
Contents

Analysis	2
Project Background	2
Project Outline	2
Research	3
Steganography Method	3
Software	5
Constraints	5
Objectives	6
Design	7
Specific Requirements	7
Encryption Algorithm	7
Decryption Algorithm	7
Metadata	7
Inheritance	8
High Level Design	9
Hierarchy Diagram	9
Class Diagram	10
Data Flow Diagram	11
Event Diagram	12
Class Explanations	13
GUI Design	15
Technical Solution	16
Challenges Faced	16
Data.cs	21
SourceImage.cs	21
EncryptedImage.cs	22
HiddenData.cs	23
Testing	28
Test Plan	28
Testing	29
Evaluation	31
Fulfilment of Objectives	31
How Program Improved	32
Feedback	32
Areas for Development	32
Appendices	33
Testing Carried out During Development	33
Source Code	38
Evidence of Program Working	48

Analysis

Project Background

When deciding which project to undertake I was deliberating between 3 options:

- A maze-solving Arduino robot.
- An IP camera with real time motion detection and tracking.
- Software to conceal data within images using steganography.

Rejected Choice – Arduino Robot

The idea of a robot solving mazes sounds impressive, but, after some research, I found that the actual code required wouldn't be particularly interesting as it would primarily be hardcoded instructions and algorithms. While the algorithms could provide some good research opportunities, they would still just be shortest path algorithms copied from the internet. As well as this, the physical construction of a robot would be required for this project but would not yield any extra credit. For these reasons, I decided against choosing this project.

Rejected Choice – IP Camera

I started to make this project, but, due to the time constraints; the software I was using; and the lack of a clear end goal, it was an infeasible project. While doing some very early testing I noticed that integrating a live video feed was potentially an entire project in itself and with that being just part of what I would be trying to achieve, I decided it best to just start from scratch with a more promising idea.

Steganography Project

I eventually decided on creating a tool to conceal data within images using steganography. The reason that I chose this project is that it is flexible in terms of its objectives i.e. they are more attainable and can be more easily tailored to fit within the constraints that I am working with. As well as this, cryptography is fascinating to me and steganography is a sort of parallel to cryptography so I thought this project could be an interesting learning experience.

Project Outline

Steganography is defined as "the practice of concealing messages or information within other non-secret text or data." (i.e. hiding in plain sight). Steganography can be used as a method of hiding data or transferring sensitive data without revealing that the data being transferred is sensitive. Messages can be encrypted in all forms of data but for this project I will create a program that can encrypt and decrypt data to and from a digital image with the image remaining seemingly unchanged.

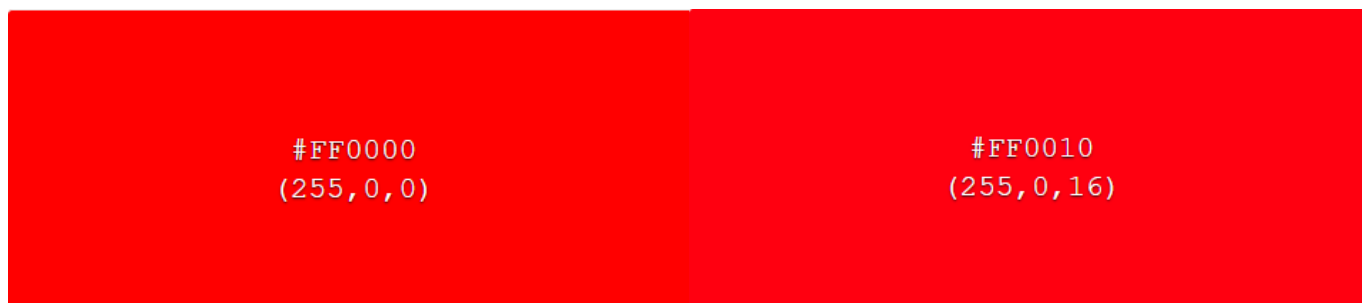
**Disclaimer* For all intents and purposes, in this project "encryption" is synonymous with "steganography" because there are no verbs for "steganography".*

Research

Steganography differs from cryptography in that the aim of cryptography is to encode a message or data in a way that only the intended recipient can decode. Steganography, however, doesn't strictly require the encryption of data but rather the concealing of data within something else so that only the recipient knows that there is more hidden data within what they have received. An example of this would be a standard letter with a secret message written in invisible ink. Because only the intended recipient knows that there is an invisible in message, to any onlooker it would appear to be just a normal letter.

Concealing messages in images requires slight manipulation of the pixels in the image so that the data in the image is different, but, to the naked eye, the image looks exactly the same.

For example:



While there are 16 different shades between these two colours above, they look practically identical to the naked eye. Using this premise, I will create a program that alters images slightly to conceal data within them.

Steganography Method

The method I will implement is the changing of the least significant bit of each pixel. Most standard RGB graphics systems have a colour depth of 24 bits (1 Byte per colour, 3 colours per pixel):

Colours	R		G		B	
Hex Value	F	F	F	F	F	F
Decimal Value	16	16	16	16	16	16
Binary Value	1111	1111	1111	1111	1111	1111
Bits	4 bits	4 bits	4 bits	4 bits	4 bits	4 bits
Bytes	1 Byte		1 Byte		1 Byte	

Because there are 16777216 (2^{24}) colour combinations, by only changing the least significant bit, each pixel will only undergo a $\frac{1}{16777216}$ or 0.00000596% change which is entirely undetectable by even the most trained eyes.

The way that this system works is by passing the data to be hidden into the system as a bitstream and adding this bitstream to the image – one bit per pixel.

Example:

1	2	3	4
5	6	7	8

This 2X4 image is read by the computer as a 1-dimensional array of pixels going left to right and top to bottom. The array of this image is [FFFF00, FFFF00, FFFF00, FFFF00, FFFF00, FFFF00, FFFF00, FFFF00]. To encode the letter 'A' into this array, the letter is first converted to a bitstream (in this case just its ASCII value: 01000001). The bitstream is then added to the image in order, one bit per pixel. The resultant array with the letter A encoded is [FFFF00, FFFF01, FFFF00, FFFF00, FFFF00, FFFF00, FFFF00, FFFF01].

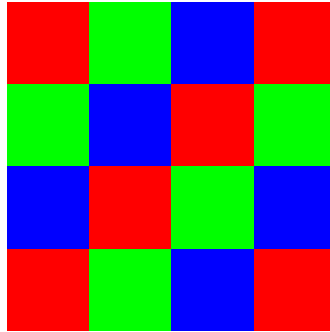
To get the data back out from an image, the altered image is compared with the original, and the output of the comparison is returned as a bitstream which is then parsed, splitting the bitstream into an array of bytes which is then converted back to the desired file type (.txt in this case).

Using this 8-bit parsing method, any file type will be able to be hidden and extracted from an image – as long as the file can be converted to a bitstream and back.

Using this method, a 3 Megapixel image can conceal 375KB of data (about 125 pages of text).

Creating a test Bitmap

In order to test the encryption and decryption algorithms, a very small image is required so that, when tracing, I can ensure that all pixels are being read and manipulated properly. To do this, I created a 4x4 bitmap in Paint with known colour values:



Note - The image appears to blend between colours because of "sampling nearest neighbour" but, in reality, the original image is only 4x4 resolution with a sharp edge between colours.

The array of the image is:

```
[ (255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 0, 0),  
  (0, 255, 0), (0, 0, 255), (255, 0, 0), (0, 255, 0), (0, 0, 255), (255, 0, 0), (0, 255, 0),  
  (0, 0, 255), (255, 0, 0) ]
```

Software

My program will be a C# windows forms application created in Visual Studio. I will use this development environment and language because I am familiar with them as I have been using them over the past 2 years in my computer science lessons. I will make it a windows forms application because I will need a simple GUI to allow easy user interaction to show the change between the source and encrypted images.

C# is a powerful language, so it will allow for relatively quick computation of large data. It will also allow me to write my program in an object-oriented manner.

Constraints

Time

Part of the reason I chose this project was due to the time constraints and the approachability of this project within those constraints. That being said, the time constraints will still somewhat limit the development of my solution:

- My user interface will be functional but basic to allow more time for writing the more in-depth code.

- The program may not be optimised as thoroughly as it could be.

Software

Although C# is a powerful language, writing an application in Windows Forms does come with its limitations:

- The use of my software will be limited to the Windows operating system.
- Another reason the user interface will be basic is down to the decision to use Windows Forms over WPF.

Objectives

1. Program can encrypt a range of **File Types** into a range of **Image File Types** using steganographic techniques.
2. Images holding concealed data are indistinguishable from original images when viewed side by side in high resolution.
3. Program can take a range of **Image File Types** and extract data in a range of **File Types** by comparing it to another image of the same type.
4. The system provides a means for people to share data securely.

File Types	Image file types
.txt	.jpg
.pdf	.png
.jpg	.gif
.gif	.tif
.mp3	

Design

Specific Requirements

Encryption Algorithm

To encrypt the images, I will have a method with 2 parameters: a bitstream (of the message to be concealed within the image) and an image in the form of a bitmap. This method is the algorithm I wrote for hiding data in an image using steganography.

Pseudocode for encrypting:

```
Function Encrypt(bitstream, image)
    i ← 0
    for y ← 0 to image height
        for x ← 0 to image width
            pixel ← (pixel at (x,y))
            pixelValue ← RGB value of pixel
            newPixelValue ← pixelValue XOR bitstream[ i ]
            newPixel ← convert newPixelValue to pixel
            pixel at (x,y) ← newPixel
            i = i + 1
```

Decryption Algorithm

To extract data from the images, I will have a method with 2 parameters: the images that are being compared

Pseudocode for extracting data:

```
Function ExtractData(image1, image2)
    bitstream = ""
    for y ← 0 to image height
        for x ← 0 to image width
            pixel1 ← pixel at img1[x, y]
            pixel2 ← pixel at img2[x, y]
            bitstream = bitstream + (pixel1 XOR pixel2)
```

Metadata

File Type

To be able to conceal multiple types of file within an image, the program will require some way of determining what type of file to save the decrypted bitstream as when an image is decrypted because, after all, a bitstream with no context is useless. To distinguish different file types, I will include some metadata at the start of bitstreams which will also be encrypted in the image. This

just requires a few bits at the start of a bitstream to correspond to a file type. My program will be able to read up to 8 file types meaning the first 3 bits (which can make numbers 0-7 in binary) of an encrypted bitstream will represent the type of file which the bitstream is to be saved as:

Bit Pattern	Corresponding File Type (Example)
000	.txt
001	.pdf
010	.jpg
011	.gif
100	.mp3
101	Available to be used for more file types in later versions of software.
110	
111	

File Size

To give the software an indication of the size of the data it is going to extract from an image, the metadata will also contain information about the size of the bitstream. The metadata containing the size will be 10 bits long and will just be a binary number stating the size of the file (rounded up to the nearest kilobyte).

Example Metadata:

If an encrypted bitstream starts with "0100011100110" the program can tell it is to be saved as JPEG (from the first 3 bits) and that it is between 229Kb and 230Kb in size (the last 10 bits – 0011100110 = 230).

Inheritance

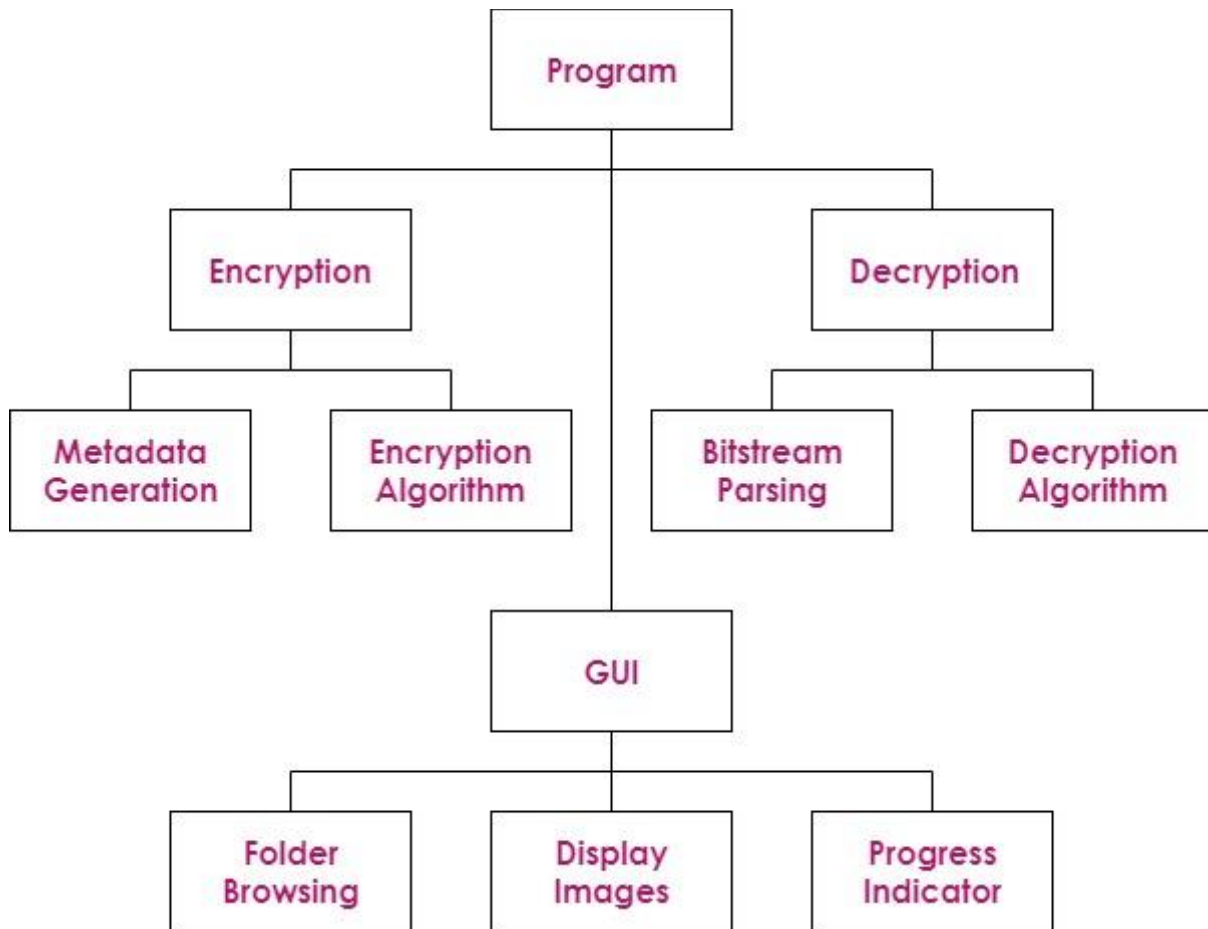
My code will be written using the object-oriented paradigm. I will use objects to represent files and, because different files always have things in common, my program will use inheritance.

