МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
федеральное государственное автономное образовательное учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ АЭРОКОСМИЧЕСКОГО ПРИБОРОСТРОЕНИЯ»

КАФЕДРА № 25

ОТЧЕТ
ЗАЩИЩЕН С ОЦЕНКОЙ

ПРЕПОДАВАТЕЛЬ

старший преподаватель		М. Н. Исаева
должность, уч. степень, звание	подпись, дата	инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ № 2

БЛОКОВЫЕ ШИФРЫ

по курсу: КРИПТОГРАФИЧЕСКИЕ МЕТОДЫ ЗАЩИТЫ ИНФОРМАЦИИ

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ ГР. № 2155		Э.Р. Курманалиев
	подпись, дата	инициалы, фамилия

г. Санкт-Петербург
2023

1 Описание задания

Вариант 3. Реализовать алгоритм шифрования AES.

1 Реализовать алгоритм шифрования согласно варианту, предусмотреть возможность работы алгоритма в режиме CBC.

2. Исследовать процесс распространения ошибок в реализуемом режиме шифрования, привести пример распространения ошибок.

3. По результатам анализа сделать выводы о качестве реализованных систем шифрования.

Цель: Реализация функционала, способного шифровать тестовый файл и изображение с помощью алгоритма шифрования AES в CBC режиме.

Описание работы алгоритма

Rijndael — итеративный блоковый алгоритм с переменной длиной информационного блока и переменной длиной ключа. Длина информационного блока и длина ключа не зависят друг от друга и могут принимать значения 128, 192 или 256 бит.

Состоянием в Rijndael называется промежуточный результат процесса шифрования. Состояние можно представить как прямоугольный массив, элементами которого являются байты. В этом массиве имеется четыре строки, а число столбцов обозначается, как N_b и равно длине информационного блока, деленной на 32 (Рис. 4.8). Ключ, используемый для шифрации, можно представить аналогичным образом в виде прямоугольного массива, имеющего четыре строки, число столбцов, в котором обозначается, как N_k и равно длине ключа, деленной на 32 (Рис. 2.21). В некоторых случаях такие матрицы можно представить как одномерные массивы, элементами которых являются четырехбайтные вектора. Такие четырехбайтные вектора будем называть словами. При этом каждый такой вектор представляет собой соответствующий столбец из матричного представления. Такие одномерные массивы будут иметь длину соответственно 4, 6 или 8.

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$

$k_{0,0}$	$k_{0,1}$	$k_{0,2}$	$k_{0,3}$
$k_{1,0}$	$k_{1,1}$	$k_{1,2}$	$k_{1,3}$
$k_{2,0}$	$k_{2,1}$	$k_{2,2}$	$k_{2,3}$
$k_{3,0}$	$k_{3,1}$	$k_{3,2}$	$k_{3,3}$

Рисунок 1 – Пример состояния (при Nb = 6) и ключа шифрования (при = 4)

Информационный блок на входе и шифрограмма на выходе алгоритма Rijndael могут быть представлены как одномерные массивы, элементами которых являются байты, пронумерованные от 0 до 4*N_b - 1. Таким образом, эти блоки имеют длину 16, 24 или 32 байта, и значения индексов лежат в диапазоне 0, ..., 15; 0, ..., 23 или 0, ..., 31. Ключ, используемый в алгоритме шифрования, представляет собой одномерный массив, элементами которого являются байты, пронумерованные от 0 до . Таким образом, блок ключа имеет длину 16, 24 или 32 байта, и значения индексов лежат в диапазоне 0, ..., 15; 0, ..., 23 или 0, ..., 31. Байты шифруемой информации ("исходного текста") записываются в байты состояния в следующем порядке: $a_{0,0}$, $a_{1,0}$, $a_{2,0}$, $a_{3,0}$, $a_{0,1}$, $a_{1,1}$, $a_{2,1}$, $a_{3,1}$, $a_{0,2}$, ..., и байты ключа шифрования записываются в соответствующий массив как $k_{0,0}$, $k_{1,0}$, $k_{2,0}$, $k_{3,0}$, $k_{0,1}$, $k_{1,1}$, $k_{2,1}$, $k_{3,1}$, $k_{0,2}$, ... По окончании процесса шифрования результирующие байты состояния переписываются в шифрограмму в таком же порядке. Таким образом, если индекс байта в одномерном массиве равен n, а соответствующий индекс в двумерном массиве — (i, j) то

i = n mod 4; j=n/4; n = i + 4 * j;

