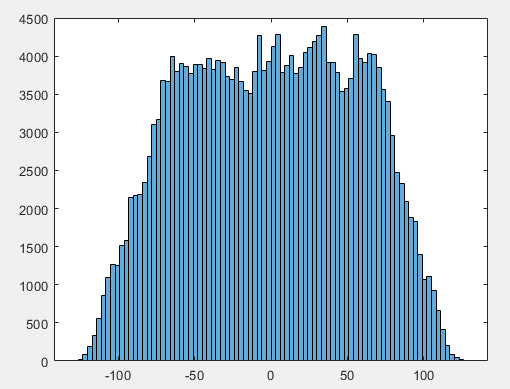
Гистограмма и тест на решетчатость существенно изменились после первого раунда шифрования. Изображение зашифровалось, но силуэты ещё различимы.

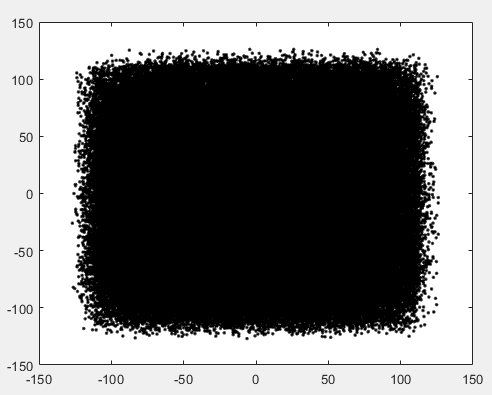
***После второго раунда шифрования изображение принимает вид:***



Рисунок 6. Второй раунд шифрования

Статистические тесты:





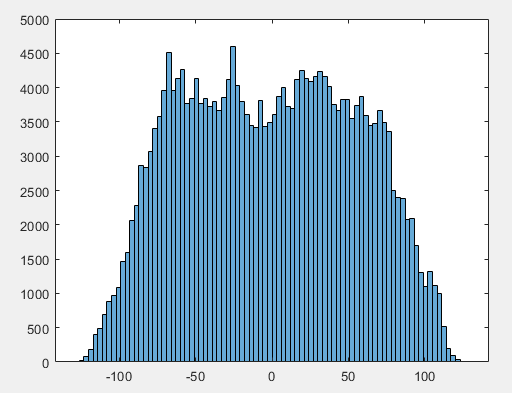
Гистограмма приобрела очертания нормального распределения значений интенсивности с широким плато в центре. Тест на решетчатость существенных изменений не приобрел.

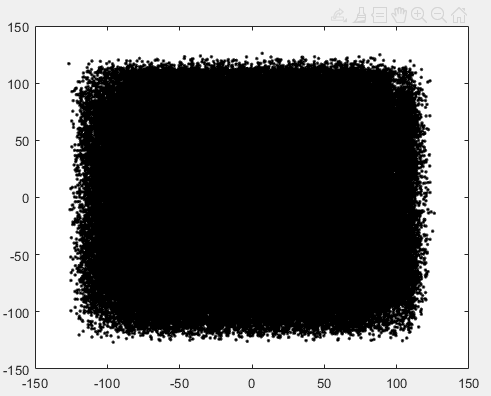
***После седьмого раунда шифрования изображение принимает вид:***



Рисунок 7. Седьмой раунд шифрования.

Статистические тесты:





После 7 раундов шифрования гистограмма похожа на трапецию с крутыми краями, что в принципе, можно принять за равномерное распределение. А тест на решетчатость показывает случайное расположение элементов без видимых зависимостей. Визуальный тест также показывает абсолютное отсутствие контуров на изображении и его шумообразный вид. Элементы нормально распределены на плоскости.

Вывод: 7 раундов шифрования достаточно.

**Дешифрование изображения:**

Дешифрование происходит семью повторениями алгоритма, так как изображение было зашифровано с помощью семи раундов. Результат расшифровки приведет на рисунке 8.



Рисунок 8. Результат дешифрования.