In terms of my project: All files have a file name, a file size and a type (e.g. .txt). Because of this, all classes will inherit from an abstract generic data class which contains these attributes.

Inherited classes from this abstract base class will be templates for other categories of file type – e.g. Images.

High Level Design

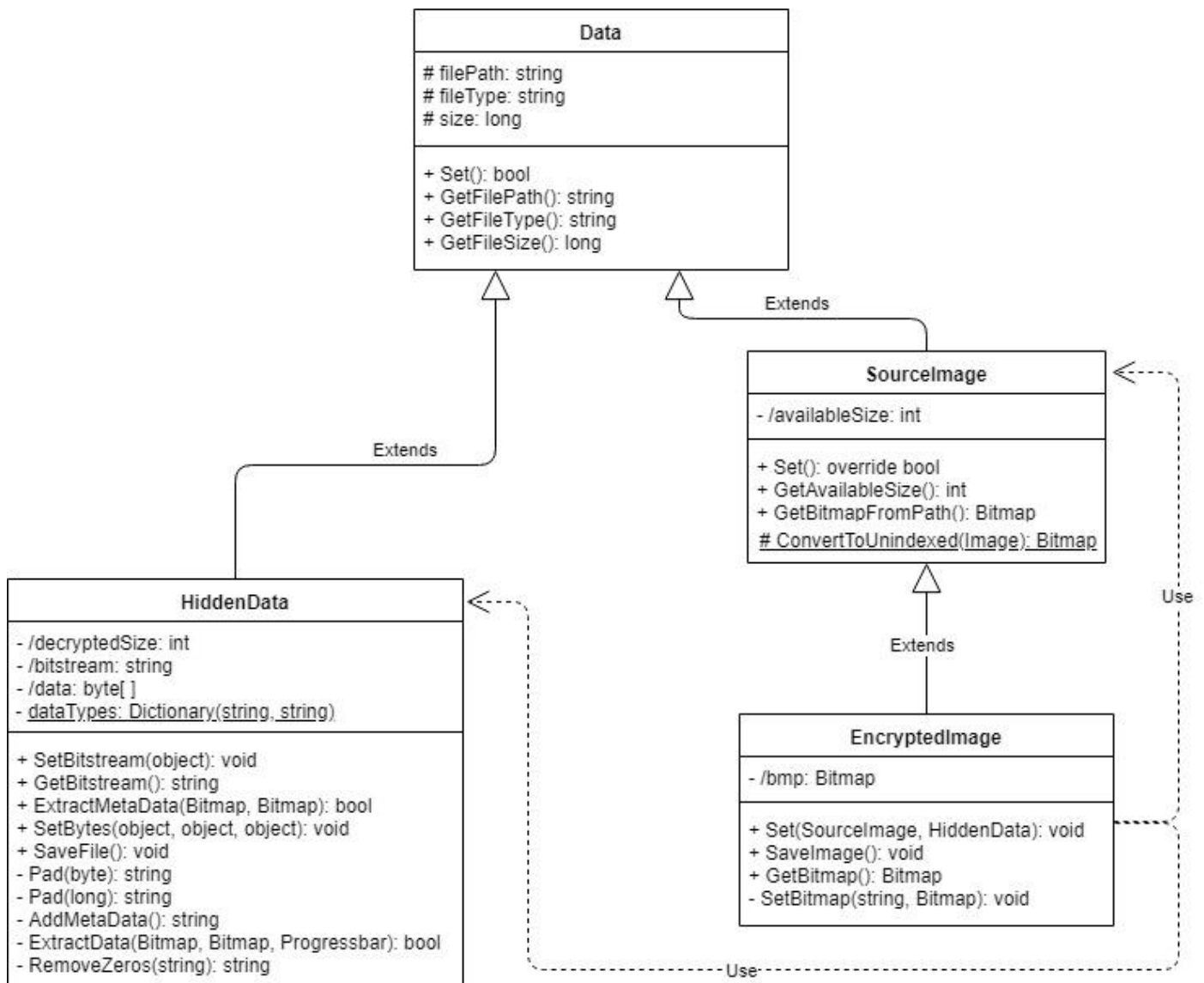
Hierarchy Diagram



The program will essentially have 2 modes:

- “Encrypt Mode” for when the program is creating a new image from a given image and data to be encrypted into that image.
- “Decrypt Mode” for when the program is comparing 2 similar images and extracting data from the difference between them.

Class Diagram

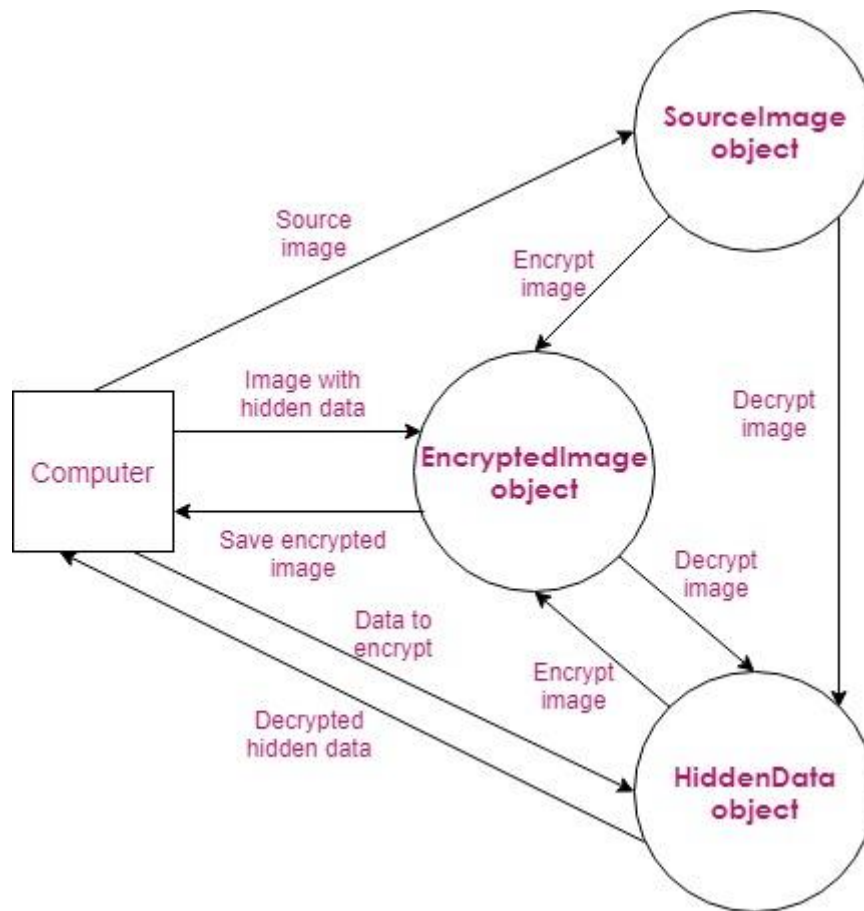


All fields have the private or protected access modifiers on them. This is to simplify when writing and reading code and to protect data from being accidentally modified. To access the properties of an object, all classes will have get and set methods

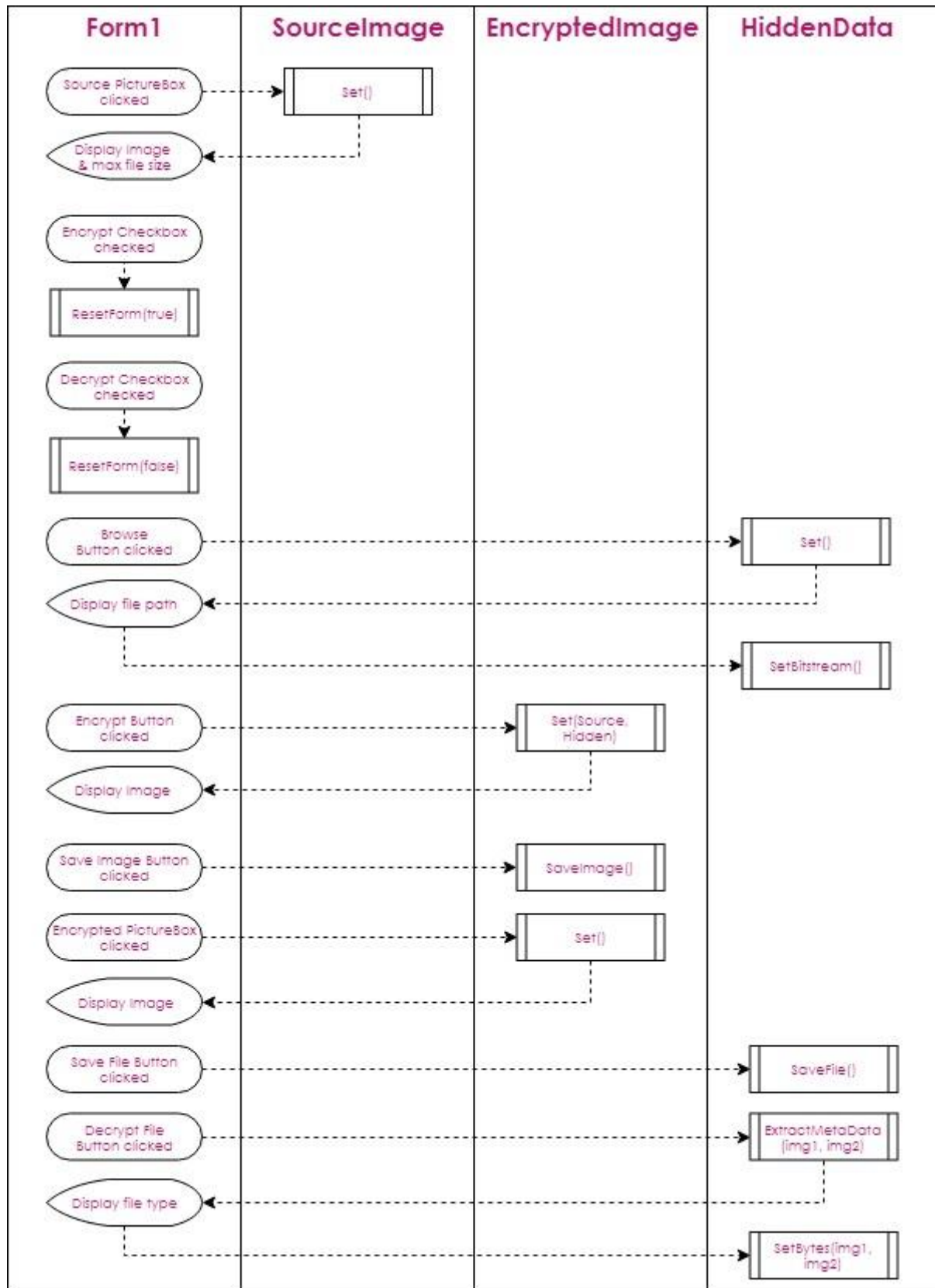
Initially, there was no SourceImage class – EncryptedImage was a derivation of the Data class and the Data class was instantiated to represent image files. The SourceImage class was created to prevent HiddenData objects from inheriting unnecessary attributes like availableSize and GetBitmapFromPath() making Data abstract and SourceImage the class to be instantiated.

PROCESS DIAGRAMS

Data Flow Diagram



Event Diagram



Class Explanations

Data.cs

Data
filePath: string # fileType: string # size: long
+ Set(): bool + GetFilePath(): string + GetFileType(): string + GetFileSize(): long

Data is an abstract class which acts as a generic template for all pieces of data being handled by the program. – All its attributes are applicable to all file types. It is a generalization of the HiddenData class and the SourceImage class.

Field	Purpose
filePath	Stores the file path as a string e.g. "C:\Users\Desktop\image.jpg"
fileType	Stores the type of file as a string e.g. ".jpg"
size	Stores the size of the size of the file (in bytes) e.g. 250402

The fields of this class use the protected access modifier. This is because they need to be inherited by derived classes but the fields should be hidden to everything apart from their respective classes

SourceImage.cs

SourceImage
- /availableSize: int
+ Set(): override bool + GetAvailableSize(): int + GetBitmapFromPath(): Bitmap <u># ConvertToUnindexed(Image): Bitmap</u>

SourceImage inherits from the Data class and is a generalization of the EncryptedImage class. Instances of this class represent image files which are:

- Copied so that the copy can have a file encrypted into it.
- Compared with copies of themselves which have already been encrypted (EncryptedImage objects) in order to extract data that was concealed via steganography.