Количество этапов (циклов) алгоритма Nr зависит от величин Nb и Nk и определяется в соответствии с таблицей 1

N_r	$N_b = 4$	$N_b = 6$	$N_b = 8$
$N_k = 4$	10	12	14
$N_k = 6$	12	12	14
$N_k = 8$	14	14	14

Таблица 1 - Количество циклов как функция от длины информационного блока и длины ключа

На каждом этапе алгоритма выполняется четыре различные преобразования информационного блока:

- Побайтовая подстановка;
- Сдвиг по строке;
- Побайтовая перестановка внутри столбцов;
- Сложение информационного блока и ключа, используемого на данном этапе.

На последнем этапе выполняются лишь три преобразования:

- Побайтовая подстановка;
- Сдвиг по строке;
- Сложение информационного блока и ключа, используемого на данном этапе

Тесты.

Шифрование текса из файла test.txt

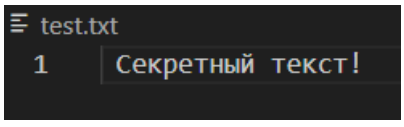

Рисунок 2 – исходный файл с текстом

Для шифрация был использован сгенерированный ключ: 2r$У"╠№SIbY↔3‼ЕдU╠◄┴│╩ъ╕a.╣┼Rk╚

Рисунок 3 – зашифрованный текст

Тест шифрация картинки в bmp формате.

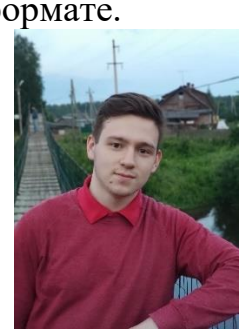

Рисунок 4 – входная картинка

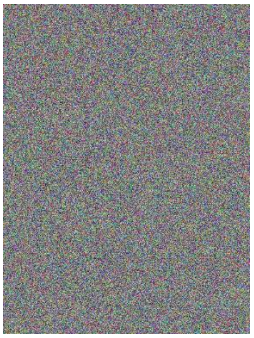

Рисунок 5 – зашифрованная картинка

На рисунке 6 показано что картинка output.bmp зашифрована с ошибкой и она отличается от картинки справа, которая была зашифрована без ошибки

