Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет информационных технологий
Кафедра информационных систем и технологий
Специальность 1-40 05 01 «Информационные системы и технологии»
Специализация 1-40 05 01 03«Информационные системы и технологии» (издательско-полиграфический комплекс)

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА КУРСОВОГО ПРОЕКТА

по дисциплине Защита информации и надёжность информационных систем
Тема: Применение эллиптических кривых в криптосистеме Эль-Гамаля и метода замены битовых плоскостей для сокрытия зашифрованных данных в изображении

Исполнитель
студент 3 курса группы 2 _____ Р. С. Давыдов
подпись, дата

Руководитель
Преподаватель-стажёр _____ А. А. Сущеня
подпись, дата

Курсовая работа защищена с оценкой _____
Руководитель _____ А. А. Сущеня
подпись

Минск 2019

Оглавление

1. Введение

В 1985 году независимо Нилом Коблицем и Виктором Миллером было предложено использовать в криптографии алгебраические свойства эллиптических кривых. С этого момента началось бурное развитие нового направления криптографии, для которого используется термин "криптография на эллиптических кривых". Роль основной криптографической операции выполняет операция скалярного умножения точки на эллиптической кривой на данное целое число, определяемая через операции сложения и удвоения точек эллиптической кривой. Последние, в свою очередь, выполняются на основе операций сложения, умножения и инвертирования в конечном поле, над которыми рассматривается кривая. Особый интерес к криптографии эллиптических кривых обусловлен теми преимуществами, которые дает её применение в беспроводных коммуникациях — высокое быстродействие и небольшая длина ключа Асимметричная криптография основана на сложности решения некоторых математических задач. Ранние криптосистемы с открытым ключом, такие как алгоритм RSA, криптостойки благодаря тому, что сложно разложить составное число на простые множители. При использовании алгоритмов на эллиптических кривых полагается, что не существует субэкспоненциальных алгоритмов для решения задачи дискретного логарифмирования в группах их точек. При этом порядок группы точек эллиптической кривой определяет сложность задачи. Считается, что для достижения такого же уровня криптостойкости как и в RSA, требуются группы меньших порядков, что уменьшает затраты на хранение и передачу информации.

Стеганография – это способ передачи или хранения информации с учётом сохранения в тайне самого факта такой передачи (хранения). В отличие от криптографии, которая скрывает содержимое тайного сообщения, стеганография скрывает сам факт его существования. Как правило, сообщение будет выглядеть как что-либо иное, например, как изображение, статья, список покупок, письмо или судоку. Стеганографию обычно используют совместно с методами криптографии, таким образом, дополняя её.

Преимущество стеганографии над чистой криптографией состоит в том, что сообщения не привлекают к себе внимания. Сообщения, факт шифрования которых не скрыт, вызывают подозрение и могут быть сами по себе уличающими в тех странах, в которых запрещена криптография[1]. Таким образом, криптография защищает содержание сообщения, а стеганография защищает сам факт наличия каких-либо скрытых посланий.

2. Постановка задачи

Необходимо реализовать вариацию алгоритма Эль-Гамаля, которая основана на применении эллиптических кривых для получения ключей шифрования. Также необходимо реализовать метод вставки зашифрованного текста в файл картинки и последующего получения вставленной информации из картинки. Создать приложение, которое включает в себя всё вышеперечисленное.

3. Описание метода

При использовании алгоритмов шифрования на эллиптических кривых полагается, что не существует быстрых алгоритмов для решения задачи дискретного логарифмирования в группах их точек. В настоящий момент известны лишь экспоненциальные алгоритмы вычисления обратных функций для эллиптических кривых. По сравнению с субэкспоненциальными алгоритмами разложения числа на простые сомножители (как, например, в RSA), это позволяет при одинаковом уровне стойкости уменьшить размерность ключа в несколько раз, а, следовательно, упростить программную и аппаратную реализацию криптосистем.

Процедура создания ключей будет основываться на эллиптической кривой и состоять из следующих этапов:

1. Выбор модуля эллиптической кривой – простое число p.

   Например пусть p = 41.

2. Выбор коэффициентов эллиптической кривой A и B.

   A и B должны быть такими, чтобы уравнение $(4A^3 + 27B^2)modp \neq 0$ было верным. Поэтому пусть A = 3 и B = 7.

3. Нахождение точки эллиптической кривой $P(x_p, y_p)$ и порядка циклической подгруппы группы точек эллиптической кривой q

   Для этого принимается произвольное x_p $(0 < x_p < p)$ и находится y_p из уравнения эллиптической кривой $y^2 modp = (x^3 + Ax + B)modp$. Так мы получили точку $P(x_p, y_p)$. Затем находится порядок q следующим образом:

   - вычисляем коэффициент $\lambda = \frac{(3x_1^2+a)}{2y_1} modp = \frac{3*7^2+3}{2*17} mod41 = 2$.

   - координаты второй точки получаем по формулам $x_2 = (\lambda^2 - 2x_1)modp = (2^2 - 2 * 7)mod41 = 31$.

   - каждую следующую точку находим по формулам $\lambda_i = \frac{y_{i-1} - y_1}{x_{i-1} - x_1} modp$, $x_i = (\lambda_i^2 - x_1 - x_{i-1})modp$ и $y_i = (\lambda_i(x_1 - x_i) - y_1)modp$. Делаем это до тех пор, пока в знаменателе первой формулы не будет получен 0.

   - к полученному числу точек добавляем точку O. В нашем примере это будет q = 46 + 1 = 47.

   При этом нужно чтобы q получилось простым. Если оно не простое, то этап 3 выполняется заново с другими x_p и y_p.

4. Тот, кто создаёт открытый ключ и, соответственно, будет расшифровывать сообщения выбирает закрытый ключ d (0 < d < q).

Пусть d = 10 для данного примера.

5. Находится точка кривой $Q(x_q, y_q) = d * P(x_p, y_p)$.

В примере $x_q = 36$ и $y_p = 20$.

6. Публикуется открытый ключ $[(A, B), P(x_q, y_q), p, Q(x_q, y_q)]$.

В нашем примере открытый ключ = [(3, 7), (7, 17), 41, (36, 20)].