Field	Purpose
availableSize	Stores the maximum size of file that can be encrypted into the image in bytes. It is calculated by the dimensions of the image / 8 (each pixel stores 1 bit and there are 8 bits in a byte).

EncryptedImage.cs

EncryptedImage
- /bmp: Bitmap
+ Set(SourceImage, HiddenData): void + SaveImage(): void + GetBitmap(): Bitmap - SetBitmap(string, Bitmap): void

EncryptedImage Inherits from the SourceImage class. Instances of this class represent images which:

- Are produced when encrypting a file represented by a HiddenData object into an image represented by a SourceImage object.
- Have already had a file hidden in them and are being compared with an image represented by a SourceImage object to extract data.

Field	Purpose
bmp	Stores the bitmap of the image represented by the EncryptedImage object. Only used when encrypting.

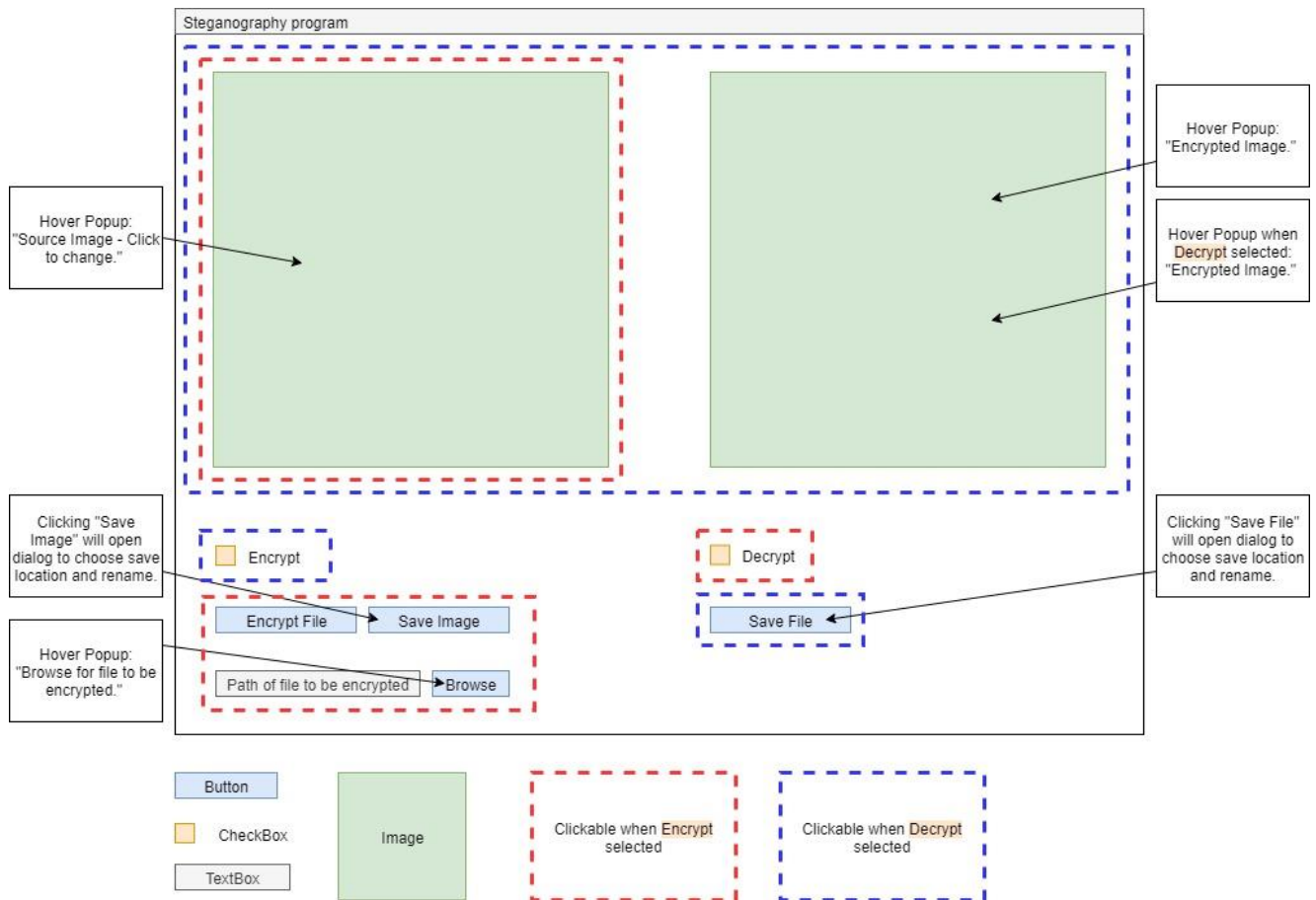
HiddenData.cs

HiddenData
- /decryptedSize: int - /bitstream: string - /data: byte[] - <u>dataTypes: Dictionary(string, string)</u>
+ SetBitstream(object): void + GetBitstream(): string + ExtractMetaData(Bitmap, Bitmap): bool + SetBytes(object, object, object): void + SaveFile(): void - Pad(byte): string - Pad(long): string - AddMetaData(): string - ExtractData(Bitmap, Bitmap, Progressbar): bool - RemoveZeros(string): string

HiddenData handles the data that is to be concealed/has been concealed within the image. It inherits from the Data class because it is a specialized form of data.

Field	Purpose
decryptedSize	Stores the size of the file in kilobytes. It is assigned by extracting the metadata from the bitstream and converting it into an integer.
bitstream	Holds the data represented by the class in the form of a binary string.
data	Holds the data represented by the class in the form of an array of bytes.
dataTypes	A dictionary containing each filetype that is compatible with the program and its corresponding 3-bit value that is used in the metadata.

GUI Design



Technical Solution

Challenges Faced

HIGH LEVEL SOLUTIONS

Storing File Types – Dictionary Storage

Data types and their corresponding binary values were initially going to be stored as an array of tuples. I changed this idea because I needed to be able to look up values using either the binary value or the file type. Another option for storing these together is just having an array of strings (the strings being the file type e.g. ".txt") and writing a method to convert the index to a 3-bit binary value. Due to there only being 8 file types required, the time saved by storing the strings in an ordered array is not significant enough for me to implement it. For this reason, I decided to store the file types in a dictionary with the key and the value both being strings and having them reverse to allow for reverse looking up of keys. E.g.

"000"	".txt"
"001"	".pdf"
"010"	".jpg"
"011"	".gif"
"100"	".mp3"
"101"	".mp4"
"110"	".zip"
"111"	
".txt"	"000"
".pdf"	"001"
".jpg"	"010"
".gif"	"011"
".mp3"	"100"
".mp4"	"101"
".zip"	"110"
	"111"

Implementation

```
static Dictionary<string, string> dataTypes = new Dictionary<string, string>()
{
    { ".txt" , "000" },
    { ".pdf" , "001" },
    { ".jpg" , "010" },
    { ".gif" , "011" },
    { ".mp3" , "100" },
    { ".mp4" , "101" },
    { ".zip" , "110" },
    { "000" , ".txt" },
    { "001" , ".pdf" },
    { "010" , ".jpg" },
    { "011" , ".gif" },
    { "100" , ".mp3" },
    { "101" , ".mp4" },
    { "110" , ".zip" },
};
```

Dictionary holding file types - HiddenData.cs Line 16-32

Adding Pad() Methods – Using Overloading

There was a bug (development tests 10 and 11a) where text files weren't saving properly because each "byte" of data was only 7 bits long (due to ASCII characters only being 7 bits long). When converting from "byte" datatype to "string", the most significant bit was a 0 and therefore removed. The bitstream would be split into chunks of 7 whereas, when decrypting, the program would parse it by splitting it into chunks of 8.

For example: "abc" is 01100001 01100010 01100011 but when adding to a bitstream, the leftmost zeros are removed so the bitstream is 110000111000101100011 which, when reconstructed, becomes 11000011 10001011 00011.

By adding the Pad() method, this is resolved (development test 11b) because "redundant" 0's at the start of the byte are added back so even if, for example, the data was "0100", it would still be saved in the bitstream as "00000100".

I used overloading to create another Pad() method which takes a long argument rather than a byte. Pad(long) is used for the part of the metadata which states the size of a file. It makes sure it is 10 bits long so when parsing after the data after extraction, it can be read properly.

Implementation

```
string Pad(byte b)
{
    string bytestring = Convert.ToString(b, 2);
    if (bytestring.Length < 8)
    {
        while (bytestring.Length != 8)
        {
            bytestring = 0 + bytestring;
        }
    }
    return bytestring;
}

string Pad(long b)
{
    b = 1 + (b / 1000);
    string bytestring = Convert.ToString(b, 2);
    if (bytestring.Length < 10)
    {
        while (bytestring.Length != 10)
        {
            bytestring = 0 + bytestring;
        }
    }
    return bytestring;
}
```

2 Pad() methods using overloading - HiddenData.cs Line 38-63

Adding Multithreading

While testing my program, I noticed that while `HiddenData.SetBitstream()` and `HiddenData.SetBytes` were running, the program would stop responding for a while. The reason for this is that these methods would take a lot of time to execute and cause the rest of the program to stop due to the entire program being on one thread. To solve this problem, and add more computer response, I implemented multithreading. This means that the program runs on a separate thread when certain subroutines are running. I did this by instantiating a `Thread` object at the start of the code and reassigning it with a new `Thread` object when required, passing the method to be executed as an argument.

Implementation

```
Thread alternateThread;
```

Instantiation of a Thread object – Form1.cs Line 15

```
alternateThread = new Thread(hidden.SetBitstream);
alternateThread.Start(progressBar);
```

*Reassigning of Thread object passing the method to be called as a constructor argument
- Form1.cs Line 114-116*

Using Lambda Expression

After implementing multithreading, an issue arose with passing arguments into the Thread constructor. The Thread class is incompatible with methods that have more than one parameter but I had to use it with a method that had 2 parameters. To overcome this, I used a lambda expression which, due to the nature of functional programming, means a method with multiple parameters can be treated as a function that takes no arguments. I could also have created a delegate pointing to the SetBytes() method for this problem and just invoke the Thread passing the delegate as an argument but using a lambda expression was a more elegant solution.

Implementation

```
alternateThread = new Thread(() => hidden.SetBytes(source.GetBitmapFromPath(), encrypted.GetBitmapFromPath(), progressBar));
alternateThread.Start();
```

Using Lambda function to allow a method with multiple parameters to be appear as a method with 1 parameter – Form1.cs Line 152-153

Set() - Overriding

Data.Set() is a generic virtual method for setting fields that are common to all classes in the project. Sourcelmage objects, however, require another field to be assigned when setting the initial values. Because of this specialization, I created Sourcelmage.Set() as an override of Data.set(). It works by calling the base Set() within the new Set() method and then adding functionality by writing more instructions beneath.

Implementation

```
public override bool Set() //Overrides Set() from parent class.
{
    bool value = base.Set(); //Calls original "Set()" from parent class (not recursion) - sets "value" to result.
    try //Exception handling - if "GetBitmapFromPath()" runs then an image was selected by the user.
    {
        availableSize = GetBitmapFromPath().Height * GetBitmapFromPath().Width / 8; //Calculates "availableSize".
        return value; //Returns result of base "Set()" - true if a file was selected, false otherwise.
    }
    catch //If the user selected a non-image file then "GetBitmapFromPath()" would have caused a runtime error.
    {
        MessageBox.Show("Please select an image.", "Error", MessageBoxButtons.OK); //Error message popup.
        return false; //Returns false to show values were not set.
    }
}
```

*Use of overriding to allow extra functionality to be added to method in child class
– Sourcelmage.cs Line 27-40*

Set() - Overloading and Dependency

When the program is in "encrypting mode", the EncryptedImage objects properties are defined by the properties of a Sourcelmage object and a HiddenData object. To set the values of an EncryptedImage object, I wrote a method that overloads the Set() method in the Data class and is dependent on a Sourcelmage object and a HiddenData object. The method takes both objects and invokes a private method within the EncryptedImage class, passing data from the objects as arguments. It also assigns a data to "fileType" (a private field in all Data derivatives) by copying it from the Sourcelmage object that was passed as an argument.

```
public void Set(SourceImage image, HiddenData data)
{
    SetBitmap(data.GetBitstream(), image.GetBitmapFromPath());
    fileType = image.GetFileType();
}
```

*Set() method written as an overload with instances of user defined classes as parameters
– EncryptedImage.cs Line 19-23*

LOWER LEVEL SOLUTIONS

Making ConvertToUnindexed()

When testing the encryption algorithm, an error occurred that returned “SetPixel is not supported for indexed file formats.” (development test 9a). Some images use “Indexed Pixel Format” which is a type of dictionary encoding. It works by defining an array of pixel values to in the metadata at the start of the image then stating each pixel as an index reference rather than an ARGB value – an index to an array requires less data to represent than an ARGB value so storage is saved. Unfortunately, the SetPixel() method is incompatible with images compressed in this way so the ConvertToUnindexed() method was created to generate uncompressed bitmaps from any image to get past this problem (development test 9b).