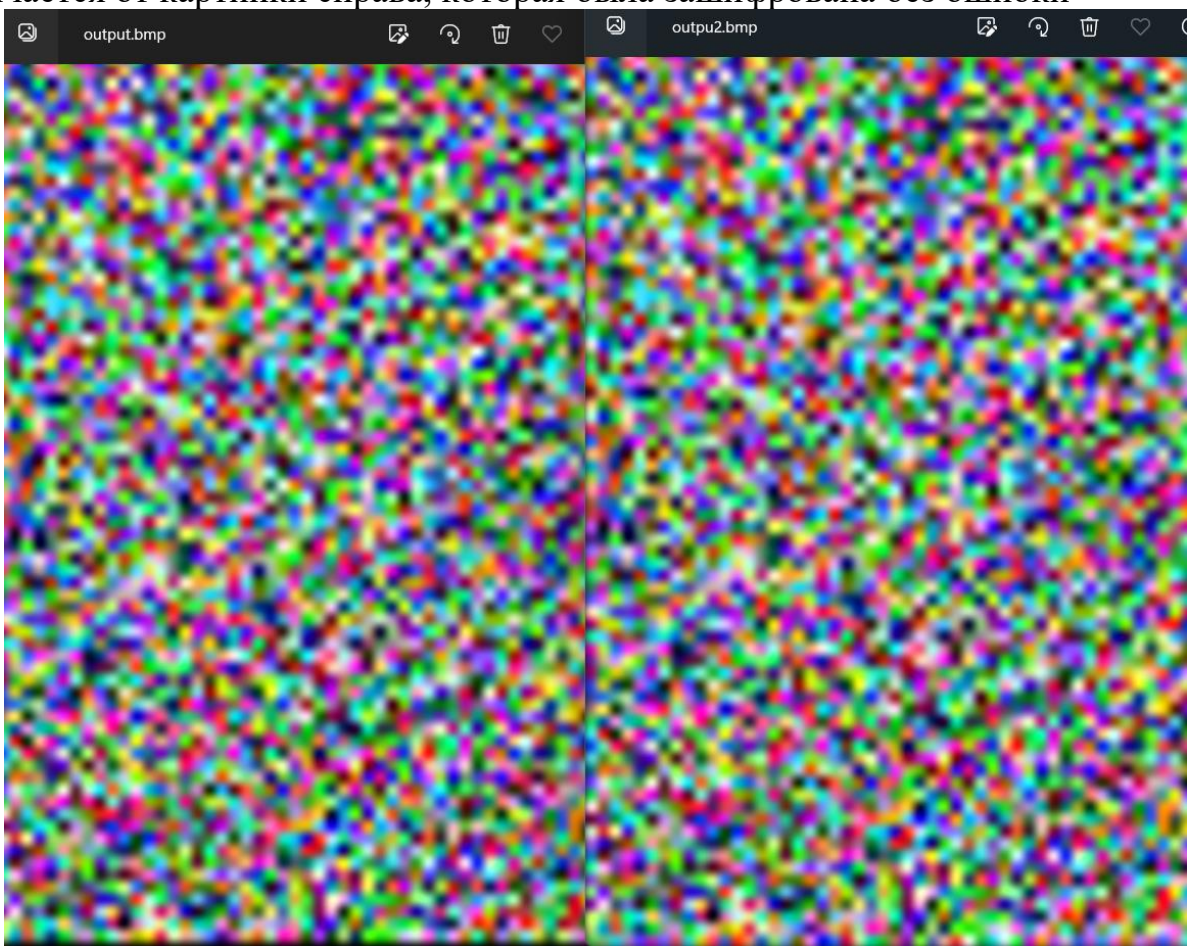

Рисунок 6 – шифрация с ошибкой

Рисунок 7 – расшифрованная картинка, которая была зашифрована с ошибкой

Коэффициент корреляции

$\hat{r}_{A,B} = \frac{\widehat{M}[(A-\widehat{M}[A])(B-B\widehat{M}[B]}{\hat{\sigma}_A\hat{\sigma}_B}$, где A и B - компоненты изображения, $\widehat{M}[]$ – оценка математического ожидания, $\hat{\sigma}_A\hat{\sigma}_B$- оценки среднеквадратичного отклонения компонент A и B.

Коэффициент корреляции в обычном режиме шифрования: 0.000315256

Коэффициент корреляции в режиме шифрования CBC: 0.000132296

Выводы

В результате выполнения лабораторной работы был реализован алгоритм шифрования AES с CBC режимом.

Алгоритм был проверен на шифровании картинки и текстового файла, так же была добавлена ошибка в зашифрованную картинку, чтобы проверить как алгоритм справиться с ней. Как показали результаты, то 1 ошибка портит дешифрацию всего блока.

Также была рассчитана корреляция исходной картинки и зашифрованной в обычном режиме и CBC.