Процедура шифрования будет выглядеть следующим образом:

№ п/п	Описание операции	Пример
1	Определяется десятичное представление буквы **t**.	Буква «К» t = 12
2	Выбирается случайное число **k** (0 < k < n).	k = 5
3	Определяется точка $\mathbf{P_k(x_{pk}, y_{pk})}$ = k * P.	P_k(25, 2)
4	Определяется точка $\mathbf{Q_k(x_{qk}, y_{qk})}$ = k * Q.	Q_k(3, 24)
5	Вычисляется **c** = (t * x_{qk}) mod n.	c = (12 * 3) mod 41 = 36
6	Шифрограмма – пара [P_k, c].	[P_k(25, 2), 36]

Рисунок 3.1 – шифрование алгоритмом Эль-Гамаля на основе эллиптической кривой

И расшифровка будет выполняться так:

№ п/п	Описание операции	Пример
1	Вычисляется точка $\mathbf{D(x_d, y_d)}$ = d * P_k.	D(3, 24)
2	Вычисляется десятичное представление зашифрованной буквы **t** = (c * x_d^{-1}) mod n, где x_d^{-1} – обратное число к x_d по модулю n.	x_d^{-1} = 14 [(3 * 14) mod 41 = 1] t = (36 * 14) mod 41 = 12
3	Определяется исходное сообщение по ее десятичному представлению.	Буква «К»

Рисунок 3.2 – расшифровка сообщения, зашифрованного шифрование алгоритмом Эль-Гамаля на основе эллиптической кривой

4. Описание программного средства

Программное средство было разработано в IDE Visual Studio на языке C#. Оно выполняет следующие функции:

- зашифровывает введённые текстовые данные

- вставляет зашифрованные данные в изображение

- достаёт информацию из изображения

- расшифровывает полученное из изображения зашифрованное сообщение

Интерфейс приложения выглядит так:

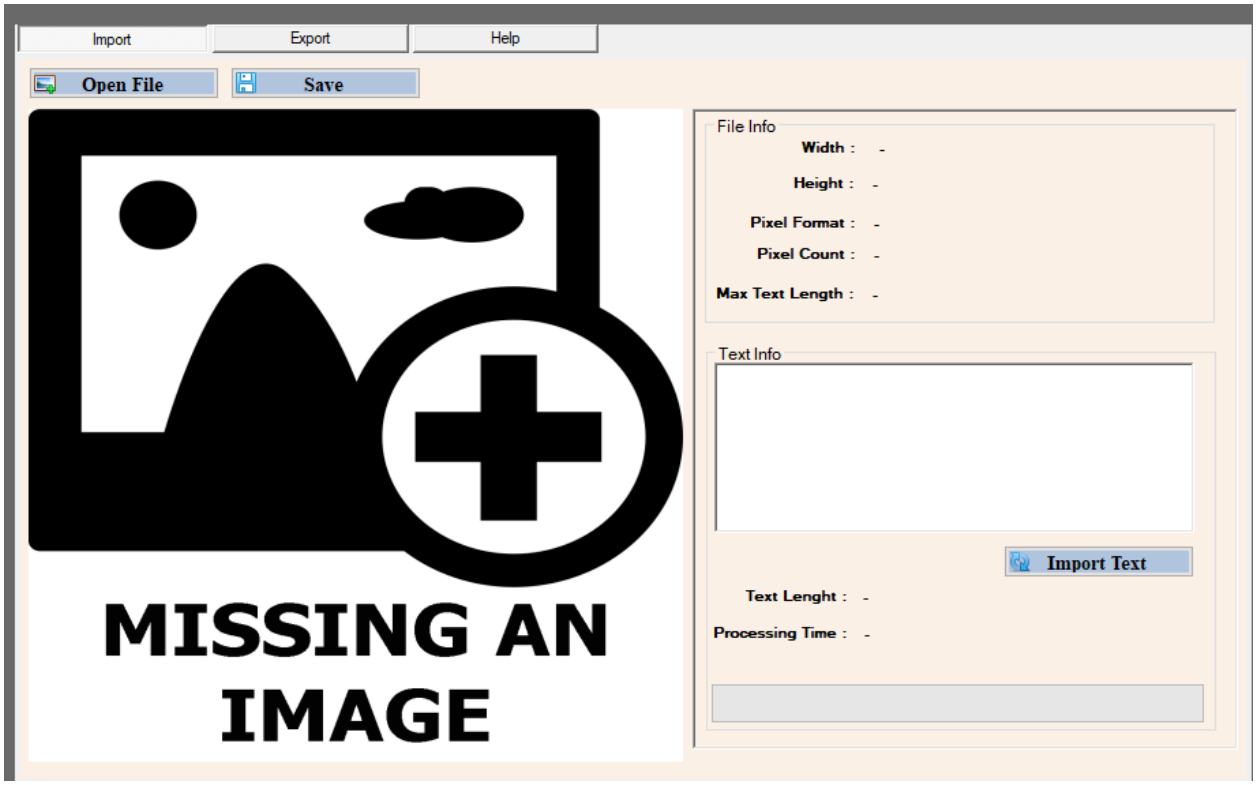

Рисунок 4.1 – интерфейс приложения

В верхней части приложения расположен выбор режима работы “Import”, “Export” и пояснение к тому, как пользоваться приложением во вкладке “Help”. В окне “Import” текстовое поле справа служит для ввода текста, соответственно в окне “Export” в текстовое поле выводится расшифрованный текст.

При нажатии на кнопку Open File открывается новое окно, в котором необходимо выбрать изображение, в которое будет помещён зашифрованный текст