**Вывод:** в ходе данной работы был изучен принцип работы шифрования с управляемыми перестановками. Для определения оптимального количества раундов, необходимого для сокрытия информации при использовании выбранной управляемой перестановки, к исходному изображению и результатам каждого раунда были применены статистические тесты, показывающие наличие зависимостей в изображении. В результате было выяснено, что для реализованной перестановки достаточно семи раундов для полного превращения изображения в шум. Также можно сделать вывод, что количество раундов, требующихся для шифрования, зависит от выбранного варианта перестановки и размера блока.

Приложение 1. Шифратор

import java.io.BufferedOutputStream;  
import java.io.File;  
import java.io.FileOutputStream;  
import java.io.IOException;  
import java.nio.file.Files;  
  
public class ImageEncryptor {  
  
 private static int *BLOCK\_SIZE* = 20;  
  
 //BGR - порядок кодирования битов  
 //формула вычисления интенсивности пикселя по стандарту BT-709 Y = 0.2125·R + 0.7154·G + 0.0721·B  
 public static void main(String[] Args) throws IOException {  
 *EncryptImageBySPBlock*("src/main/Photos/picture.bmp", "src/main/Photos/output.bmp");  
 }  
  
  
 public static void EncryptImageBySPBlock(String source, String destination) throws IOException {  
 byte[] bitmap = *ReadBytes*(source); //чтение  
 bitmap = *encrypted\_bitmap*(bitmap); //шифрование  
 *WriteBytesToFile*(destination, bitmap); //запись  
 }  
  
 public static void WriteBytesToFile(String source, byte[] bytes) throws IOException {  
 File output\_file = new File(source);  
 BufferedOutputStream output\_stream = new BufferedOutputStream(new FileOutputStream(output\_file));  
 output\_stream.write(bytes);  
 output\_stream.flush();  
 output\_stream.close();  
 }  
  
 public static byte[] ReadBytes(String source) throws IOException {  
 File input\_file = new File(source);  
 return Files.*readAllBytes*(input\_file.toPath());  
 }  
  
 public static byte[] encrypted\_bitmap(byte[] bitmap) {  
 int pixels\_count = bitmap.length;  
 byte[] output\_bitmap = new byte[pixels\_count];  
 //заполняем шапку файла  
 System.*arraycopy*(bitmap, 0, output\_bitmap, 0, 54);  
 //то есть это 30 байтов, кодирующие 2 последовательных пикселя  
 int data\_block = 30; //ДЛИНА БЛОКА ДАННЫХ  
 for (int index = 54; index < pixels\_count; index += data\_block) {  
 //здесь на каждой итерации обрабатываем 30 последовательных байтов в значимой массе байтов  
 //и заполняем их биты в общий битовый блок bits на 240 разрядов  
 byte[] bits = new byte[data\_block \* 8]; //блок данных  
 for (int byte\_index = 0; byte\_index < data\_block; byte\_index++) { //читаю по порядку байты одного блока  
 for (int bit = byte\_index \* 8 + 7; bit >= byte\_index \* 8; bit--) { //читаю биты одного байта из блока  
 if ((bitmap[index + byte\_index] % 2 == 1) || (bitmap[index + byte\_index] % 2 == -1)) bits[bit] = 1;  
 bitmap[index + byte\_index] = (byte) (bitmap[index + byte\_index] >> 1);  
 }  
 }  
 //обработка блока данных S и P блоками  
 bits = *BlockEncryptor*(bits);  
 //далее происходит запись в выходную карту байтов  
 for (int i = index; i < index + data\_block; i++) {  
 int pre\_byte = 0;  
 for (int local\_bit = (i - index) \* 8; local\_bit < (i - index) \* 8 + 8; local\_bit++) {  
 pre\_byte += bits[local\_bit] \* Math.*pow*(2, (7 - (local\_bit - ((i - index) \* 8))));  
 }  
 output\_bitmap[i] = (byte) pre\_byte;  
 }  
 }  
 return output\_bitmap;  
  