Листинг кода

```
#include "AES.h"
#include <iostream>
#include <fstream>
#include <vector>
using namespace std;
AES::AES(const AESKeyLength keyLength)
{
   switch (keyLength)
   {
   case AESKeyLength::AES_128:
      this->Nk = 4;
      this->Nr = 10;
      break;
   case AESKeyLength::AES_192:
      this->Nk = 6;
      this->Nr = 12;
      break;
   case AESKeyLength::AES_256:
      this->Nk = 8;
      this->Nr = 14;
      break;
   }
}

unsigned char *AES::EncryptCBC(unsigned char in[], unsigned int inLen,
                           const unsigned char key[],
                           const unsigned char *iv)
{

   unsigned char *out = new unsigned char[inLen];
   unsigned char block[blockBytesLen];
   unsigned char *roundKeys = new unsigned char[4 * Nb * (Nr + 1)];
   KeyExpansion(key, roundKeys);
   memcpy(block, iv, blockBytesLen);
   unsigned int i = 0;
   for (i = 0; i < inLen; i += blockBytesLen)
   {
      XorBlocks(block, in + i, block, blockBytesLen);
      EncryptBlock(block, out + i, roundKeys);
      memcpy(block, out + i, blockBytesLen);
   }
   delete[] roundKeys;

   return out;
}

unsigned char *AES::DecryptCBC(const unsigned char in[], unsigned int inLen,
                           const unsigned char key[],
                           const unsigned char *iv)
{
   unsigned char *out = new unsigned char[inLen];
   unsigned char block[blockBytesLen];
   unsigned char *roundKeys = new unsigned char[4 * Nb * (Nr + 1)];
   KeyExpansion(key, roundKeys);
```

```
    memcpy(block, iv, blockBytesLen);
    for (unsigned int i = 0; i < inLen; i += blockBytesLen)
    {
        DecryptBlock(in + i, out + i, roundKeys);
        XorBlocks(block, out + i, out + i, blockBytesLen);
        memcpy(block, in + i, blockBytesLen);
    }

    delete[] roundKeys;

    return out;
}

void AES::EncryptBlock(const unsigned char in[], unsigned char out[],
                       unsigned char *roundKeys)
{
    unsigned char state[4][Nb];
    unsigned int i, j, round;

    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            state[i][j] = in[i + 4 * j];
        }
    }

    AddRoundKey(state, roundKeys);

    for (round = 1; round <= Nr - 1; round++)
    {
        SubBytes(state);
        ShiftRows(state);
        MixColumns(state);
        AddRoundKey(state, roundKeys + round * 4 * Nb);
    }

    SubBytes(state);
    ShiftRows(state);
    AddRoundKey(state, roundKeys + Nr * 4 * Nb);

    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            out[i + 4 * j] = state[i][j];
        }
    }
}

void AES::DecryptBlock(const unsigned char in[], unsigned char out[],
                       unsigned char *roundKeys)
{
    unsigned char state[4][Nb];
    unsigned int i, j, round;
```

```
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            state[i][j] = in[i + 4 * j];
        }
    }

    AddRoundKey(state, roundKeys + Nr * 4 * Nb);

    for (round = Nr - 1; round >= 1; round--)
    {
        InvSubBytes(state);
        InvShiftRows(state);
        AddRoundKey(state, roundKeys + round * 4 * Nb);
        InvMixColumns(state);
    }

    InvSubBytes(state);
    InvShiftRows(state);
    AddRoundKey(state, roundKeys);

    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            out[i + 4 * j] = state[i][j];
        }
    }
}

void AES::SubBytes(unsigned char state[4][Nb])
{
    unsigned int i, j;
    unsigned char t;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            t = state[i][j];
            state[i][j] = sbox[t / 16][t % 16];
        }
    }
}

void AES::ShiftRow(unsigned char state[4][Nb], unsigned int i,
                   unsigned int n)
{
    unsigned char tmp[Nb];
    for (unsigned int j = 0; j < Nb; j++)
    {
        tmp[j] = state[i][(j + n) % Nb];
    }
    memcpy(state[i], tmp, Nb * sizeof(unsigned char));
```

```
}

void AES::ShiftRows(unsigned char state[4][Nb])
{
   ShiftRow(state, 1, 1);
   ShiftRow(state, 2, 2);