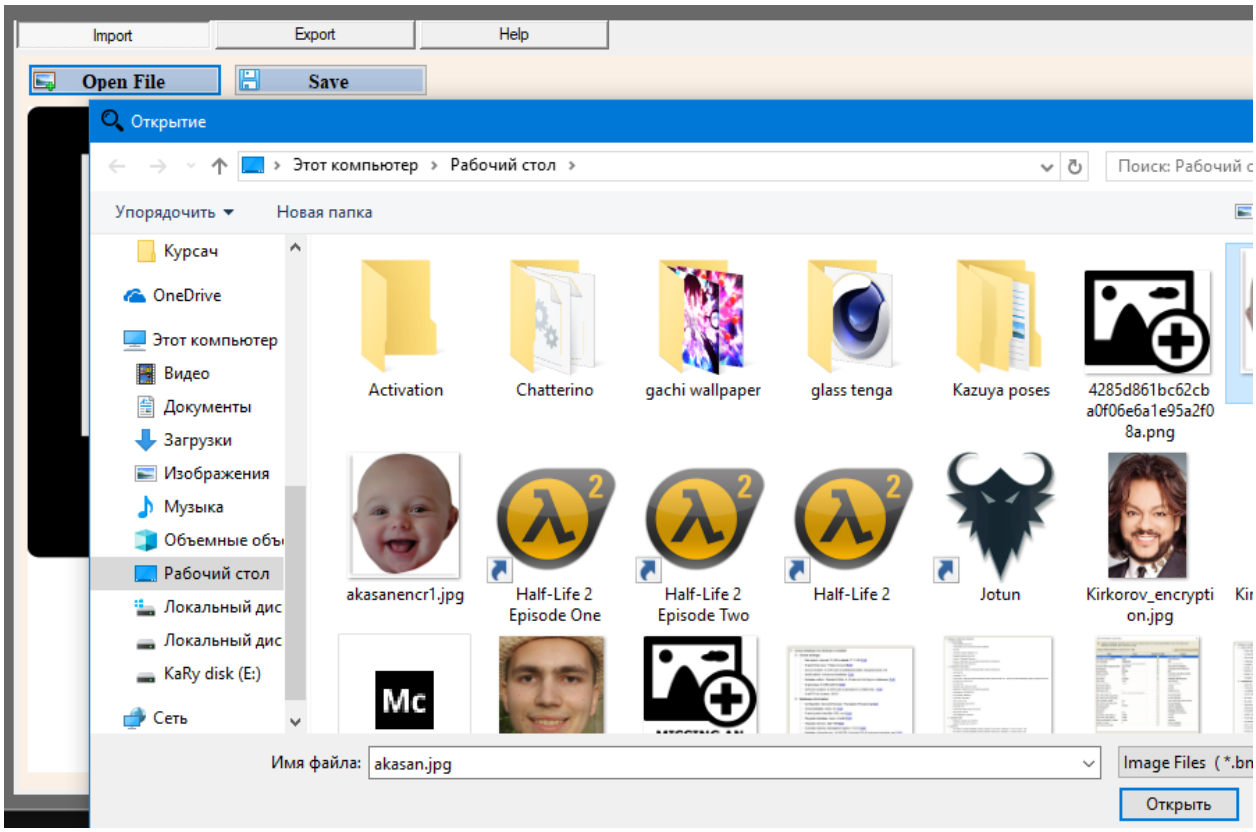

Рисунок 4.2 – окно выбора изображения

Также, в правом нижнем и верхнем углах выводится некоторая информация о выбранной картинке (в верхнем) и о зашифрованном тексте (в нижнем):

Рисунок 4.3 – некоторая информация о картинке и зашифрованном тексте

5. Тестирование программного средства

Тестирование производится с целью проверки того, что данные на самом деле шифруются и дешифруются.

Сначала введём какую-нибудь строку, нажмём кнопку “Import Text” чтобы зашифровать сообщение и вставить его в картинку и затем кнопку “Save” чтобы сохранить это изображение:

5.1 – Шифрование строки и помещение её в изображение

Затем перейдём во вкладку “Export”, выберем изображение с зашифрованным текстом и нажмём на кнопку “Export Text” чтобы получить зашифрованное сообщение:

Рисунок 5.2 – получение зашифрованного сообщения

Каждые 3 последовательности цифр – это один зашифрованный символ исходного сообщения. Чтобы показать, что шифрование на самом деле работает, я закомментировал строку расшифровки специально для этого тестирования потому что в нормальном режиме работы приложение сразу выдаёт расшифрованное сообщение.

```
_form.ExportTextBoxText = importedText;
//_form.ExportTextBoxText = EllipticCurvesEncryption.DecryptString(importedText);
SetInfoLabels();
```

Рисунок 5.3 – изменение для вывода шифротекста

А нормальный код выглядит так:

```
//_form.ExportTextBoxText = importedText;
_form.ExportTextBoxText = EllipticCurvesEncryption.DecryptString(importedText);
SetInfoLabels();
```

Рисунок 5.4 – код для вывода сразу расшифрованного сообщения

И результат будет выглядеть так:

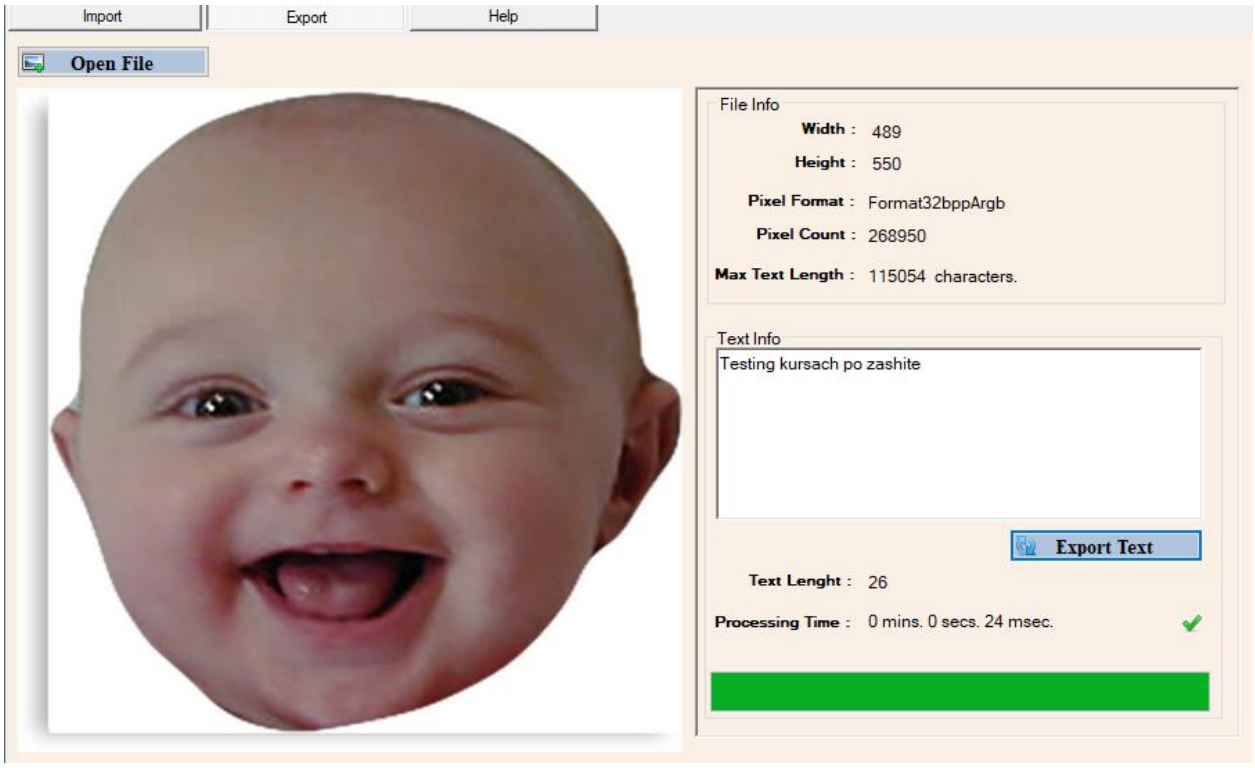

Рисунок 5.5 – вывод расшифрованного сообщения

6. Руководство пользователя

Процесс работы с приложением довольно прост. Для того чтобы зашифровать сообщение и поместить его в изображение следует произвести следующие этапы:

