PicSteg

Ben Kobiske & Andrew Yang

Dec 14, 2024

CONTENTS:

CHAPTER

ONE

USER GUIDE

1.1 How do I hide a message?

When you launch PicSteg, you'll be greeted by the main menu. From here you can select from a few options: message reading (r), message writing (w), and program exit (q).

```
Would you like to read or write to this image or exit (r/w/q)?
```

To write an image, you'll want to type 'w' (without the single quotes). You'll have to provide a path to an image to hide your message in and provide a message. Once you provide an output path, the process is complete.

```
Would you like to read or write to this image or exit (r/w/q)? w
Enter an image path: my_image_path.png
Enter a message to write: A VERY secret, hidden message. I'm hiding this one using a␣
↪tool named PicSteg!!!
Enter an output image path: my-output-image.png
```

1.2 How do I retrieve a message from an altered image?

It couldn't be simpler, type 'r' this time and enter a path to an image you embedded a secret message in. You'll see your message appear in the terminal.

```
Would you like to read or write to this image or exit (r/w/q)? r
Enter an image path: my_image_path.png
--Message Start--
A VERY secret, hidden message. I'm hiding this one using a tool named PicSteg!!!
--Message End--
```

CHAPTER

TWO

PICSTEG'S DESIGN AND DATA PIPELINES

PicSteg is designed to embed data within images, in a way that isn't noticable to the casual observer. To do so, PicSteg embeds data into the least significant bit of each color channel. We achieve a data density of 3 bits per pixel using our grouped bit striping technique and efficient reads on large images using our image length header.

Warning

PicSteg should not be treated as secure for important or covert communication, since it may be vulnurable to some statistical based detection methods. If you expect inspection by a skilled adversary, use a program that uses more complex (and harder to detect) embedding methods, like HSI color model embedding.

2.1 How is data encoded for grouped bit striping?

Let's use the following demonstration bitstream to demonstrate how data is seperated into color streams.

```
Hexadecimal / Binary / Decimal
0x1F1E7E / 0b000111110001111001111110 / 2,039,422
```

Now, to turn the above 3 bytes of binary data (displayed in Hexadecimal, Binary, and Decimal for your viewing pleasure) we can use alternate between the three colors (RGB) and produce a bytestream. Our result for the above example will come out as the below (we are alternating in RGB order).

```
Red: 0x1F / 0b00011111 / 31
Green: 0x1E / 0b00011110 / 30
Blue: 0x7E / 0b01111110 / 126
```

We've split up our colors, now what? How do we sneak this inside an image?

2.2 How do we embed our message?

We have our color data, let's get an example 8 image pixels with a color depth of 8 bits per pixel with three color channels. (We support PNG and JPEG, with JPEG lacking support for an alpha channel. Since it lacks support, we won't add 4 channel embedding support for PNG only and we'll just use a single embedding process for all supported formats).

```
HEX / Binary (8x 8-bit values)
R: 7E 59 40 F8 D9 28 59 38 / 01111110 01011001 01000000 11111000 11011001 00101000␣
↪01011001 00111000
G: 49 D8 9E 8A 78 98 D8 99 / 01001001 11011000 10011110 10001010 01111000 10011000␣
```

(continues on next page)

(continued from previous page)

```
↪11011000 10011001
B: 93 58 93 25 48 23 09 58 / 10010011 01011000 10010011 00100101 01001000 00100011␣
↪00001001 01011000
```

With our sample data, let's add it to the above pixel data. First, let's go into how LSB (Least Significant Bit) steganography works. We each 8 bit value and look for our least significant bit (this is the bit that affects the number's value the least when flipped or the rightmost bit) and change it to a 1 or 0 depending on the value of our message bit. Let's start with our Red values.

```
HEX / Binary (8x 8-bit values)
R: 7E 59 40 F8 D9 28 59 38 / 01111110 01011001 01000000 11111000 11011001 00101000␣
↪01011001 00111000
Least Significant Bits (LSBs) in the above binary: 01001010
Binary to Embed: 00011111
New R: 7E 58 40 F9 D9 29 59 39 / 01111110 01011000 01000000 11111001 11011001 00101001␣
↪01011001 00111001
```

We can follow the same process to embed our G (green) and B (blue) values as well. We get the following result once we're done.

```
Red Message: 0x1F / 0b00011111
Green Message: 0x1E / 0b00011110
Blue Message: 0x7E / 0b01111110
New R: 7E 58 40 F9 D9 29 59 39 / 01111110 01011000 01000000 11111001 11011001 00101001␣
↪01011001 00111001
New G: 48 D8 9E 8B 79 99 D9 98 / 01001000 11011000 10011110 10001011 01111001 10011001␣
↪11011001 10011000
New B: 92 59 93 25 49 23 09 58 / 10010010 01011001 10010011 00100101 01001001 00100011␣
↪00001001 01011000
```

To recover our plaintext message, we can read the LSB of each color channel of our image.