SourceImage.cs Line 11-19

ResetForm()

When switching between encrypt and decrypt mode, I required a way to quickly refresh the form – essentially making it as if the program had just been opened. To do this I wrote the ResetForm() method which takes a boolean as an argument – true representing encrypt mode and false representing decrypt mode. The method sets all objects to null and disables the use of buttons depending on what mode the program is in.

Form1.cs Line 14-37

Adding RemoveZeros()

There was a bug when saving images that had been encrypted into other images. The decrypted image would be scrambled and have a lot of black pixels at the bottom when saved. This was because the decrypting loop runs for the length of the size of the file in kilobytes meaning there could be up to 1Kb of 0's at the end of the bitstream. These zeros are interpreted as black pixels when saved as an image. RemoveZeros() removes the redundant data from the back of the bitstream by removing groups of 8 zeros at a time.

HiddenData.cs Line 170-187

Splitting Data Class into SourceImage Class

The Data class was initially instantiable and the SourceImage class didn't exist. While writing the class, I realized that a lot of redundant attributes were being inherited by the HiddenData class. To resolve this, I created a new child class (SourceImage) inheriting from Data to handle all the attributes relevant only to images.

Data.cs

Set()

Used for setting all necessary variables in the object using a file dialog. This method is virtual which allows it to be overridden in the Sourcelmage class. The method also uses a temporary instance of the OpenFileDialog class to allow for a window to pop up to provide a GUI to select a file rather than typing in the entire file path.

```
public virtual bool Set()
{
    using (OpenFileDialog dialog = new OpenFileDialog())
    {
        if (dialog.ShowDialog() == DialogResult.OK)
        {
            filePath = dialog.FileName;
            fileType = Path.GetExtension(filePath);
            size = new FileInfo(filePath).Length;
            return true;
        }
        else
        {
            return false;
        }
    }
}

//bool - return value used as validation check to ensure file was selected and values set.
//virtual - allows overriding in the derived class.
//Creates a temporary OpenFileDialog object to browse computers files.
//Shows the dialog box and runs code if choice was selected (i.e. OK button pressed).
//Sets "filePath" to the file path of the file selected in the dialog window.
//Sets "fileType" to the image file type so it can be saved in the same format once encrypted.
//Sets "size" the size of the file in bytes.
//Returns false if "cancel" was selected in dialog.
```

Sourcelmage.cs

Set()

This method overrides the original "Set()" method, from the Data class, to add functionality that is only required for objects representing images (i.e. setting the "availableSize" variable). The method calls the base "Set()" method (Data.Set()) at the start. It then goes on to try to set "availableSize". If a runtime error occurs at this point it could only be because the file that was selected during base.Set() was a non-image file. This is known because the only time an error could occur would be if GetBitmapFromPath() were to try and create a bitmap from a non-image file. If an error does occur the Catch block runs which displays an error message popup and returns false which indicates that the values were not set successfully.

```
public override bool Set()
{
    //Overrides Set() from parent class.
    bool value = base.Set();
    try
    {
        //Calls original "Set()" from parent class (not recursion) - sets "value" to result.
        //Exception handling - if "GetBitmapFromPath()" runs then an image was selected by the user.
        availableSize = GetBitmapFromPath().Height * GetBitmapFromPath().Width / 8;
        return value;
    }
    catch
    {
        //Calculates "availableSize".
        //Returns result of base "Set()" - true if a file was selected, false otherwise.
        //If the user selected a non-image file then "GetBitmapFromPath()" would have caused a runtime error.
        //Error message popup.
        //Returns false to show values were not set.
        MessageBox.Show("Please select an image.", "Error", MessageBoxButtons.OK);
        return false;
    }
}
```

GetBitmapFromPath()

Returns a Bitmap object – created from the file at "filePath". This method was created to quickly return a bitmap given just the file path of the image.

```

public Bitmap GetBitmapFromPath()           //Method returns the unindexed bitmap of the image represented by "Data".
{
    Bitmap bmp = new Bitmap(@filePath);    //Creates a new bitmap from the file stored at "filepath".
    bmp = ConvertToUnindexed(bmp);         //Converts the bitmap to an unindexed version of itself.
    return bmp;                            //Returns newly generated Bitmap.
}

```

ConvertToUnindexed(Image)

Produces an identical but uncompressed bitmap version of the image passed as an argument. This method was created to overcome an issue with compressed images. Images compressed with a dictionary don't have direct values for each pixel but have a reference to a dictionary item which is stored as metadata in the image file. This caused the program to crash when calling the built in GetPixel() method. To prevent this, all images being processed are passed through this method first.

```

protected static Bitmap ConvertToUnindexed(Image img) //This method creates an uncompressed bitmap from an indexed image.
{
    // (Below) Creates new Bitmap object with dimensions of "img" - using 24 bits per pixel (8 bits per colour).
    Bitmap bmp = new Bitmap(img.Width, img.Height, System.Drawing.Imaging.PixelFormat.Format24bppRgb);
    using (Graphics g = Graphics.FromImage(bmp))      //Creates a new Graphics object from the bitmap.
    {
        g.DrawImage(img, 0, 0);                      //Fills the empty bitmap with the pixels from "img" starting at pixel co-ordinates (0, 0).
    }
    return bmp;                                       //Returns the unindexed bitmap created with "g".
}

```

EncryptedImage.cs

Set(SourceImage, HiddenData)

This method uses overloading to allow for 2 different set methods – Set() used for when the program is in “decrypt mode” and the both images being handled just need to be loaded and read from. The second Set() overload is Set(SourceImage, HiddenData) for when the program is in “encrypt mode” and the second image is being created from the first (hence the parameters of a SourceImage object and a HiddenData object).

```

public void Set(Data image, HiddenData data)
{
    SetBitmap(data.GetBitstream(), image.GetBitmapFromPath()); //Calls "SetBitmap()" within the class using "Data" and "HiddenData" objects as objects.
    fileType = image.GetFileType();                             //Sets "fileType" - used so the encrypted image is saved as the same type as original image.
}

```

SaveImage()

Method is used to save image files. Has validation checks in the form of ensuring that "bmp" has been assigned a value (so that there is an image to save) and checking that the image to be saved has been named by ensuring the file name isn't empty.

```
public void SaveImage()
{
    if(bmp != null)                //Codepath runs if "bmp" has been generated (i.e. if there is an image to save).
    {
        using (SaveFileDialog dialog = new SaveFileDialog())    //Creates temporary SaveFileDialog object.
        {
            dialog.Title = "Save Image";                        //Sets title of the window.
            dialog.ShowDialog();                                //Shows the dialog window.
            if (dialog.FileName != "")                          //Codepath runs if a save name has been entered.
            {
                bmp.Save(dialog.FileName + fileType);           //Saves the file in the format specified by "fileType".
                                                                //-works by concatenating the filetype at the end of the name.
            }
        }
    }
    else                            //If "bmp" is null then the data hasn't been encrypted.
    {
        MessageBox.Show("Encrypt data first.", "Error", MessageBoxButtons.OK);    //Error message popup.
    }
}
```

SetBitmap(string, Bitmap)

Method generates a new bitmap by XORing each bit of the bitstream with a pixel in a one to one ratio.

```
private void SetBitmap(string bitstream, Bitmap tempImage)    //Assigns "bmp" a value - generated from the objects passed as arguments.
{
    Color colour;                                            //Creates an RGB "Colour" object.
    int i = 0;                                              //Variable used to count through the bitstream.
    int blueValue;                                          //Stores the integer value of the blue component of each pixel.
    for (int y = 0; y < tempImage.Height; y++)            //Loops through the height of the image.
    {
        for (int x = 0; x < tempImage.Width; x++)          //Loops through the width of the image.
        {
            colour = tempImage.GetPixel(x, y);              //Sets "colour" to the current pixel.
            blueValue = colour.B;                            //Sets "blueValue" to the integer value of the blue component of the current pixel.
            blueValue = blueValue ^ int.Parse(bitstream[i].ToString()); //Sets "blueValue" to the result of a bitwise XOR between the current "blueValue" and the current bit in "bitstream".
            colour = Color.FromArgb(colour.A, colour.R, colour.G, blueValue); //Changes the blue component of "colour" to the newly generated blue value.
            tempImage.SetPixel(x, y, colour);                //Replaces the current pixel with the newly generated one.
            i++;                                              //Increments "i" to count through "bitstream".
            if (i == bitstream.Length)                      //Checks if whole bitstream has been looped through.
            {
                x = tempImage.Width;                        //Sets "x" and "y" to their maximum values causing the loops to stop.
                y = tempImage.Height;
            }
        }
    }
    bmp = tempImage;                                        //Sets "bmp" to the newly generated encrypted image.
}
```

HiddenData.cs

Pad(byte) and Pad(long)

```
string Pad(byte b)    //Method for padding bytes to make them 8 bits.
{
    string bytestring = Convert.ToString(b, 2);            //Converts from "byte" datatype to string in base 2.
    if (bytestring.Length < 8)                              //Code runs if string is less than 8 characters.
    {
        while (bytestring.Length != 8)                    //Loops through string until its 8 characters long.
        {
            bytestring = 0 + bytestring;                  //Adds a 0 to the start of the string.
        }
    }
    return bytestring;
}
```



```

string Pad(long b)                                //Method for padding longs to make them 10 bit binary.
{
    b = 1 + (b / 1000);                          //Converts bytes to kilobytes and adds 1kb for padding.
    string bytestring = Convert.ToString(b, 2);   //Converts from "long" datatype to string in base 2.
    if (bytestring.Length < 10)                   //Code runs if string is less than 10 characters.
    {
        while (bytestring.Length != 10)          //Loops through string until its 10 characters long.
        {
            bytestring = 0 + bytestring;          //Adds a 0 to the start of the string.
        }
    }
    return bytestring;
}

```

AddMetaData()

Method used for creating metadata to add at the start of the bitstream.

```

string AddMetaData()                             //Method generates metadata from file type and size.
{
    string metadata = dataTypes[fileType];        //Looks up "fileType" in "dataTypes" dictionary and sets string to result.
    metadata = metadata + Pad(size);              //Concatenates the size of the file in binary (after padding).
    return metadata;
}

```

SetBitstream(object)

Method is used for converting any file type into a bitstream.

```

public void SetBitstream(object bar)              //Method for converting any filetype into bitstream.
{
    var progressBar = (ProgressBar)bar;          //"object" as argument because method is called using multithreading.
    if (dataTypes.ContainsKey(fileType))          //Creating ProgressBar object from argument.
    {                                              //Validates that the selected file is an allowed file (contained in "dataTypes").
        string stream = AddMetaData();            //Sets the start of the string to be the 13 bit metadata.
        byte[] bytes = File.ReadAllBytes(filePath); //Creates a "byte" array filled with each byte of data from the file at "filepath".
        int l = bytes.Length;                    //Sets l to the length of the array.
        progressBar.Maximum = l;                 //Sets the maximum value of the progress bar to the length of the array.
        for (int i = 0; i < l; i++)              //Loops through each element of the array.
        {
            stream = stream + Pad(bytes[i]);      //Concatenates the current byte of data to "stream" after converting it to binary.
            progressBar.Increment(1);             //Increments the progress bar.
        }
        bitstream = stream;                      //Sets "bitstream" to the temporary "stream" variable.
        progressBar.Value = 0;                   //Resets the progress bar.
    }
    else                                          //If the filetype isn't in "fileTypes".
    {
        MessageBox.Show("Invalid file selected.", "Error", MessageBoxButtons.OK); //Error message popup
    }
}

```


ExtractMetaData(Bitmap, Bitmap)

Method that compares the first 13 pixels of 2 images and returns the type and size of the concealed file.

```
public bool ExtractMetaData(Bitmap img1, Bitmap img2) //Method sets "decryptedSize" and "fileType" values.
{
    if (img1.Height != img2.Height && img1.Width != img2.Width) //Validation - makes sure images are the same size.
    {
        return false; //Returns false if not.
    }
    string metadata = ""; //Sets string to empty to allow concatenation.
    int counter = 0;
    for (int y = 0; y < img1.Height; y++) //Loops through the height of the image.
    {
        for (int x = 0; x < img1.Width; x++) //loops through the width of the image.
        {
            metadata = metadata + (img1.GetPixel(x, y).B ^ img2.GetPixel(x, y).B); //^Concatenates result of bitwise XOR between pixel values.
            if (counter == 12) //If looped 13 times.
            {
                y = img1.Height; //Break out of x and y loops.
                x = img1.Width;
            }
            counter = counter + 1;
        }
    }
    fileType = dataTypes[metadata.Substring(0, 3)]; //Looks value with the metadata as the key and sets "fileType".
    decryptedSize = Convert.ToInt32(metadata.Substring(3, 2)); //Sets "decryptedSize" to denary equivalent of binary metadata.
    return true; //Returns true if method executed fully.
}
```

SetBytes(object, object, object)

Method parses the bitstream and converts it into an array of bytes. This method can take a while to complete so it also handles the progress bar. Method has 3 generic "object" parameters because it is invoked with multithreading so can not directly take instances of specific classes as arguments.

```
public void SetBytes(object i1, object i2, object bar) //Method to parse bitstream into array of bytes.
{
    var img1 = (Bitmap)i1; //Converts objects passed as arguments
    var img2 = (Bitmap)i2; //to instances of their actual classes.
    var progressBar = (ProgressBar)bar;
    ExtractData(img1, img2, progressBar); //Calls ExtractData() with converted objects.
    bitstream = bitstream.Remove(0, 13); //Removes metadata from start of bitstream.
    bitstream = RemoveZeros(bitstream); //Removes redundant data from back of bitstream.
    List<byte> bytes = new List<byte>(); //Creates new list, each element is 1 byte of data.
    for (int i = 0; i < bitstream.Length - 9; i = i + 8) //Loops through the string 8 characters at a time
    { //to parse bitstream into a list of bytes.
        bytes.Add(Convert.ToByte(bitstream.Substring(i, 8), 2)); //Adds next chunk of 8 bits to a new element in the list.
    }
    data = bytes.ToArray(); //Converts list to array.
}
```