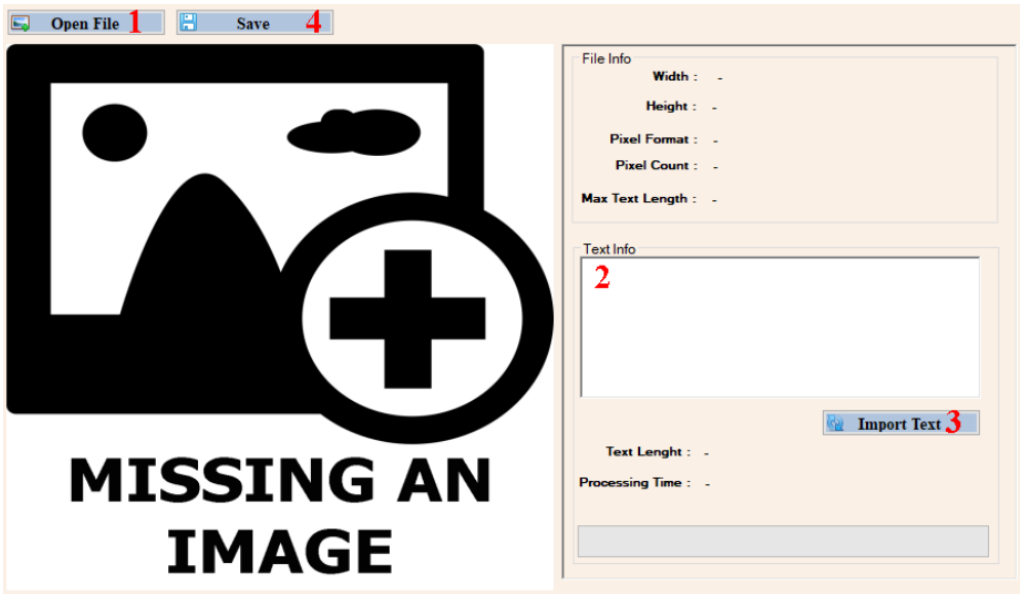

Рисунок 6.1 – работа во вкладке “Import”

Т. е. сначала открываем изображение, в которое хотим поместить шифротекст, вводим сообщение в текстовое поле, шифруем и, наконец, сохраняем изображение со вставленным в него шифротекстом.

Процесс экспорта следующий:

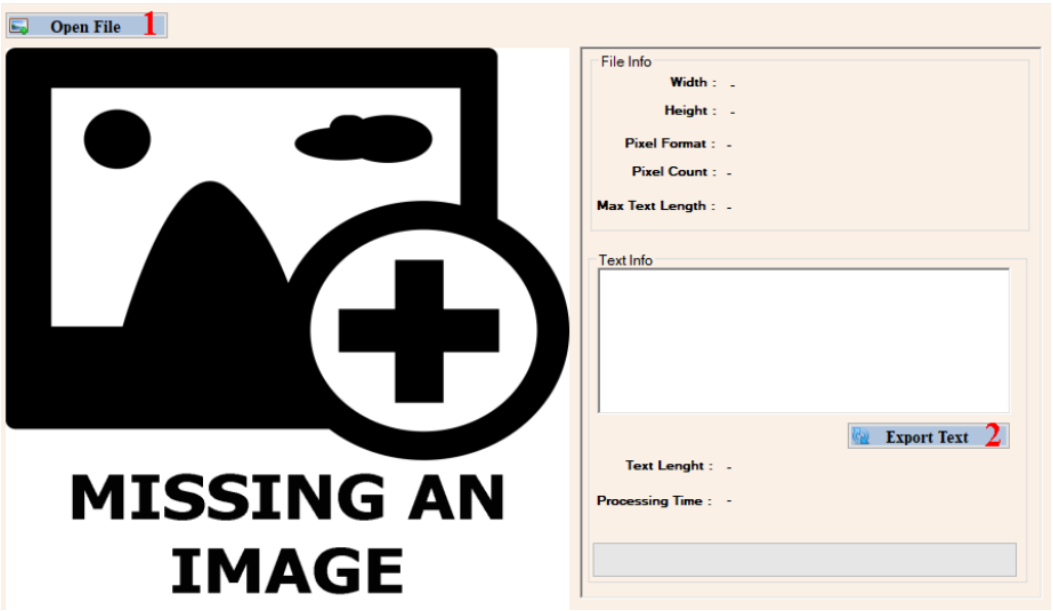

Рисунок 6.2 – процесс получения сообщения из изображения

Тут всё предельно просто – выбираем сообщение с зашифрованным сообщением и получаем его.

7. Заключение

В ходе выполнения курсовой работы разработано приложение, которое позволяет зашифровать сообщение алгоритмом Эль-Гамаля на основе эллиптических кривых, вставить зашифрованное сообщение в изображение, получить зашифрованный текст из изображения и расшифровать его.

Интерфейс приложения не вызывает трудностей в понимании, нет никаких лишних, “запутывающих” кнопок или панелей.

8. Список используемых источников

1) https://ru.wikipedia.org/wiki/%D0%AD%D0%BB%D0%BB%D0%B8%D0%BF%D1%82%D0%B8%D1%87%D0%B5%D1%81%D0%BA%D0%B0%D1%8F_%D0%BA%D1%80%D0%B8%D0%BF%D1%82%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F
2) https://ru.wikipedia.org/wiki/%D0%A1%D1%82%D0%B5%D0%B3%D0%B0%D0%BD%D0%BE%D0%B3%D1%80%D0%B0%D1%84%D0%B8%D1%8F
3) https://eax.me/elliptic-curves-crypto/
4) https://www.sites.google.com/site/anisimovkhv/learning/kripto/lecture/tema8
5) https://habr.com/ru/post/253045/