   ShiftRow(state, 3, 3);
}

unsigned char AES::xtime(unsigned char b) // multiply on x
{
   return (b << 1) ^ (((b >> 7) & 1) * 0x1b);
}

void mixSingleColumn(unsigned char *r)
{
   unsigned char a[4];
   unsigned char b[4];
   unsigned char c;
   unsigned char h;

   for (c = 0; c < 4; c++)
   {
      a[c] = r[c];

      h = (unsigned char)((signed char)r[c] >> 7);
      b[c] = r[c] << 1;
      b[c] ^= 0x1B & h;
   }
   r[0] = b[0] ^ a[3] ^ a[2] ^ b[1] ^ a[1];
   r[1] = b[1] ^ a[0] ^ a[3] ^ b[2] ^ a[2];
   r[2] = b[2] ^ a[1] ^ a[0] ^ b[3] ^ a[3];
   r[3] = b[3] ^ a[2] ^ a[1] ^ b[0] ^ a[0];
}

void AES::MixColumns(unsigned char state[4][Nb])
{
   unsigned char *temp = new unsigned char[4];

   for (int i = 0; i < 4; ++i)
   {
      for (int j = 0; j < 4; ++j)
      {
         temp[j] = state[j][i];
      }
      mixSingleColumn(temp);
      for (int j = 0; j < 4; ++j)
      {
         state[j][i] = temp[j];
      }
   }
   delete temp;
}

void AES::AddRoundKey(unsigned char state[4][Nb], unsigned char *key)
```

```
{
    unsigned int i, j;
    for (i = 0; i < 4; i++)
    {
        for (j = 0; j < Nb; j++)
        {
            state[i][j] = state[i][j] ^ key[i + 4 * j];
        }
    }
}

// void AES::SubWord(unsigned char *a)
// {
//     int i;
//     for (i = 0; i < 4; i++)
//     {
//         a[i] = sbox[a[i] / 16][a[i] % 16];
//     }
// }

void AES::SubWord(unsigned char *a)
{
    unsigned int a1[] = {0b10001111, 0b11000111, 0b11100011, 0b11110001, 0b11111000, 0b01111100,
0b00111110, 0b00011111};
    unsigned int b = 0b11000110;
    for (int i = 0; i < 4; i++)
    {
        a[i] = (a1[i] * a[i]) ^ b;
    }
}

void AES::RotWord(unsigned char *a)
{
    unsigned char c = a[0];
    a[0] = a[1];
    a[1] = a[2];
    a[2] = a[3];
    a[3] = c;
}

void AES::XorWords(unsigned char *a, unsigned char *b, unsigned char *c)
{
    int i;
    for (i = 0; i < 4; i++)
    {
        c[i] = a[i] ^ b[i];
    }
}

void AES::Rcon(unsigned char *a, unsigned int n)
{
    unsigned int i;
    unsigned char c = 1;
    for (i = 0; i < n - 1; i++)
    {
```

```
        c = xtime(c);
    }

    a[0] = c;
    a[1] = a[2] = a[3] = 0;
}

void AES::KeyExpansion(const unsigned char key[], unsigned char w[])
{
    unsigned char temp[4];
    unsigned char rcon[4];

    unsigned int i = 0;
    while (i < 4 * Nk)
    {
        w[i] = key[i];
        i++;
    }

    i = 4 * Nk;
    while (i < 4 * Nb * (Nr + 1))
    {
        temp[0] = w[i - 4 + 0];
        temp[1] = w[i - 4 + 1];
        temp[2] = w[i - 4 + 2];
        temp[3] = w[i - 4 + 3];

        if (i / 4 % Nk == 0)
        {
            RotWord(temp);
            SubWord(temp);
            Rcon(rcon, i / (Nk * 4));
            XorWords(temp, rcon, temp);
        }
        else if (Nk > 6 && i / 4 % Nk == 4)
        {
            SubWord(temp);
        }

        w[i + 0] = w[i - 4 * Nk] ^ temp[0];
        w[i + 1] = w[i + 1 - 4 * Nk] ^ temp[1];
        w[i + 2] = w[i + 2 - 4 * Nk] ^ temp[2];
        w[i + 3] = w[i + 3 - 4 * Nk] ^ temp[3];
        i += 4;
    }
}

void AES::InvSubBytes(unsigned char state[4][Nb])
{
    unsigned char t;
    for (int i = 0; i < 4; i++)
    {
        for (int j = 0; j < Nb; j++)
        {
            t = state[i][j];
```

```
            state[i][j] = inv_sbox[t / 16][t % 16];
        }
    }
}

void AES::InvMixColumns(unsigned char state[4][Nb])
{
    unsigned char temp_state[4][Nb];

    for (size_t i = 0; i < 4; ++i)
    {
        memset(temp_state[i], 0, 4);
    }

    for (size_t i = 0; i < 4; ++i)
    {
        for (size_t k = 0; k < 4; ++k)
        {
            for (size_t j = 0; j < 4; ++j)
            {
                temp_state[i][j] ^= GF_MUL_TABLE[INV_CMDS[i][k]][state[k][j]];
            }
        }
    }

    for (size_t i = 0; i < 4; ++i)
    {
        memcpy(state[i], temp_state[i], 4);
    }
}

void AES::InvShiftRows(unsigned char state[4][Nb])
{
    ShiftRow(state, 1, Nb - 1);