ExtractData(Bitmap, Bitmap, ProgressBar)

Method compares 2 images and extracts data from the difference between them. Data is extracted using the extraction algorithm described in the Design section of this document. The method has a ProgressBar object as a parameter because it takes a long time to execute so the progress bar gives some user feedback to indicate that the program is working.

```

void ExtractData(Bitmap img1, Bitmap img2, ProgressBar bar)
{
    int counter = 0;
    string stream = "";
    bar.Maximum = decryptedSize*1000*8;
    for (int y = 0; y < img1.Height; y++)
    {
        for (int x = 0; x < img1.Width; x++)
        {
            stream = stream + (img1.GetPixel(x, y).B ^ img2.GetPixel(x, y).B);
            //Concatenates result of bitwise XOR between current pixel values.
            if (counter > (decryptedSize*1000*8))
            {
                y = img1.Height;
                x = img1.Width;
            }
            bar.Increment(1);
            counter = counter + 1;
        }
    }
    bitstream = stream;
    bar.Value = 0;
}

```

RemoveZeros(string)

Method cleans up bitstream by deleting bytes with a value of 0 from the back of it.

```

string RemoveZeros(string bits)
{
    bool done = false;
    int i;
    while(done == false)
    {
        i = bits.Length - 8;
        if (bits.Substring(i) == "00000000")
        {
            bits = bitstream.Remove(i);
        }
        else
        {
            done = true;
        }
    }
    return bits;
}

```

SaveFile()

Method for handling the file dialog when saving a decrypted file.

```
public void SaveFile()
{
    if(data != null) //Ensures that "data" has been assigned a value.
    {
        SaveFileDialog dialog = new SaveFileDialog(); //Creates new SaveFileDialog object.
        dialog.Title = "Save File"; //Sets window title.
        dialog.ShowDialog();
        if (dialog.FileName != "") //Ensures the file has been named.
        {
            File.WriteAllBytes(dialog.FileName + fileType, data); //Writes the data to a file with the
                                                                    //file type at the end of the name
                                                                    //to save it in the right format.
        }
    }
    else //Code runs if "data" hasn't been set.
    {
        MessageBox.Show("Decrypt data before saving.");
    }
}
```

Testing

Test Plan

Test No.	Description	Expected Result
Testing Program Features		
1	Display images in the form.	Images are displayed correctly in the form.
2	Select a file by clicking "Browse".	File dialog pop-up, correct file path displayed.
3	Encrypt file by clicking "Encrypt File".	Image appears in picture box on the right.
4	Save image by clicking "Save Image".	File dialog pop-up, image saves.
5	Decrypt data by clicking "Decrypt File".	Data extracted from images – correct file type displayed.
6	Save file by clicking "Save File".	File dialog pop-up, file saves.
7	Images save correctly.	Encrypted images are saved in the same format as original image.
8	Files save correctly.	Decrypted files are saved in the same format as original file.
9	Certain features inaccessible in encrypt mode.	"Decrypt File" and "Save File" inaccessible.
10	Certain features inaccessible in decrypt mode.	"Browse", "Encrypt" and "Save Image" inaccessible.
11	Multithreading works.	Program still responsive while loading bar is running.
12	"Maximum file size" label displays correct value.	Number displayed is 1/8 * dimensions of the image.
Validation Tests		
13	Invalid file type selected for image.	Suitable error message.
14	Invalid file type selected for file to encrypt.	Suitable error message.
15	"Encrypt File" selected before image/file have been selected.	Suitable error message.
16	"Save Image" selected before encryption.	Suitable error message.
17	"Decrypt File" selected before image/file have been selected.	Suitable error message.
18	"Save File" selected before decryption.	Suitable error message.
19	Try to decrypt from 2 clearly different images.	Suitable error message.
20	Try to encrypt a file that is too large to conceal in the image.	Suitable error message.
Testing Against Objectives		
21	Encrypt data in .jpg image.	Image displayed appears identical to original.
22	Encrypt data in .png image.	Image displayed appears identical to original.

23	Encrypt data in .gif image.	Image displayed appears identical to original.
24	Encrypt data in .tif image.	Image displayed appears identical to original.
25	Encrypt/decrypt .txt file in an image.	Decrypted data is exact copy of original data.
26	Encrypt/decrypt .pdf file in an image.	Decrypted data is exact copy of original data.
27	Encrypt/decrypt .jpg file in an image.	Decrypted data is exact copy of original data.
28	Encrypt/decrypt .gif file in an image.	Decrypted data is exact copy of original data.
29	Encrypt/decrypt .mp3 file in an image.	Decrypted data is exact copy of original data.
30	Decrypt data from .jpg image.	Decrypted data is exact copy of original data.
31	Decrypt data from .png image.	Decrypted data is exact copy of original data.
32	Decrypt data from .gif image.	Decrypted data is exact copy of original data.
33	Decrypt data from .tif image.	Decrypted data is exact copy of original data.

Testing

Evidence for these tests can be found at the following link:

<https://www.youtube.com/watch?v=nP6VlplyAbo>

The table includes a timestamp for each test to allow easy navigation.

Test No. **Timestamp** **Pass/Fail**

Testing Program Features		
1	01:00	Pass
2	00:43	Pass
3	01:37	Pass
4	01:42	Pass
5	03:50	Pass
6	03:55	Pass
7	02:49	Pass
8	04:20	Pass
9	01:32	Pass
10	03:17	Pass
11	11:29	Pass
12	01:00	Pass
Validation Tests		
13	14:27	Pass
14	14:33	Pass
15	15:56	Pass
16	01:35	Pass
17	14:45	Pass
18	14:54	Pass
19	15:08	Pass
20	15:21	Pass

Testing Against Objectives

21	02:15	Pass
22	05:08	Pass
23	07:47	Pass
24	09:33	Pass
25	04:20	Pass
26	06:25	Pass
27	08:50	Pass
28	12:28	Pass
29	10:33	Pass
30	04:20	Pass
31	06:25	Pass
32	08:50	Pass
33	10:33	Pass

Evaluation

Fulfilment of Objectives

All the objectives of the project were fulfilled.

Objective 1

Objective: Program can encrypt a range of **File Types** into a range of **Image File Types** using steganographic techniques.

Fulfilled: Yes

Evidence: Test No. 3, 21-30

Objective 2

Objective: Images holding concealed data are indistinguishable from original images when viewed side by side in high resolution.

Fulfilled: Yes

Evidence: See video test or appendix comparing images.

Objective 3

Objective: Program can take a range of **Image File Types** and extract data in a range of **File Types** by comparing it to another image of the same type.

Fulfilled: Yes

Evidence: Test No.5, 25-33

Objective 4

Objective: The system provides a means for people to share data securely.

Fulfilled: Yes

Evidence: Peer feedback – images sent between computers with software installed were sent securely and able to be decrypted. A 3rd party intercepting the images would not be able to access the hidden data due to the lack of software and lack of knowledge that data is, in fact, hidden.

How Program Improved

To begin with, I had planned for the program to only be compatible with certain file types and I intended to write separate protocols to handle each file type. During the development of the project, however, I discovered that my software could easily be compatible with virtually all file types due to the encryption and decryption algorithms working on a binary level and returning exactly the same data that was encrypted. The only limiting factor for the range of file types that can be encrypted and decrypted is the size of the metadata stored at the start of the bitstream – with only 3 bits to represent the file type there is a maximum of 8 possible file types that can be represented with the current version of the software.

Feedback

I shared my program with my peers to test and give me feedback – the only problem highlighted was a problem where saved images aren't always the same size as the originals (as mentioned below).

The only other feedback I received was that the program works exactly how it is intended – providing a method of concealing data to allow for sharing without revealing the presence of sensitive data.

Areas for Further Development

Compression Issue

An issue that arose while writing the encryption side of the application is that the program doesn't compress images before saving (tests 4a, 4b and 4c). Because the program converts images to bitmaps to process the data, images become decompressed when they are being processed. This is not an issue other than the fact that, when it comes to saving, images are converted back into their original format and not compressed in the process which makes them much larger than their original counterparts:

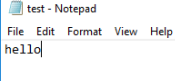
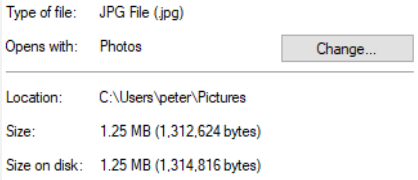
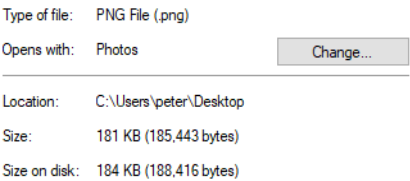
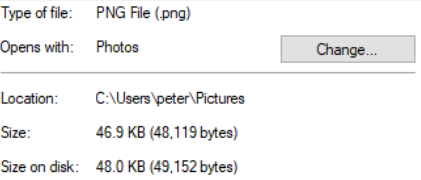
Original Image		Encrypted Copy	
Location:	C:\Users\peter\Desktop	Location:	C:\Users\peter\Desktop
Size:	181 KB (185,443 bytes)	Size:	35.7 MB (37,463,510 bytes)
Size on disk:	184 KB (188,416 bytes)	Size on disk:	35.7 MB (37,466,112 bytes)

This size difference makes no difference to the encryption and decryption of data – images can still be compared pixel by pixel and have the correct data extracted.

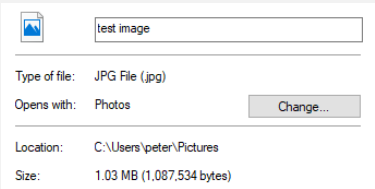


Had I had more time to develop my project I would have addressed this issue and added functionality to compress images before saving them.

Appendices

Testing Carried Out During Development


#	Description	Input
1	Test that the change source method works by calling it and checking the returned file path is correct	Select a test image from desktop.
2	Test that the generic "Bitstream" method creates the same bitstream as the "TextBitsream" method when given the same data.	
3	Testing that images save in the correct format with the correct name.	Select "Mandel_zoom_00_mandelbrot_set.jpg" from "C:\Users\peter\Pictures" and save it through the program.
4	Comparing the sizes of unencrypted and encrypted versions of the same image in various file formats. I will use the same lorem ipsum placeholder text for each test.	Column will show the original file size and format of the image that is being processed.
a	Jpg	
b	Png	
c	Png (test 2)	

After Refactoring

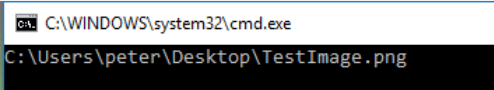
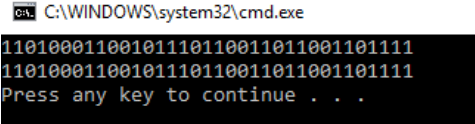
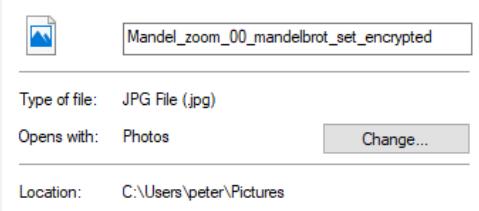
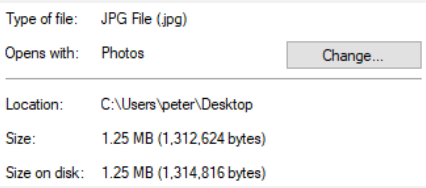
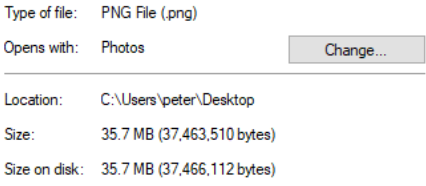
5	"Set()" sets "filePath", "fileType" and "size" correctly	
6	Selecting a file in the dialog opens shows that image in the form.	
7a	Program allows user to cancel selection of file.	Select "cancel" in image browse window.
7b	Program allows user to cancel selection of file (after modifications to code).	Select "cancel" in image browse window.
8a	Program only allows selection of image files.	Select a text file instead of an image in browse window.
8b	Program only allows selection of image files (after modifications to code).	Select a text file instead of an image in browse window.
9a	Test that SetBitmap() works.	
9b	Test that SetBitmap() works.	Select the 4x4 bitmap for the source image then select a text file, which contains only the letter 'a', to encrypt into the image.
10	Test that "ExtractData()" extracts bitstreams from 2 images correctly	Selecting 4x4testbitmap and a version of it which has been encrypted with a text file containing the character 'a'.
11a	Test that setBitstream works	Testing using a text file containing "a 0~".
11b	Test that setBitstream works (with ammended code)	Testing using a text file containing "a 0~".
12	Test that "ExtractData()" extracts bitstreams from 2 images correctly (png file).	Selecting an unencrypted image and one with a 128x128 png encrypted and saving the results.
13	Test to check that image doesn't encrypt data if the data to encrypt is larger than the available space provided by the source image (1 bit per pixel i.e. 1/24 the size of the image excluding metadata).	 <p>Select the same uncompressed image (above) for source and encrypted - shouldn't be able to encrypt as second file has to be about 1/24 size of original file.</p>
14	Test to check the values stored in source.availaleSize and encrypted.fileSize are correct.	