2.3 How do we know when to stop reading?

Images can be quite large and our message may not take up the entire image. We know how long our message is by taking the length of our message, during the embedding stage, and attaching it to the front of the message. Now, we can read the first 16 pixels and get our length (stored as a 32 bit unsigned integer), so we know exactly how much of the image to read.

CHAPTER

THREE

API DOCUMENTATION

string_serialize.**bytesToUint32**(*byte_string: bytearray*) → int

Combine a bytearray of length 4 into a 32-bit unsigned integer

Parameters
byte_string (*bytearray*) – An array to combine

Returns
a number representing the array content

Raises
AttributeError – if the array length is under 4 bytes

Return type
bytearray

string_serialize.**deserialize**(*rgb_bytes: tuple[bytearray(b''), bytearray(b''), bytearray(b'')]*) → str

Takes values pulled from RGB image channels and turns them back into a message string

Parameters
rgb_bytes (*tuple[bytearray(), bytearray(), bytearray()]*) – Bytes taken from color image channels

Returns
a string created from the UTF-8 (assumed) bytestream pulled from the image channels

Raises
UnicodeDecodeError – if the bytes provided are not valid UTF-8

Return type
str

string_serialize.**serialize**(*some_string: str*, *pixels: int*) → tuple[bytearray(b''), bytearray(b''), bytearray(b'')]

Takes a string and splits it into bytestreams for each color channel, checks that data can fit in pixels provided. The message may be corrupted due to error resolution (replace) in UTF-8 encoding, if non-UTF-8 encodable characters are present.

Parameters

- **some_string** (*str*) – a string to encode as a message
- **pixels** (*int*) – the total pixels in the target image

Returns
a bytearray for each color channel, for embedding into a target image

Raises
AttributeError – if the string length is larger than 2^32-1 (can't be fit in length header) or message is too long for image size

Return type
tuple[bytearray(), bytearray(), bytearray()]

string_serialize.**uint32ToBytes**(*number: int*) → bytearray

Split a 32-bit unsigned integer into a bytearray of length 4

Parameters
number (*int*) – Some number to split up

Returns
a byte array containing the number

Raises
AttributeError – if the number is too small (negative) or too large (over 2^32 - 1)

Return type
bytearray

image_embed.**bitsToByte**(*bits: list[int]*) → int

Turns a list of 1s and 0s into a byte of data

Parameters
bits (*tuple[bytearray(), bytearray(), bytearray()]*) – a list of 1s and 0s (length of 8)

Returns
an integer between 0 and 255

Raises
AttributeError – if the list's length is less than 8

Return type
int

image_embed.**embed**(*image: ImageFile*, *data: tuple[bytearray, bytearray, bytearray]*)

Takes an image and three bytestreams, embeds the bytestreams into the image

Parameters

- **image** (*ImageFile.ImageFile*) – an image to embed the data into
- **bits** (*tuple[bytearray(), bytearray(), bytearray()]*) – three bytestreams to embed into the Red, Green, and Blue color channels respectively

Returns
None

Return type
None

image_embed.**extract**(*image: ImageFile*) → tuple[bytearray, bytearray, bytearray]

Reads the embedded message from an image file, returns the bytestreams within the image

Parameters
image (*ImageFile.ImageFile*) – a list of 1s and 0s (length of 8)

Returns
the bytestreams found within the color channels of the image

Raises
AttributeError – if the header contains a size above the image's capacity

Return type
tuple[bytearray, bytearray, bytearray]

image_embed.**findNewValue**(*currentValue: int*, *embedValue: int*, *embedIndex: int*) → int

Takes the bit at the index provided in the embedValue integer and sets the LSB of currentValue to it, returns this value

Parameters

- **currentValue** (*int*) – a pixel value from an image
- **embedValue** (*int*) – a byte being embedded into an image
- **embedIndex** (*int*) – the index of the pixel within its pixelCluster

Returns
an updated pixel value, with data embedded

Raises

- **AttibuteError** – if the currentValue or embedValue are not between 0 and 255
- **AttibuteError** – if the embedIndex is not between 0 and 7

Return type
int

image_embed.**getEmbeddedValue**(*value: int*) → int

Returns the least significant bit of the value provided

Parameters
value (*int*) – an integer between 0 and 255

Returns
a 1 or a 0 (the LSB)

Raises
AttributeError – if value is not between 0 and 255

Return type
int

image_embed.**getMessageSize**(*image: ImageFile*) → int

Reads the message header within the image to determine the message size, returns it

Parameters
image (*ImageFile.ImageFile*) – an image containing an embedded message

Returns
a message size between 0 and 2^32 - 1

Return type
int

PYTHON MODULE INDEX

INDEX