    ShiftRow(state, 2, Nb - 2);
    ShiftRow(state, 3, Nb - 3);
}

void AES::XorBlocks(const unsigned char *a, const unsigned char *b,
                    unsigned char *c, unsigned int len)
{
    for (unsigned int i = 0; i < len; i++)
    {
        c[i] = a[i] ^ b[i];
    }
}

void AES::printHexArray(unsigned char a[], unsigned int n)
{
    for (unsigned int i = 0; i < n; i++)
    {
        printf("%02x ", a[i]);
    }
}
```

```
void AES::printHexVector(std::vector<unsigned char> a)
{
   for (unsigned int i = 0; i < a.size(); i++)
   {
      printf("%02x ", a[i]);
   }
}

std::vector<unsigned char> AES::ArrayToVector(unsigned char *a,
                                    unsigned int len)
{
   std::vector<unsigned char> v(a, a + len * sizeof(unsigned char));
   return v;
}

unsigned char *AES::VectorToArray(std::vector<unsigned char> &a)
{
   return a.data();
}

std::vector<unsigned char> AES::EncryptCBC(std::vector<unsigned char> in,
                                 std::vector<unsigned char> key,
                                 std::vector<unsigned char> iv)
{
   while (in.size() % blockBytesLen != 0)
   {
      in.push_back('\0');
   }
   unsigned char *out = EncryptCBC(VectorToArray(in), (unsigned int)in.size(),
                           VectorToArray(key), VectorToArray(iv));
   std::vector<unsigned char> v = ArrayToVector(out, in.size());
   delete[] out;
   return v;
}

std::vector<unsigned char> AES::DecryptCBC(std::vector<unsigned char> in,
                                 std::vector<unsigned char> key,
                                 std::vector<unsigned char> iv)
{
   unsigned char *out = DecryptCBC(VectorToArray(in), (unsigned int)in.size(),
                           VectorToArray(key), VectorToArray(iv));
   std::vector<unsigned char> v = ArrayToVector(out, (unsigned int)in.size());
   delete[] out;
   while (v.back() == '\0')
   {
      v.pop_back();
   }
   return v;
}

std::vector<uint8_t> AES::bytesFromFile(std::string inputFilePath)
{
   std::vector<uint8_t> bytesOfFile;
   std::ifstream file(inputFilePath, std::ios::binary);
```

```
    if (file)
    {
        file.seekg(0, std::ios::end);
        std::streampos fileSize = file.tellg();
        file.seekg(0, std::ios::beg);
        bytesOfFile.resize(static_cast<size_t>(fileSize));
        file.read(reinterpret_cast<char *>(bytesOfFile.data()), fileSize);
        file.close();
    }
    else
    {
        throw std::runtime_error("Failed to read file: " + inputFilePath);
    }
    this->dataBytes = bytesOfFile;
    return bytesOfFile;
}

std::string AES::fileFromBytes(const std::string &outputFilePath, std::vector<unsigned char> inp)
{
    std::vector<uint8_t> combinedBytes(headerBytes);
    combinedBytes.insert(combinedBytes.end(), inp.begin(), inp.end());

    std::ofstream file(outputFilePath, std::ios::binary);

    if (file)
    {
        file.write(reinterpret_cast<const char *>(combinedBytes.data()), combinedBytes.size());
        file.close();
        return outputFilePath;
    }
    else
    {
        std::cerr << "Failed to create or write to file: " << outputFilePath << std::endl;
        throw std::runtime_error("Failed to create or write to file: " + outputFilePath);
    }
}

std::vector<unsigned char> AES::bytesFromImage(std::string inputFilePath)
{
    std::ifstream file(inputFilePath, std::ios::binary);
    if (!file)
    {
        throw std::runtime_error("Failed to read file: " + inputFilePath);
    }
    const size_t headerSize = 54;
    headerBytes.resize(headerSize);
    file.read(reinterpret_cast<char *>(headerBytes.data()), headerSize);

    file.seekg(0, std::ios::end);
    const size_t fileSize = file.tellg();
    const size_t pixelDataSize = fileSize - headerSize;

    dataBytes.resize(pixelDataSize);
    file.seekg(headerSize);
    file.read(reinterpret_cast<char *>(dataBytes.data()), pixelDataSize);
```

```
    file.close();

    return dataBytes;
}
```