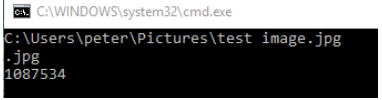
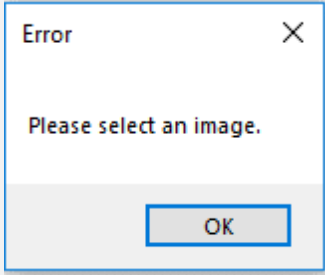
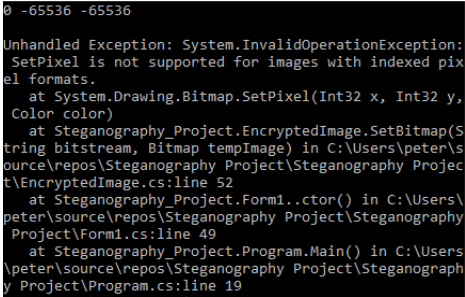
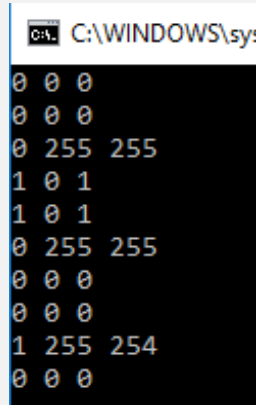
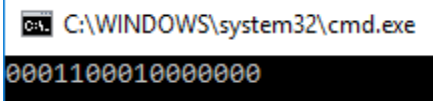
15


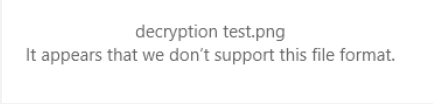

Test to check that image doesn't encrypt data if the data to encrypt is larger than the available space provided by the source image (1 bit per pixel i.e. 1/24 the size of the image excluding metadata) (ammended code).



Select the same uncompressed image (above) for source and encrypted - shouldn't be able to encrypt as second file has to be about 1/24 size of original file.

#	Expected Output	Actual Output	
1	Console outputs the file path of the image.		Pass
2	Console outputs 2 identical bitstreams representing "hello" in ASCII (1101000110010111011001101100110111)		Pass
3	"Mandel_zoom_00_mandelbrot_set.jpg" saved in "C:\Users\peter\Pictures".		Pass
4		Column will show the file size and format of the processed image.	
a			Pass
b			Fail

c		Error message.	Fail
5	C:\Users\peter\Pictures\test image.jpg .jpg 1087534		Pass
6			Pass
7a	Image browse window closes and form remains at state before window was opened.	Error message	Fail
7b	Image browse window closes and form remains at state before window was opened.	Image browse window closes and form remains at state before window was opened.	Pass
8a	Browse window doesn't allow selection.	Error message	Fail
8b	Message box shows prompting user to select an image file.		Pass
9a	0 4294901760 4294901760 0 4278255360 4278255360 0 4278190335 4278190335 1 4294901760 4294901761 1 4278255360 4278255361 1 4278190335 4278190334		Fail
9b	('b' is represented by 98 in ASCII and therefore 1100010 in binary) (metadata will also have been added so bitstream will be 0001100010) In console: 0 0 0 0 0 0 0 255 255 1 0 1 1 0 1 0 255 255 0 0 0 0 0 0 1 255 254 0 0 0		Pass
10	In console: 0000110000100000 (first 3 bits signify it is a text file, the next 8 are the character and the last 5 are unencrypted bits)		Fail

11a	In console: 000011000010010000000011000001111110		Fail
11b	In console: 000011000010010000000011000001111110		Pass
12	This image saved 		Fail
13	Error message –“File too large” 	Image has been encrypted.	Fail
14	Available size should be $256 * 256 / 8 = 8192$ (height * width / 8). Encrypted file size should be 32,886		Fail
15	Error message –“File too large” 		Pass

Source Code

Form1.cs

```
using System;
using System.Threading;
using System.Windows.Forms;

namespace Steganography_Project
{
    public partial class Form1 : Form
    {
        Thread alternateThread;

        SourceImage source = new SourceImage();
        EncryptedImage encrypted = new EncryptedImage();

        HiddenData hidden = new HiddenData();

        void resetForm(bool state)
        {
            source = new SourceImage();
            encrypted = new EncryptedImage();
            hidden = new HiddenData();

            sourceImg.Image = null;
            encryptedImg.Image = null;

            textBoxFileToEncrypt.Text = null;
            labelFileType.Text = "File type:";
            labelMaxData.Text = "Maximum file size:";

            checkBoxDecrypt.Checked = !state;

            checkBoxEncrypt.Checked = state;
            btnEncrypt.Enabled = state;
            btnSaveImage.Enabled = state;
            textBoxFileToEncrypt.Enabled = state;
            btnBrowseLoadFile.Enabled = state;
            btnDecryptFile.Enabled = !state;
            btnSaveFile.Enabled = !state;
            toolTipEncryptEnabled.Active = state;
            toolTipDecryptEnabled.Active = !state;
        }

        public Form1()
        {
            InitializeComponent();
        }

        private void sourceImg_Click(object sender, EventArgs e)
        {
            if (source.Set())
            {
                sourceImg.Image = source.GetBitmapFromPath();

                labelMaxData.Text = ("Maximum file size: " + source.GetAvailableSize() + " bytes.");
            }
            else
            {
                source = new SourceImage();

                sourceImg.Image = null;
            }
        }

        private void checkBoxEncrypt_CheckedChanged(object sender, EventArgs e)
        {
            if (checkBoxEncrypt.Checked)
            {
                //Thread used at points where program slows down
                //allows for a loading bar to show the program still
                //working.
                //Object handles the source image.
                //Object handles the encrypted version of the source
                //image.
                //Object handles the data thats hidden within the
                //image.

                //Method to reset the form
                //Resets the objects

                //Resets images

                //Resets text

                //Enables/disables buttons based on whether true or
                //false argument passed

                //Runs Set() in "source" - if true returned:
                //values for "source" have been set and
                //following code can run.

                //Sets the "sourceImg" PictureBox in the form to
                //the image at filepath of "source".

                //Resets the Data object to avoid problems with
                //values having already been set.
            }
        }
    }
}
```

```

        resetForm(true); // "true" represents state of checkBoxEncrypt
    }
}

private void checkBoxDecrypt_CheckedChanged(object sender, EventArgs e)
{
    if (checkBoxDecrypt.Checked)
    {
        resetForm(false);
    }
}

#region Code for encrypting images

// Events that may occur when "checkBoxEncrypt" is checked.

private void btnEncrypt_Click(object sender, EventArgs e)
{
    try // Exception handling - ensures program doesn't crash if subroutines are trying to execute with null values.

    {
        if (source.GetAvailableSize() > hidden.GetFileSize()) // Checks that the file represented by "source" is big enough to conceal the file represented by "hidden".

        {
            encrypted.Set(source, hidden); // Overloading - sets "encrypted" with the Data and HiddenData objects as arguments.
            encryptedImg.Image = encrypted.GetBitmap(); // Sets the "encryptedImg" PictureBox in the form to the image just created by encrypting file into "source".
        }
        else // Runs if the file represented by "hidden" is too large to fit into file represented by "source".

        {
            MessageBox.Show("File too large to encrypt - please select a larger image or a smaller file.", "Error", MessageBoxButtons.OK);
        }
    }
    catch // Runtime error will occur if values for "source" or "hidden" haven't been set yet.

    {
        MessageBox.Show("Ensure all data values are filled.", "Error", MessageBoxButtons.OK);
    }
}

private void btnSaveImage_Click(object sender, EventArgs e)
{
    encrypted.SaveImage();
}

private void btnBrowseLoadFile_Click(object sender, EventArgs e)
{
    if (hidden.Set()) // Runs Set() in "hidden" - if true returned: values for "hidden" have been set and following code can run.

    {
        textBoxFileToEncrypt.Text = hidden.GetFilePath(); // Shows the filepath of the file on the form.
        alternateThread = new Thread(hidden.SetBitstream); // Creates a new thread used to execute "hidden.SetBitstream()" as the method takes a long time to finish.
        // Having another thread means that the form can still be active while "hidden.SetBitstream()" works in the background.

        alternateThread.Start(progressBar); // Starts the thread passing "progressBar" as an argument - this effectively runs "hidden.SetBitstream(progressBar)" in the background.
    }
}

#endregion

#region Code for decrypting images

// Events that may occur when "checkBoxDecrypt" is checked.

```

```

private void encryptedImg_Click(object sender, EventArgs e)
{
    //Method works in the same way as
    // "sourceImg_Click()" but for "encrypted" and
    // "encryptedImg" rather than "source" and
    // "sourceImg".
    if (checkBoxDecrypt.Checked && encrypted.Set())
    {
        //Runs Set() with no arguments in "encrypted"
        // - if true returned: values for "encrypted"
        // have been set and following code can run.
        encryptedImg.Image = encrypted.GetBitmapFromPath();
        Console.WriteLine(encrypted.GetFilePath());
    }
    else
    {
        encrypted = new EncryptedImage();
        encryptedImg.Image = null;
    }
}

private void btnSaveFile_Click(object sender, EventArgs e)
{
    hidden.SaveFile();
}

private void btnDecryptFile_Click(object sender, EventArgs e)
{
    if(source.GetFilePath() != null && encrypted.GetFilePath() != null)
    {
        //Checks "filePath" in "source"
        // and "filePath" in "encrypted" have
        // been assigned values
        if(hidden.ExtractMetaData(source.GetBitmapFromPath(), encrypted.GetBitmapFromPath()))
        {
            //Code
            // only runs if ExtractMetaData returns
            // true (if it executes correctly)
            {
                labelFileType.Text = "File type: " + hidden.GetFileType();
                //Displays the file type on the form.
                alternateThread = new Thread(() => hidden.SetBytes(source.GetBitmapFromPath(),
                encrypted.GetBitmapFromPath(), progressBar)); //use lambda function to allow multiple arguments to be passed.
                alternateThread.Start(); //Starts the new thread
            }
            else
            {
                //Codepath runs if an error occurred while executing ExtractMetadata()
                {
                    MessageBox.Show("Enusure images are the same size.", "Error", MessageBoxButtons.OK);
                }
            }
        }
        else
        {
            //Codepath runs if "filePath" in "source" or "filePath" in
            // "encrypted" are null.
            {
                MessageBox.Show("Select 2 images.", "Error", MessageBoxButtons.OK);
            }
        }
    }
    #endregion
}
}

```


Data.cs

```
using System.Windows.Forms;
using System.IO;

namespace Steganography_Project
{
    abstract class Data //Class is abstract and therefore never instantiated - is
                        //generalization of SourceImage and HiddenData.
    {
        protected string filePath; //Protected allows access only to this and derived classes.
        protected string fileType;
        protected long size;

        public virtual bool Set() //bool - return value used as validation check to
                                //ensure file was selected and values set.
                                //virtual - allows overriding in the derived
                                //class.
        {
            using (OpenFileDialog dialog = new OpenFileDialog()) //Creates a temporary OpenFileDialog object to
                                                                //browse computers files.
            {
                if (dialog.ShowDialog() == DialogResult.OK) //Shows the dialog box and runs code if choice was
                                                            //selected (i.e. OK button pressed).
                {
                    filePath = dialog.FileName; //Sets "filePath" to the file path of the file
                                                //selected in the dialog window.
                    fileType = Path.GetExtension(filePath); //Sets "fileType" to the image file type so it can
                                                            //be saved in the same format once encrypted.
                    size = new FileInfo(filePath).Length; //Sets "size" the size of the file in bytes.
                    return true;
                }
                else
                {
                    return false; //Returns false if "cancel" was selected in
                                //dialog.
                }
            }
        }

        public string GetFilePath()
        {
            return filePath;
        }

        public string GetFileType()
        {
            return fileType;
        }

        public long GetFileSize()
        {
            return size;
        }
    }
}
```

SourceImage.cs

```
using System.Windows.Forms;
using System.Drawing;

namespace Steganography_Project
{
    class SourceImage : Data
    {
        private int availableSize; //Stores the maximum file size that can be
                                   //concealed in bytes.

        //Method written due to bug with images using dictionary based compression.
        protected static Bitmap ConvertToUnindexed(Image img) //This method creates an uncompressed bitmap from
                                                                //an indexed image.
        {
            //((Below) Creates new Bitmap object with
            //dimensions of "img" - using 24 bits per pixel
            //(8 bits per colour).
            Bitmap bmp = new Bitmap(img.Width, img.Height, System.Drawing.Imaging.PixelFormat.Format24bppRgb);
            using (Graphics g = Graphics.FromImage(bmp)) //Creates a new Graphics object from the bitmap.
            {
                g.DrawImage(img, 0, 0); //Fills the empty bitmap with the pixels from
                                         //"img" starting at pixel co-ordinates (0, 0).
            }
            return bmp; //Returns the unindexed bitmap created with "g".
        }

        public override bool Set() //Overrides Set() from parent class.
        {
            bool value = base.Set(); //Calls original "Set()" from parent class (not recursion) - sets "value"
                                     //to result.
            try //Exception handling - if "GetBitmapFromPath()" runs then an image was
                {
                    availableSize = GetBitmapFromPath().Height * GetBitmapFromPath().Width / 8; //Calculates
                                                                                               //"availableSize".
                    return value; //Returns result of base "Set()" - true if a file was selected, false
                                 //otherwise.
                }
            catch //If the user selected a non-image file then "GetBitmapFromPath()" would
                {
                    have caused a runtime error.
                    {
                        MessageBox.Show("Please select an image.", "Error", MessageBoxButtons.OK); //Error message popup.
                        return false; //Returns false to show values were not set.
                    }
                }
        }

        public int GetAvailableSize()
        {
            return availableSize;
        }

        public Bitmap GetBitmapFromPath() //Method returns the unindexed bitmap of the image represented by
                                          //"Data".
        {
            Bitmap bmp = new Bitmap(@filePath); //Creates a new bitmap from the file stored at "filepath".
            bmp = ConvertToUnindexed(bmp); //Converts the bitmap to an unindexed version of itself.
            return bmp; //Returns newly generated Bitmap.
        }
    }
}
```