9. Приложение

EllipticCurverEncryption.cs

```
class EllipticCurvesEncryption
   {
      static int p = 223; // модуль эллиптической кривой, часть открытого ключа, обязано быть простым числом
      static int d; // закрытый ключ
      static int[] P, Q, D;
      static int[] arrAB;
      public static string EncryptString(string stringToEncrypt)
      {
         string encryptedStr = "";
         // G из Википедии = P из того сайта конченого (P это xy начальные)
         // Ep из Википедии = AB с того сайта конченого
         //int d = 10; // то, на сколько умножать координаты; 1 < d < q - 1
         arrAB = GetAB(p); // часть открытого ключа
         Random rand = new Random();
         int x1 = 0, y1 = 0;
         int q; // порядок циклической подгруппы группы точек эллиптич. кривой
         do
         {
            x1 = 0;
            y1 = 0;
            while (true)
            {
               if (Math.Pow(y1, 2) % p == (Math.Pow(x1, 3) + arrAB[0] * x1 + arrAB[1]) % p) // уравнение кривой
                  break;
               x1 = rand.Next(1, p - 1);
               y1 = rand.Next(1, p - 1);
            }
            P = new int[] { x1, y1 }; // часть открытого ключа
            q = Get_q(p, arrAB, x1, y1);
         } while (!isSimple(q));
         d = rand.Next(1, q - 1); // получаем рандомный закрытый ключ
         Q = NumMultiplyByCoord(p, arrAB[0], P, d); // часть открытого ключа

         // Encryption
         //int letter_id = 12;
         //encryptedStr = $"Pk = {Pk[0]}; {Pk[1]}; Qk = {Qk[0]}; {Qk[1]}; c = {c}; string to encrypt = {stringToEncrypt}";
         foreach(char c in stringToEncrypt)
```

```
            {
                encryptedStr += EncryptLetter(c, rand, arrAB[0], P, Q);
            }

            return encryptedStr;
        }

        static string EncryptLetter(char letterToEncrypt, Random rand, int A, int[] P,
int[] Q)
        {
            int letterToEncryptASCII = letterToEncrypt;
            string encryptedLetter = "";
            int rand_k = rand.Next(1, p - 1);
            //rand_k = 5;
            int[] Pk = NumMultiplyByCoord(p, A, P, rand_k);
            //Pk = new int[] { 25, 2 };
            int[] Qk = NumMultiplyByCoord(p, A, Q, rand_k);
            //Qk = new int[] { 3, 24 };
            double c = (letterToEncryptASCII * Qk[0]) % p;
            encryptedLetter = $"{Pk[0]};{Pk[1]};{c};";

            return encryptedLetter;
        }

        public static string DecryptString(string encryptedStr)
        {
            string decryptedStr = "";
            //int[] numbers = encryptedStr.Split(';').Select(snum =>
int.Parse(snum)).ToArray();
            int[] numbers = encryptedStr.Split(";".ToCharArray(),
StringSplitOptions.RemoveEmptyEntries)
                .Select(x => Convert.ToInt32(x)).ToArray();
            for(int i = 0; i < numbers.Count(); i += 3)
            {
                decryptedStr += DecryptLetter(numbers[i], numbers[i+1], numbers[i+2],
arrAB[0]);
            }

            return decryptedStr;
        }

        static char DecryptLetter(int Pk0, int Pk1, int c, int A)
        {
            D = NumMultiplyByCoord(p, A, new int[]{ Pk0, Pk1 }, d);
```

```
            int xdminusone = 1;
            while ((D[0] * xdminusone) % p != 1)
            {
                xdminusone++;
            }
            //xdminusone = 14;
            double letterASCII = (c * xdminusone) % p;
            return (char)letterASCII;
        }

        static int[] GetAB(int p)
        {
            int[] result = new int[2];
            Random rand = new Random();
            result[0] = rand.Next(1, p - 1); // = A
            result[1] = rand.Next(1, p - 1); // = B

            while ((4 * Math.Pow(result[0], 3) + 27 * Math.Pow(result[1], 2)) % p ==
0) // проверка чтобы A и B были такими что (4A^3 + 27B^2)mod(p) != 0
            {
                result[0] = rand.Next(1, p - 1);
                result[1] = rand.Next(1, p - 1);
            }

            return result;
        }

        static int Get_q(int p, int[] arrAB, int x1, int y1)
        {
            int x2 = 0, y2 = 0, delta = 0, counter = 2;
            do
                delta++;
            while
                ((2 * y1 * delta) % p != (3 * Math.Pow(x1, 2) + arrAB[0]) % p);

            int tempx = (int)Math.Pow(delta, 2) - 2 * x1;
            while (tempx < p)
                tempx += p;
            x2 = tempx % p;
            int tempy = delta * (x1 - x2) - y1;
            while (tempy < p)
                tempy += p;
            y2 = tempy % p;
            int[] arrPrevXY = { x2, y2 };
```

```
            while (arrPrevXY[0] - x1 != 0)
            {
                arrPrevXY = GetXY(p, arrPrevXY, x1, y1);
                counter++;
            }

            counter++;
            return counter;
        }

        static int[] GetXY(int p, int[] arrPrevXY, int x1, int y1) // counts x and y after x2 and y2 (e.g. x3, x4, x5, ...)
        {
            int[] arrXY = new int[2];
            int delta_temp_x = arrPrevXY[0] - x1;
            while (delta_temp_x < p)
                delta_temp_x += p;
            int delta_temp_y = arrPrevXY[1] - y1;
            while (delta_temp_y < p)
                delta_temp_y += p;
            int delta = 0;
            while ((delta * delta_temp_x) % p != delta_temp_y % p)
                delta++;

            int arrX_temp = (int)(Math.Pow(delta, 2) - x1 - arrPrevXY[0]);
            while (arrX_temp < p)
                arrX_temp += p;
            arrXY[0] = arrX_temp % p;
            int arrY_temp = delta * (x1 - arrXY[0]) - y1;
            while (arrY_temp < p)
                arrY_temp += p;
            arrXY[1] = arrY_temp % p;

            return arrXY;
        }

        static int[] NumMultiplyByCoord(int p, int A, int[] coordArr, int n) // p - то, что стоит после модуля; A - начальное A; coordArr - xy координаты; n - на какое число умножать координаты
        {
            int[] newXYarr = new int[2];
            int delta = 1;
            delta = GetDelta(coordArr, coordArr, A, p);
```

```
            newXYarr = GetNewXAndYUsingDelta(coordArr, coordArr, delta, p);

            if (n > 2)
            {
                for (int i = 2; i < n; i++)
                {
                    delta = GetDelta(newXYarr, coordArr, A, p);
                    newXYarr = GetNewXAndYUsingDelta(newXYarr, coordArr, delta,
p);
                }
            }

            return newXYarr;
        }

        static int GetDelta(int[] arrCurr, int[] arrStart, int A, int p)
        {
            int result = 1;

            if (arrCurr[0] == arrStart[0] && arrCurr[1] == arrCurr[1])
            {
                result = ModFromDiv((3 * (int)Math.Pow(arrCurr[0], 2) + A), (2 *
arrCurr[1]), p);
            }
            else
            {
                result = ModFromDiv((arrCurr[1] - arrStart[1]), (arrCurr[0] -
arrStart[0]), p);
            }

            return result;
        }

        static int ModFromDiv(int x, int y, int z) // (x/y)mod(z)
        {
            while (x < 0 || y < 0) // проверка на отрицательные значения в x-се или
y-ке
            {
                x += z;
                y += z;
            }

            int d = 1, e = 0, f, g = z, h, i, j = y;
            for (i = 0; j > 0; i++)
```

```
        {
            try
            { h = y / g; }
            catch
            { h = 0; }

            j = y - h * g;
            if (i != 0)
            {
                f = d * h + e;
                e = d;
                d = f;
            }
            y = g;
            g = j;
        }

        if (i % 2 != 0)
            e = z - e;
        try
        { d = (x * e) % z; }
        catch
        { d = (x * e); }
        return d;
    }

    static int[] GetNewXAndYUsingDelta(int[] arrCurr, int[] arrStart, int delta, int
p)
    {
        int[] result = new int[2];
        int newX = (int)Math.Pow(delta, 2) - arrCurr[0] - arrStart[0];
        while (newX < p)
            newX += p;
        result[0] = newX % p;
        int newY = delta * (arrCurr[0] - result[0]) - arrCurr[1];
        while (newY < p)
            newY += p;
        result[1] = newY % p;
        return result;
    }

    static bool isSimple(int number)
    {
        bool prost = true;
```

```
            for(int i = 2; i <= number / 2; i++)
            {
                if(number % i == 0)
                {
                    prost = false;
                    break;
                }
            }

            return prost;
        }
    }
```