 }  
  
 public static byte[] BlockEncryptor(byte[] block) {  
 byte[] output = block;  
 int rounds = 7;  
 for (int index = 0; index < rounds; index++) {  
 output = *EncruptWithControlPermutation*(output);  
 }  
 return output;  
 }  
  
 public static byte[] EncruptWithControlPermutation(byte[] block) { //Блок шифрования управляемой перестановкой  
 byte[] output = new byte[block.length];  
 int block\_size = *BLOCK\_SIZE*;  
 int count\_of\_S\_blocks = block.length / block\_size;  
 for (int S\_index = 0; S\_index < count\_of\_S\_blocks; S\_index++) {  
 int sourcePosition = S\_index \* block\_size;  
 byte[] LeftPart = new byte[block\_size / 2];  
 byte[] RightPart = new byte[block\_size / 2];  
 System.*arraycopy*(block, sourcePosition, LeftPart, 0, block\_size / 2);  
 System.*arraycopy*(block, sourcePosition + block\_size / 2, RightPart, 0, block\_size / 2);  
  
 *PermuteFirstBySecond*(LeftPart, RightPart); //Переставляем биты левого блока под управлением правого  
 LeftPart = *BlockXOR\_LEFT*(LeftPart, RightPart); //К левому блоку прибавляем по модулю два правый  
 *PermuteFirstBySecond*(RightPart, LeftPart); //Переставляем биты правого блока под управлением левого  
 RightPart = *BlockXOR\_LEFT*(RightPart, LeftPart); //К левому блоку прибавляем по модулю два правый  
 LeftPart = *AddKey*(LeftPart, new byte[]{0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1}); //Добавление ключа к левому блоку  
 RightPart = *AddKey*(RightPart, new byte[]{1, 0, 1, 0, 1, 0, 1, 0, 1, 0, 1, 0}); //Добавление ключа к правому блоку  
  
 System.*arraycopy*(LeftPart, 0, output, sourcePosition, block\_size / 2);  
 System.*arraycopy*(RightPart, 0, output, sourcePosition + block\_size / 2, block\_size / 2);  
 }  
 return output;  
  
 }  
  
 public static byte[] PermuteFirstBySecond(byte[] LeftPart, byte[] RightPart) {  
 for (int i = 0; i < LeftPart.length; i++) {  
 byte buffer;  
 if (RightPart[i] == 1) {  
 Permutation permutation = Permutation.*values*()[i];  
 buffer = LeftPart[permutation.a];  
 LeftPart[permutation.a] = LeftPart[permutation.b];  
 LeftPart[permutation.b] = LeftPart[buffer];  
 }  
 }  
 //*TODO Как то инкапсулировать это?* //блок дополнительных сгенерированных перестановок  
 byte[] genericPermutations = new byte[12];  
 genericPermutations[0] = *xor*(RightPart[0], RightPart[1]);  
 genericPermutations[1] = *xor*(RightPart[1], RightPart[2]);  
 genericPermutations[2] = *xor*(RightPart[2], RightPart[3]);  
 genericPermutations[3] = *xor*(RightPart[3], RightPart[4]);  
 genericPermutations[4] = *xor*(RightPart[4], RightPart[5]);  
 genericPermutations[5] = *xor*(RightPart[5], RightPart[6]);  
 genericPermutations[6] = *xor*(RightPart[6], RightPart[7]);  
 genericPermutations[7] = *xor*(RightPart[7], RightPart[8]);  
 genericPermutations[8] = *xor*(RightPart[8], RightPart[9]);  
 genericPermutations[9] = *xor*(RightPart[9], RightPart[1]);  
 genericPermutations[10] = *xor*(RightPart[1], RightPart[8]);  
 genericPermutations[11] = *xor*(RightPart[2], RightPart[7]);  
 for (int i = 0; i < genericPermutations.length; i++) {  
 byte buffer;  
 if (genericPermutations[i] == 1) {  
 Permutation permutation = Permutation.*values*()[i + *BLOCK\_SIZE*/2];  
 buffer = LeftPart[permutation.a];  
 LeftPart[permutation.a] = LeftPart[permutation.b];  
 LeftPart[permutation.b] = LeftPart[buffer];  
  
 }  
 }  
 return LeftPart;  
 }  
  
 public static byte xor(byte A, byte B) {  
 return (byte) (A == B ? 1 : 0);  
 }  
  
 public static byte[] BlockXOR\_LEFT(byte[] FirstBlock, byte[] SecondBlock) {  
 byte[] output = new byte[*BLOCK\_SIZE*/2];  
 for (int i = 0; i < output.length; i++) {  
 output[i] = *xor*(FirstBlock[i], SecondBlock[i]);  
 }  
 return output;  
 }  
  
 public static byte[] AddKey(byte[] Block, byte[] key) {  
 byte[] output = new byte[*BLOCK\_SIZE*/2];  
 for (int i = 0; i < output.length; i++) output[i] = *xor*(Block[i], key[i]);  
 return output;  
 }  
}