EncryptedImage.cs

```
using System.Drawing;
using System.Windows.Forms;
```

```
namespace Steganography_Project
```

```
{
    class EncryptedImage : SourceImage
    {
        //All attributes used for encrypting. When decrypting the object acts as a plain "data" object.

        private Bitmap bmp;

        public void Set(SourceImage image, HiddenData data)

        {
            SetBitmap(data.GetBitstream(), image.GetBitmapFromPath());

            fileType = image.GetFileType();
        }

        private void SetBitmap(string bitstream, Bitmap tempImage)

        {
            Color colour;
            int i = 0;
            int blueValue;
            for (int y = 0; y < tempImage.Height; y++)
            {
                for (int x = 0; x < tempImage.Width; x++)
                {
                    colour = tempImage.GetPixel(x, y);

                    blueValue = colour.B;

                    blueValue = blueValue ^ int.Parse(bitstream[i].ToString());

                    colour = Color.FromArgb(colour.A, colour.R, colour.G, blueValue);

                    tempImage.SetPixel(x, y, colour);

                    i++;
                    if (i == bitstream.Length)
                    {
                        x = tempImage.Width;

                        y = tempImage.Height;
                    }
                }
            }
            bmp = tempImage;
        }

        public void SaveImage()
        {
            if (bmp != null)
                //Codepath runs if "bmp" has been generated (i.e. if there is an image to
                save).
```

```
        //Overloading - Both set methods
        set values but this method is used
        when creating new image from the
        original image and a bistream.
```

```
        //Calls "Setbitmap()" within the
        class using "Data" and "HiddenData"
        objects as objects.
        //Sets "fileType" - used so the
        encrypted image is saved as the same
        type as original image.
```

```
        //Assigns "bpm" a value
        generated from the objects passed as
        arguments.
```

```
        //Creates an RGB "Colour"
        object.
        //Variable used to count
        through the bitstream.
        //Stores the integer value of
        the blue component of each pixel.
        //Loops through the height of
        the image.
```

```
        //Loops through the width of
        the image.
```

```
        //Sets "colour" to the current
        pixel.
        //Sets "blueValue" to the
        integer value of the blue component
        of the current pixel.
        //Sets "blueValue" to the
        result of a bitwise XOR between the
        current "blueValue" and the current
        bit in "bitstream".
        //Changes the blue component
        of "colour" to the newly generated
        blue value.
        //Replaces the current pixel
        with the newly generated one.
        //Increments "i" to count
        through "bitstream".
        //Checks if whole bistream has
        been looped through.
```

```
        //Sets "x" and "y" to their
        maximum values causing the loops to
        stop.
```

```
        //Sets "bmp" to the newly
        generated encrypted image.
```

```

{
    using (SaveFileDialog dialog = new SaveFileDialog()) //Creates temporary SaveFileDialog object.
    {
        dialog.Title = "Save Image"; //Sets title of the window.
        dialog.ShowDialog(); //Shows the dialog window.
        if (dialog.FileName != "") //Codepath runs if a save name has been entered.
        {
            bmp.Save(dialog.FileName + fileType); //Saves the file in the format specified by "fileType".
                                                //-works by concatenating the filetype at the end of the name.
        }
    }
}
else //If "bmp" is null then the data hasn't been encrypted.
{
    MessageBox.Show("Encrypt data first.", "Error", MessageBoxButtons.OK); //Error message popup.
}
}

public Bitmap GetBitmap()
{
    return ConvertToUnindexed(bmp);
}
}
}

```

HiddenImage.cs

```
using System;
using System.Collections.Generic;
using System.IO;           //Allows Use of "File" class.
using System.Drawing;
using System.Windows.Forms;

namespace Steganography_Project
{
    class HiddenData : Data
    {
        int decryptedSize;           //Holds the size of the decrypted file in kb;

        static Dictionary<string, string> dataTypes = new Dictionary<string, string>() //dictionary has both values
                                                                                       as keys and pairs to allow
                                                                                       reverse lookup

        {
            { ".txt", "000" },
            { ".pdf", "001" },
            { ".jpg", "010" },
            { ".gif", "011" },
            { ".mp3", "100" },
            { ".mp4", "101" },
            { ".zip", "110" },
            { "000", ".txt" },
            { "001", ".pdf" },
            { "010", ".jpg" },
            { "011", ".gif" },
            { "100", ".mp3" },
            { "101", ".mp4" },
            { "110", ".zip" },
        };

        string bitstream;

        #region Attributes used for encrypting data

        string Pad(byte b)           //Method for padding bytes to make them 8 bits.
        {
            string bytestring = Convert.ToString(b, 2); //Converts from "byte" datatype to string in base 2.
            if (bytestring.Length < 8) //Code runs if string is less than 8 characters.
            {
                while (bytestring.Length != 8) //Loops through string until its 8 characters long.
                {
                    bytestring = 0 + bytestring; //Adds a 0 to the start of the string.
                }
            }
            return bytestring;
        }

        string Pad(long b)           //Method for padding longs to make them 10 bit binary.
        {
            b = 1 + (b / 1000); //Converts bytes to kilobytes and adds 1kb for padding.
            string bytestring = Convert.ToString(b, 2); //Converts from "long" datatype to string in base 2.
            if (bytestring.Length < 10) //Code runs if string is less than 10 characters.
            {
                while (bytestring.Length != 10) //Loops through string until its 10 characters long.
                {
                    bytestring = 0 + bytestring; //Adds a 0 to the start of the string.
                }
            }
            return bytestring;
        }

        string AddMetaData()           //Method generates metadata from file type and size.
        {
            string metadata = dataTypes[fileType]; //Looks up "fileType" in "dataTypes" dictionary and sets string
            metadata = metadata + Pad(size); //Concatenates the size of the file in binary (after padding).
            return metadata;
        }

        public void SetBitstream(object bar) //Method for converting any filetype into
                                                                                       bitstream.
    }
}
```

```

{
    var progressBar = (ProgressBar)bar;
    if (dataTypes.ContainsKey(fileType))
    {
        string stream = AddMetaData();

        byte[] bytes = File.ReadAllBytes(filePath);

        int l = bytes.Length;
        progressBar.Maximum = l;

        for (int i = 0; i < l; i++)
        {
            stream = stream + Pad(bytes[i]);

            progressBar.Increment(1);
        }
        bitstream = stream;

        progressBar.Value = 0;
    }
    else
    {
        MessageBox.Show("Invalid file selected.", "Error", MessageBoxButtons.OK); //Error message popup
    }
}

public string GetBitstream()
{
    return bitstream;
}

#endregion

#region Attributes used for decrypting data

byte[] data;

public bool ExtractMetaData(Bitmap img1, Bitmap img2) //Method sets "decryptedSize" and "fileType" values.
{
    if (img1.Height != img2.Height && img1.Width != img2.Width) //Validation - makes sure images are the same size.
    {
        return false; //Returns false if not.
    }
    string metadata = ""; //Sets string to empty to allow concatenation.

    int counter = 0;
    for (int y = 0; y < img1.Height; y++) //Loops through the height of the image.
    {
        for (int x = 0; x < img1.Width; x++) //loops through the width of the image.
        {
            metadata = metadata + (img1.GetPixel(x, y).B ^ img2.GetPixel(x, y).B); //^Concatenates result of bitwise XOR between pixel values.
            //If looped 13 times.

            if (counter == 12) //Break out of x and y loops.
            {
                y = img1.Height;
                x = img1.Width;
            }
            counter = counter + 1;
        }
    }
    fileType = dataTypes[metadata.Substring(0, 3)]; //Looks value with the metadata as the key and sets "fileType".
    decryptedSize = Convert.ToInt32(metadata.Substring(3), 2); //Sets "decryptedSize" to denary equivalent of binary metadata.
    return true; //Returns true if method executed fully.
}

void ExtractData(Bitmap img1, Bitmap img2, ProgressBar bar)
//Method to compare 2 images and extract data from them - sets "bitstream" string

```

```

{
    int counter = 0;
    string stream = "";
    bar.Maximum = decryptedSize*1000*8;
    for (int y = 0; y < img1.Height; y++)
    {
        for (int x = 0; x < img1.Width; x++)
        {
            stream = stream + (img1.GetPixel(x, y).B ^ img2.GetPixel(x, y).B);
            //Concatenates result of bitwise XOR between current
            //pixel values.

            if (counter > (decryptedSize*1000*8))
            {
                y = img1.Height;
                x = img1.Width;
            }
            bar.Increment(1);
            counter = counter + 1;
        }
    }
    bitstream = stream;
    bar.Value = 0;
    //Resets the bar value.
}

public void SetBytes(object i1, object i2, object bar)
{
    var img1 = (Bitmap)i1;
    var img2 = (Bitmap)i2;
    var progressBar = (ProgressBar)bar;
    ExtractData(img1, img2, progressBar);

    bitstream = bitstream.Remove(0, 13);

    bitstream = RemoveZeros(bitstream);

    List<byte> bytes = new List<byte>();

    for (int i = 0; i < bitstream.Length - 9; i = i + 8)
    {
        bytes.Add(Convert.ToByte(bitstream.Substring(i, 8), 2));
    }
    data = bytes.ToArray();
}

string RemoveZeros(string bits)
{
    bool done = false;
    int i;
    while(done == false)
    {
        i = bits.Length - 8;
        if (bits.Substring(i) == "00000000")
        {
            bits = bitstream.Remove(i);
        }
        else
        {
            done = true;
        }
    }
    return bits;
}

public void SaveFile()
{
    if(data != null)
    {
        SaveFileDialog dialog = new SaveFileDialog();
        dialog.Title = "Save File";
        dialog.ShowDialog();
        if (dialog.FileName != "")
        {
            File.WriteAllBytes(dialog.FileName + fileType, data);
        }
    }
}

```

//Sets string to empty to allow concatenation.
//Sets bar values to allow easy incrementation.
//Loops through the height of the image.
//Loops through the width of the image.
//Checks if loop has executed enough times
//for all data to have been extracted.
//Breaks out of x and y loops.
//Method to parse bitstream into array of bytes.
//Converts objects passed as arguments
//to instances of their actual classes.
//Calls ExtractData() with converted objects.
//Removes metadata from start of bitstream.
//Removes redundant data from back of bitstream.
//Creates new list, each element is 1 byte of data.
//Loops through the string 8 characters at a time
//to parse bitstream into a list of bytes.
//Adds next chunk of 8 bits to a new element in the list.
//Converts list to array.
//Method to remove redundant data from back of a bitstream.
//Used for while loop.
//Counter variable.
//Loop decrements 8 characters (1 byte) at a time.
//If the last 8 characters are 0.
//Remove them.
//If the last 8 characters arent 0.
//Break while loop.
//Creates new SaveFileDialog object.
//Sets window title.
//Ensures the file has been named.
//Writes the data to a file with the
//file type at the end of the name
//to save it in the right format.

```

else //Code runs if "data" hasn't been set.
{
    MessageBox.Show("Decrypt data before saving.");
}
}
#endregion
}
}