Exporter.cs

```
public class Exporter
    {
        private MainForm _form;
        private Helper _helper;
        public Exporter(MainForm form)
        {
            _form = form;
            _helper = new Helper(form);
        }

        /// <summary>
        /// Checks selected file before starting the process
        /// </summary>
        /// <returns></returns>
        private int CheckBeforeExport()
        {
            try
            {
                return ExportTextLength();
            }
            catch
            {
                return 0;
            }
        }

        /// <summary>
        /// Sets selected file info on the form.
        /// </summary>
```

```
        public void SetExportedImageInfo()
        {
            _form.ImageWidth = _form.ExportPictureBoxImage.Width;
            _form.ImageHeight = _form.ExportPictureBoxImage.Height;
            _form.ExportWidthLabelText = _form.ImageWidth.ToString();
            _form.ExportHeightLabelText = _form.ImageHeight.ToString();
            _form.ExportFormatLabelText =
_form.ExportPictureBoxImage.PixelFormat.ToString();
            _form.ExportPixelsLabelText = (_form.ImageWidth *
_form.ImageHeight).ToString();
            _form.ExportMaxLengthLabelText = ((_form.ImageWidth *
(_form.ImageHeight - 1) * 3) / 7) + "  characters.";
        }

        /// <summary>
        /// Exports the text length using the bottom pixels line of the selected image
file
        /// </summary>
        /// <returns>Returns the length of the text</returns>
        private int ExportTextLength()
        {
            var bitmap = new Bitmap(_form.ExportPictureBoxImage);
            string thirtyBytes = string.Empty, textLength = string.Empty;

            for (int i = 0; i < 10; i++)  // gettint last 30 bytes of the image
            {
                thirtyBytes += Convert.ToString(bitmap.GetPixel(i, _form.ImageHeight
- 1).ToArgb(), 2).Substring(8);
            }

            int pointer = 7;
            for (int i = 0; i < 30; i++)   // getting the last bit of the each bytes
            {
                textLength += thirtyBytes.Substring(pointer, 1);
                pointer += 8;
            }

            var textLengthDecimalForm = new char[5];
            int m, tmp = 0, decrease = 0, k = 0;

            for (m = 0; m < textLength.Length / 6; m++)
            {
                for (int n = k; n < k + 6; n++)
                {
```

```
                tmp += Convert.ToInt32(textLength.Substring(n, 1)) *
(int)Math.Pow(2, (5 - decrease));
                decrease++;
            }

            textLengthDecimalForm[m] = Convert.ToChar(tmp);
            k += 6;
            tmp = 0;
            decrease = 0;
        }
        return Convert.ToInt32(new
string(textLengthDecimalForm).TrimStart('0'));
    }

    /// <summary>
    /// Exports the text from the selected image file by starting from top left
corner.
    /// </summary>
    public void Export()
    {
        _form.ExportTextBoxText = string.Empty;
        _form.Sw = new Stopwatch();
        _form.Sw.Start();
        _form.ExportProgressBarValue = 0;

        var importedTextLength = CheckBeforeExport();
        if (importedTextLength != 0)
        {
            _form.Bmp = new Bitmap(_form.ExportPictureBoxImage);
            int totalBytes = importedTextLength * 7;
            int totalPixels = totalBytes / 3;
            int totalBytesMod3 = totalBytes % 3;
            if (totalBytesMod3 != 0)
                totalPixels++;
            _form.ExportProgressBarMaximum = importedTextLength + totalPixels;

            var bytesToExport =
_helper.GetOnlyNecessaryBytesFromImage(totalPixels, totalBytesMod3,
_form.ExpProgressBar);
            int pointer = 6; //++++
            string bytesToExportLast = string.Empty;
            string importedText = null;
```

```
            for (int l = 0; l < totalBytes; l++) // Getting the last bit of each bytes and
stores to 'bytesToExportast'
            {
                bytesToExportLast += bytesToExport.Substring(pointer, 2);
                pointer += 8;
            }
            int decrease = 0, k = 0, temp = 0;
            importedText = string.Empty;

            for (int j = 0; j < bytesToExportLast.Length / 14; j++)
            {
                for (int i = k; i < k + 14; i++)
                {
                    temp += Convert.ToInt32(bytesToExportLast.Substring(i, 1)) *
(int)Math.Pow(2, (13 - decrease));
                    decrease++;
                }

                if (temp >= 1000)
                {
                    importedText +=
Encoding.Unicode.GetString(BitConverter.GetBytes(temp +
43032)).TrimEnd((Char)0);
                }
                else if (temp < 13)
                {
                    importedText += _helper.NumberToTurkishChar(temp);
                }
                else
                {
                    importedText +=
Encoding.Unicode.GetString(BitConverter.GetBytes(temp)).TrimEnd((Char)0);
                }

                _form.ExpProgressBar.Increment(1);
                k += 14; temp = decrease = 0;
            }

            //_form.ExportTextBoxText = importedText;
            _form.ExportTextBoxText =
EllipticCurvesEncryption.DecryptString(importedText);
            SetInfoLabels();
        }
        else
```

```
            {
                MessageBox.Show("This is not a stego file!",
CommonConstants.WarningCaption, MessageBoxButtons.OK,
MessageBoxIcon.Warning);
            }
        }

        /// <summary>
        /// Set process result info on the screen.
        /// </summary>
        private void SetInfoLabels()
        {
            _form.Sw.Stop();
            _form.ExportedTextLengtLabelText =
_form.ExportTextBoxText.Length.ToString();
            _form.ExportDurationLabelText = string.Format("{0} mins. {1} secs. {2}
msec.", _form.Sw.Elapsed.Minutes, _form.Sw.Elapsed.Seconds,
+_form.Sw.Elapsed.Milliseconds);
            _form.ExportTickPictureBoxVisible = true;
        }
    }
```