Приложение 2. Схема перестановок

public enum Permutation {  
 *PER\_1*(1,2),  
 *PER\_2*(3,5),  
 *PER\_3*(6,9),  
 *PER\_4*(0,2),  
 *PER\_5*(3,6),  
 *PER\_6*(7,8),  
 *PER\_7*(0,6),  
 *PER\_8*(2,8),  
 *PER\_9*(0,9),  
 *PER\_10*(1,8),  
 *PER\_11*(2,7),  
 *PER\_12*(3,6),  
 *PER\_13*(4,5),  
 *PER\_14*(0,5),  
 *PER\_15*(1,6),  
 *PER\_16*(2,7),  
 *PER\_17*(3,8),  
 *PER\_18*(4,9),  
 *PER\_19*(1,8),  
 *PER\_20*(2,6),  
 *PER\_21*(3,8),  
 *PER\_22*(0,5),  
  
 ;  
  
  
 public final int a;  
 public final int b;  
  
 Permutation(int a, int b) {  
 this.a = a;  
 this.b = b;  
 }  
}

Приложение 3. Подготовка данных для математического моделирования

import java.io.File;  
import java.io.FileWriter;  
import java.io.IOException;  
import java.nio.file.Files;  
  
public class MathTesting {  
 public static void main (String[] args) throws IOException {  
 byte[] bitmap = *ReadBytes*("src/main/resources/output/output6.bmp");  
 byte[] targeted\_bitmap = new byte[bitmap.length-54];  
 System.*arraycopy*(bitmap, 54, targeted\_bitmap, 0, bitmap.length-54);  
 float[] ready\_vector = new float[targeted\_bitmap.length/3];  
 //вычисляем интенсивность  
 for (int pixel\_index = 0; pixel\_index < ready\_vector.length; pixel\_index++){  
 ready\_vector[pixel\_index] = ((float) (targeted\_bitmap[pixel\_index\*3] \* 0.0721))+ //BLUE  
 ((float) (targeted\_bitmap[pixel\_index\*3+1] \* 0.7154))+ //GREEN  
 ((float) (targeted\_bitmap[pixel\_index\*3+2] \* 0.2125)); //RED  
 }  
 try(FileWriter writer = new FileWriter("src/main/NumericData/source.txt", false))  
 {  
 for (int i = 0; i < ready\_vector.length; i++){  
 writer.write(String.*valueOf*(ready\_vector[i]));  
 writer.append('\n');  
 }  
 writer.flush();}  
 catch(IOException ex){  
 System.*out*.println(ex.getMessage());  
 }  
 try(FileWriter writer2 = new FileWriter("src/main/NumericData/source1.txt", false))  
 {  
 for (int i = 1; i < ready\_vector.length; i++){  
 writer2.write(String.*valueOf*(ready\_vector[i]));  
 writer2.append('\n');  
 }  
 writer2.write(String.*valueOf*(ready\_vector[ready\_vector.length-1]));  
 writer2.flush();  
 }  
 catch(IOException ex){  
 System.*out*.println(ex.getMessage());  
 }  
 }  
 public static byte[] ReadBytes (String source) throws IOException {  
 File input\_file = new File(source);  
 return Files.*readAllBytes*(input\_file.toPath());  
 }  
  
}

Приложение 4. M-файл для графических тестов

%читаем первый файл

fileID = fopen('source.txt', 'r');

formatSpec = '%f';

sizeA = [1 Inf];

A = fscanf(fileID, formatSpec, sizeA);

A = A';

fclose(fileID);

%читаем второй файл

fileID1 = fopen('source1.txt', 'r');

formatSpec1 = '%f';

sizeB = [1 Inf];

B = fscanf(fileID1, formatSpec1, sizeB);

B = B';

fclose(fileID1);

histogram(A);

figure();

plot (A, B, 'k.');