```

Evidence of Program Working

Encrypt Mode

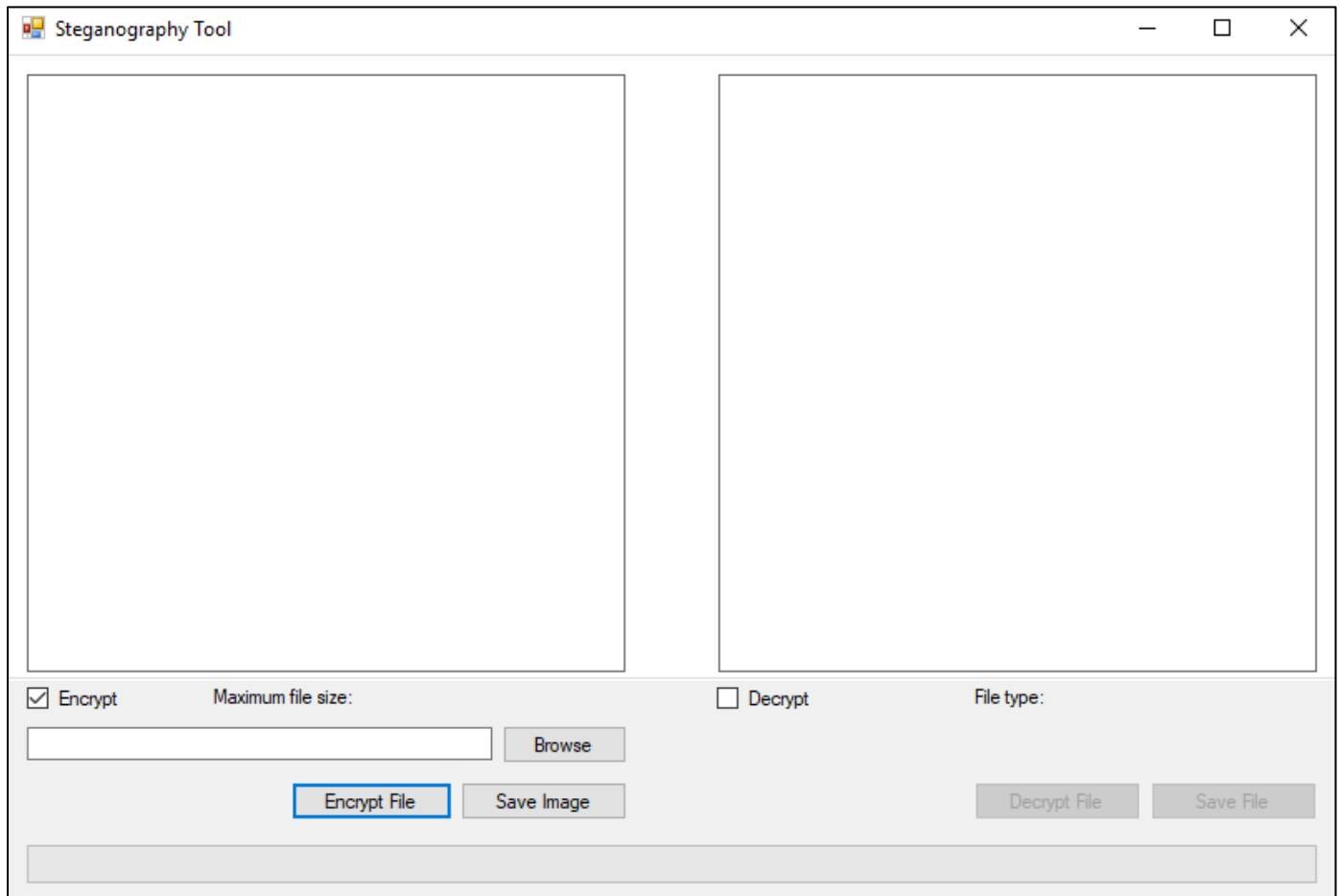
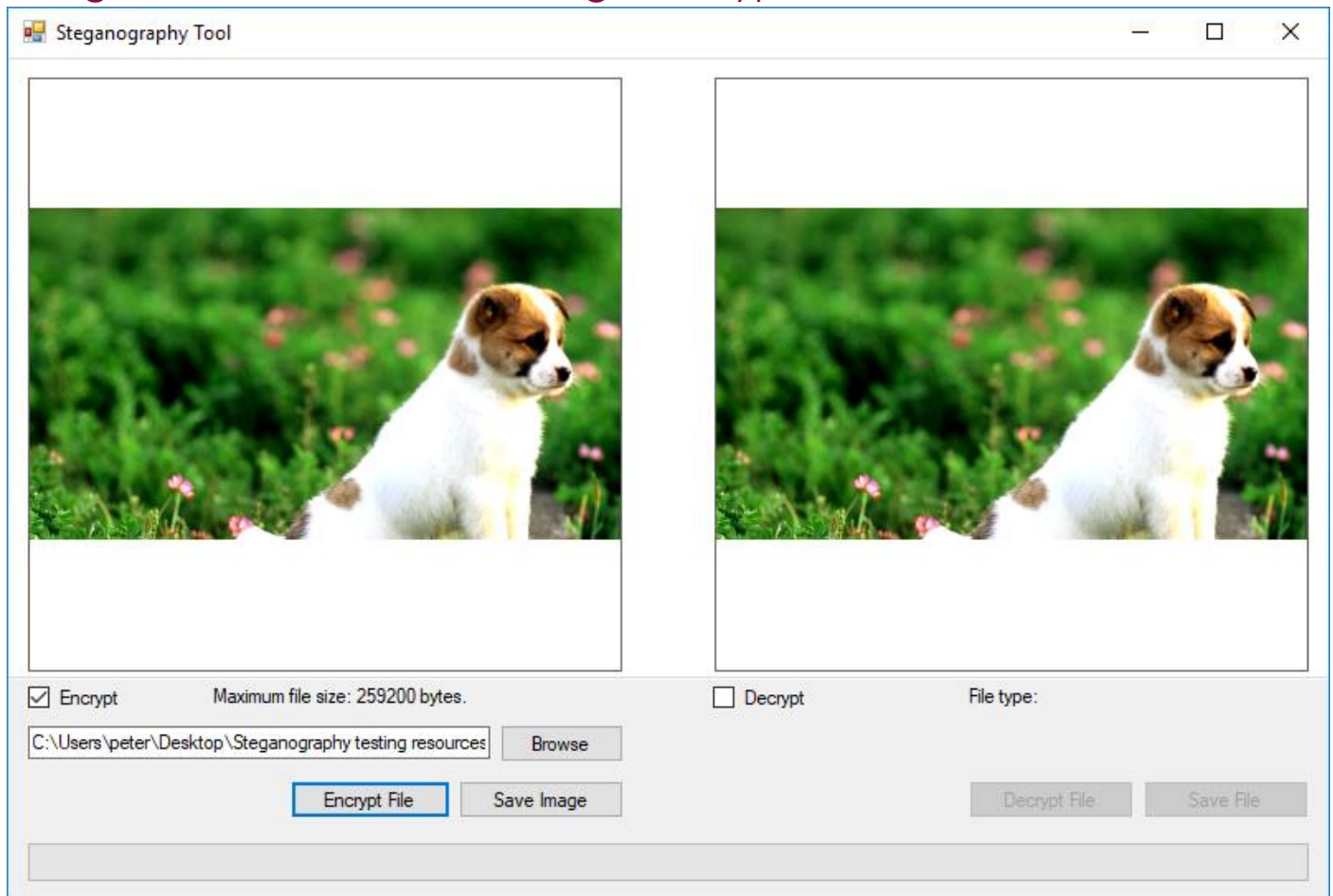
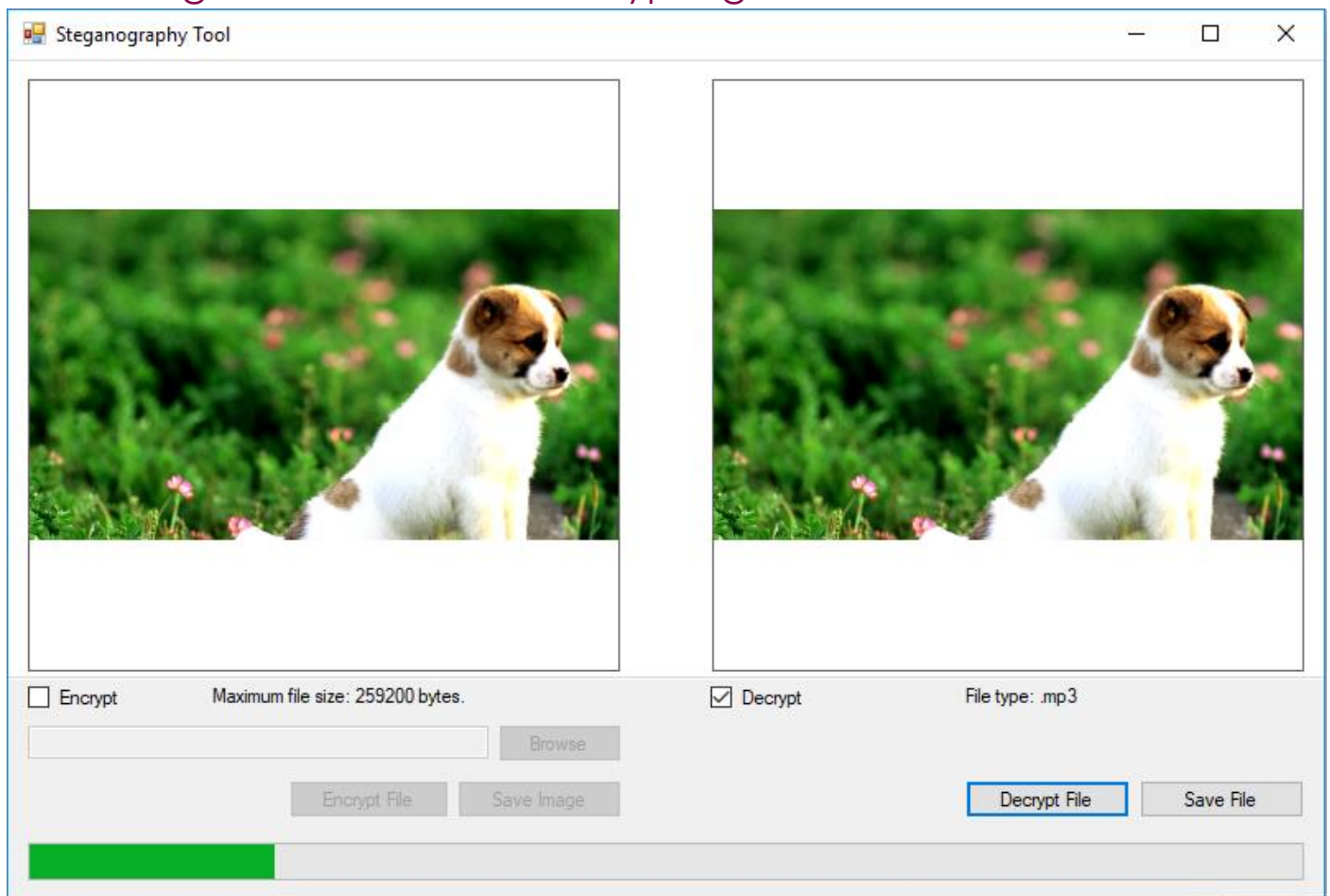


Image and File Loaded – Image Encrypted



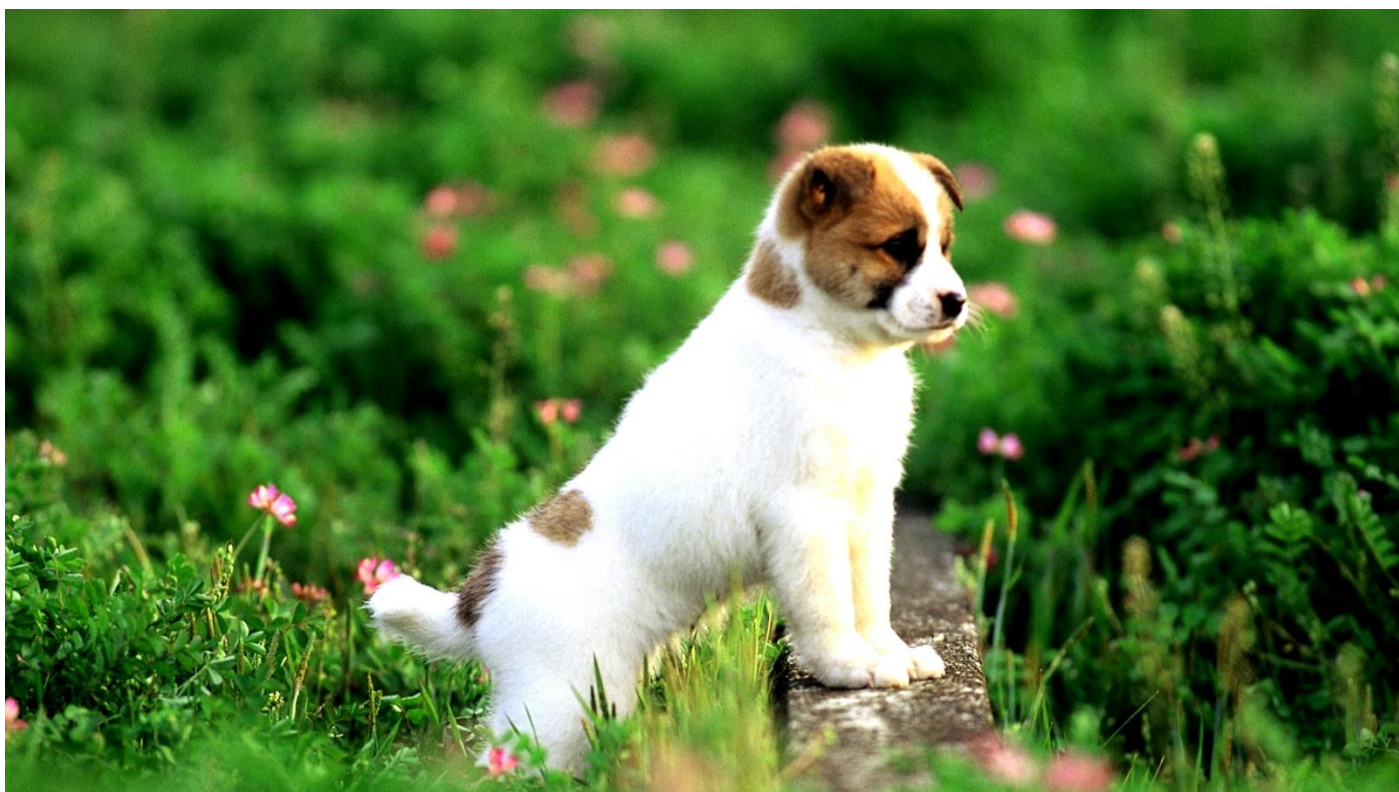
Both Images Loaded – File Decrypting



Comparison of Original and Encrypted Image



Original



Encrypted with 47KB GIF