Improrter.cs

```
public class Importer
    {
        public static int Korean_id = 0;
        private MainForm _form;
        private Helper _helper;
        public int Korea_First_Index = 44032;
        public string cryptText;
        public Importer(MainForm form)
        {
            _form = form;
            _helper = new Helper(_form);
        }

        /// <summary>
        /// Checks selected file before starting the process
        /// </summary>
        /// <returns></returns>
        private bool CheckBeforeImport()
        {
            if (_form.ImageHeight == 0)
```

```
            {
                MessageBox.Show("Open an image file!",
CommonConstants.WarningCaption, MessageBoxButtons.OK,
MessageBoxIcon.Warning);
                return false;
            }
            if (((_form.ImageWidth * (_form.ImageHeight - 1) * 3) / 7) <
cryptText.Length)
            {
                MessageBox.Show("Too much text for the selected image!",
CommonConstants.WarningCaption, MessageBoxButtons.OK,
MessageBoxIcon.Warning);
                return false;
            }
            return true;
        }

        /// <summary>
        /// Sets selected file info on the form.
        /// </summary>
        public void SetImportedImageInfo()
        {
            _form.ImageWidth = _form.ImportPictureBoxImage.Width;
            _form.ImageHeight = _form.ImportPictureBoxImage.Height;
            _form.ImportWidthLabelText = _form.ImageWidth.ToString();
            _form.ImportHeightLabelText = _form.ImageHeight.ToString();
            _form.ImportFormatLabelText =
_form.ImportPictureBoxImage.PixelFormat.ToString();
            _form.ImportPixelsLabelText = (_form.ImageWidth *
_form.ImageHeight).ToString();
            _form.ImportMaxLengthLabelText = ((_form.ImageWidth *
(_form.ImageHeight - 1) * 3) / 7) + "  characters.";
        }

        /// <summary>
        /// Imports the text length using the bottom pixels line of the selected image
file.
        /// </summary>
        private void ImportTextLength()
        {
            var thirtyBytes = string.Empty;
            int counter = 0, y = 0;
            var newBottom = string.Empty;
            var textLength = cryptText.Length.ToString().PadLeft(5, '0');
```

```
            var textLengthDecimalArray = textLength.ToCharArray(); // 0 -> 0
            var textLengthBinary = textLengthDecimalArray.Aggregate(string.Empty,
(current, t) => current + Convert.ToString(t, 2)); // 0 -> 110000

            for (var i = 0; i < 10; i++)
                thirtyBytes += Convert.ToString(_form.Bmp.GetPixel(i,
_form.ImageHeight - 1).ToArgb(), 2).Substring(8);

            while (30 > counter) // (max. 99999 chars means max 30 bytes
            {
                newBottom = newBottom + thirtyBytes.Substring(y, 7) +
int.Parse(textLengthBinary.Substring(counter, 1));
                counter++;
                y += 8;
            }

            y = 0;
            var arrayToImport = _helper.StringToByteArray(newBottom);
            for (var i = 0; i < 10; i++)
            {
                _form.Bmp.SetPixel(i, _form.ImageHeight - 1,
Color.FromArgb(arrayToImport[y], arrayToImport[y + 1], arrayToImport[y + 2]));
                y += 3;
            }
        }

        /// <summary>
        /// Imports the text into the selected image file by starting from top left corner.
        /// </summary>
        public void Import()
        {
            //cryptText = EllipticCurve.CryptInfo(_form.ImportTextBoxText);
            cryptText =
EllipticCurvesEncryption.EncryptString(_form.ImportTextBoxText);
            _form.Sw = new Stopwatch();
            _form.Sw.Start();
            _form.ImportProgressBarValue = 0;

            if (CheckBeforeImport())
            {
                string bitsToImport = string.Empty;
                var charsToImport = cryptText.ToCharArray();
                var totalBytes = charsToImport.Length * 7;
                var totalPixels = totalBytes / 3; // Total number of pixels to be used..
```

```
            var totalBytesMod3 = totalBytes % 3;
            if (totalBytesMod3 != 0)
                totalPixels++;
            _form.ImportProgressBarMaximum = totalPixels + totalBytes;

            var charToBits = string.Empty;

            foreach (var chr in charsToImport)// bitsToImport holds the bit form of
the character that will be imported.
            {
                if (chr >= Korea_First_Index)
                {
                    charToBits = Convert.ToString(chr - Korea_First_Index + 1000, 2);
                    //(Chr - Korean_First_Index is to reduce 16 bits to 14 bits) (The
reason that add 1000 to text is to protect overlapping with turkish).
                }
                else
                {
                    charToBits = Convert.ToString(chr, 2); //if chr is not Korean,
change to binary
                }

                if (charToBits.Length > 7 && charToBits.Length < 9) // if it's binary
is between 7 and 9
                {
                    bitsToImport += _helper.TurkishCharTo7Bit(chr); // set to 14 bits
in Turkish letter
                }
                else
                {
                    bitsToImport += charToBits.PadLeft(14, '0'); // If less than 14 bits,
padding the leading 0

                }
            }

            var imageBits = _helper.GetOnlyNecessaryBytesFromImage(totalPixels,
totalBytesMod3, _form.ImpProgressBar);
            int sevenBitPointer = 0, oneBitPointer = 0;
            var imageBitsLast = string.Empty;

            while (bitsToImport.Length > oneBitPointer) //bitsToImport is importing
into imagebits. the result is stored in imageBitLast.
            {
```

```
                imageBitsLast += imageBits.Substring(sevenBitPointer, 6) +
(bitsToImport.Substring(oneBitPointer, 2));
                oneBitPointer = oneBitPointer + 2;
                sevenBitPointer += 8;
                _form.ImpProgressBar.Increment(1);
            }

            ImportTextLength();
            imageBitsLast += imageBits.Substring(sevenBitPointer);
            _form.ImportPictureBoxImage = _form.Bmp =
_helper.ByteArrayToBitmap(_helper.StringToByteArray(imageBitsLast));
            SetInfoLabels();
        }
    }

    /// <summary>
    /// Set process result info on the screen.
    /// </summary>
    private void SetInfoLabels()
    {
        _form.Sw.Stop();
        _form.ImportedTextLengtLabelText = cryptText.Length.ToString();
        _form.ImportDurationLabelText = string.Format("{0} mins. {1} secs. {2}
msec.", _form.Sw.Elapsed.Minutes, _form.Sw.Elapsed.Seconds,
+_form.Sw.Elapsed.Milliseconds);
        _form.ImportTickPictureBoxVisible = true;
    }
}
```

MainForm.cs

```
public partial class MainForm : Form
{
    #region Fields
    private readonly Importer _importer;
    private readonly Exporter _exporter;
    #endregion

    #region Properties

    public Bitmap Bmp;
    public Stopwatch Sw;
    public int ImageWidth, ImageHeight;

    public ProgressBar ImpProgressBar
```

```
{
    get { return ImportProgressBar; }
}
public int ImportProgressBarValue
{
    set { ImportProgressBar.Value = value; }
}
public int ImportProgressBarMaximum
{
    set { ImportProgressBar.Maximum = value; }
}

public ProgressBar ExpProgressBar
{
    get { return ExportProgressBar; }
}
public int ExportProgressBarValue
{
    set { ExportProgressBar.Value = value; }
}
public int ExportProgressBarMaximum
{
    set { ExportProgressBar.Maximum = value; }
}

public Image ImportPictureBoxImage
{
    get { return ImportPictureBox.Image; }
    set { ImportPictureBox.Image = value; }
}
public Image ExportPictureBoxImage
{
    get { return ExportPictureBox.Image; }
    set { ExportPictureBox.Image = value; }
}

public string ImportTextBoxText
{
    get { return ImportTextBox.Text; }
    set { ImportTextBox.Text = value; }
}
public string ImportWidthLabelText
{
    get { return ImportWidthLabel.Text; }
```

```
    set { ImportWidthLabel.Text = value; }
}
public string ImportHeightLabelText
{
    get { return ImportHeightLabel.Text; }
    set { ImportHeightLabel.Text = value; }
}
public string ImportFormatLabelText
{
    get { return ImportFormatLabel.Text; }
    set { ImportFormatLabel.Text = value; }
}
public string ImportPixelsLabelText
{
    get { return ImportPixelsLabel.Text; }
    set { ImportPixelsLabel.Text = value; }
}
public string ImportMaxLengthLabelText
{
    get { return ImportMaxLengthLabel.Text; }
    set { ImportMaxLengthLabel.Text = value; }
}
public string ImportedTextLengtLabelText
{
    get { return ImportedTextLengtLabel.Text; }
    set { ImportedTextLengtLabel.Text = value; }
}
public string ImportDurationLabelText
{
    get { return ImportDurationLabel.Text; }
    set { ImportDurationLabel.Text = value; }
}

public string ExportTextBoxText
{
    get { return ExportTextBox.Text; }
    set { ExportTextBox.Text = value; }
}
public string ExportWidthLabelText
{
    get { return ExportWidthLabel.Text; }
    set { ExportWidthLabel.Text = value; }
}
public string ExportHeightLabelText
```

```
{
    get { return ExportHeightLabel.Text; }
    set { ExportHeightLabel.Text = value; }
}
public string ExportFormatLabelText
{
    get { return ExportFormatLabel.Text; }
    set { ExportFormatLabel.Text = value; }
}
public string ExportPixelsLabelText
{
    get { return ExportPixelsLabel.Text; }
    set { ExportPixelsLabel.Text = value; }
}
public string ExportMaxLengthLabelText
{
    get { return ExportMaxLengthLabel.Text; }
    set { ExportMaxLengthLabel.Text = value; }
}
public string ExportedTextLengtLabelText
{
    get { return ExportedTextLengtLabel.Text; }
    set { ExportedTextLengtLabel.Text = value; }
}
public string ExportDurationLabelText
{
    get { return ExportDurationLabel.Text; }
    set { ExportDurationLabel.Text = value; }
}

public bool ImportTickPictureBoxVisible
{
    set { ImportTickPictureBox.Visible = value; }
}
public bool ExportTickPictureBoxVisible
{
    set { ExportTickPictureBox.Visible = value; }
}
#endregion

#region Ctor

public MainForm()
{
```

```
            InitializeComponent();
            _exporter = new Exporter(this);
            _importer = new Importer(this);
        }

        #endregion

        #region Button Click Events
        private void OpenFileImportButton_Click(object sender, EventArgs e)
        {
            OpenFileDialog.Filter = CommonConstants.OpenFileFilter;
            if (OpenFileDialog.ShowDialog() == DialogResult.OK)
            {
                ImportPictureBox.Image = Image.FromFile(OpenFileDialog.FileName);
                Bmp = new Bitmap(ImportPictureBox.Image);
                _importer.SetImportedImageInfo();
            }
        }
        private void OpenFileExportButton_Click(object sender, EventArgs e)
        {
            OpenFileDialog.Filter = CommonConstants.OpenFileFilter;
            if (OpenFileDialog.ShowDialog() == DialogResult.OK)
            {
                ExportPictureBox.Image = Image.FromFile(OpenFileDialog.FileName);
                _exporter.SetExportedImageInfo();
            }
        }
        private void ImportButton_Click(object sender, EventArgs e)
        {
            if (ImportTextBox.TextLength > 0)
            {
                _importer.Import();
            }
            else
            {
                MessageBox.Show("Enter text before!",
CommonConstants.WarningCaption, MessageBoxButtons.OK,
MessageBoxIcon.Warning);
            }
        }
        private void ExportButton_Click(object sender, EventArgs e)
        {
            if (ExportWidthLabel.Text != "-")
            {
```

```
            _exporter.Export();
        }
        else
        {
            MessageBox.Show("Open an image file before!",
CommonConstants.WarningCaption, MessageBoxButtons.OK,
MessageBoxIcon.Warning);
        }
    }

    private void Help_Click(object sender, EventArgs e)
    {

    }

    private void ExportPictureBox_Click(object sender, EventArgs e)
    {

    }

    private void textBox2_TextChanged(object sender, EventArgs e)
    {

    }

    private void label6_Click(object sender, EventArgs e)
    {

    }

    private void MainForm_Load(object sender, EventArgs e)
    {

    }

    private void SaveButton_Click(object sender, EventArgs e)
    {
        SaveFileDialog saveFileDialog = new SaveFileDialog();
        saveFileDialog.Filter = CommonConstants.SaveDialogFilter;
        saveFileDialog.Title = CommonConstants.SaveDialogTitle;
        saveFileDialog.ShowDialog();

        if (saveFileDialog.FileName != string.Empty)
        {
```

```
            switch (saveFileDialog.FilterIndex)
            {
                case 1:
                    Bmp.Save(saveFileDialog.FileName);
                    break;
                case 2:
                    Bmp.Save(saveFileDialog.FileName);
                    break;
                case 3:
                    System.IO.FileStream fs2 =
(System.IO.FileStream)saveFileDialog.OpenFile();
                    Bmp.Save(fs2, System.Drawing.Imaging.ImageFormat.Bmp);
                    fs2.Close();
                    break;
            }
        }
    }
    #endregion
}
```