StegaStamp: Invisible Hyperlinks in Physical Photographs
Supplement

1. StegaStamp Examples

See Figure 1 for additional examples of encoded images and their residuals.

2. Supplemental Videos

https://youtu.be/E8OqgNDBGO0
This video provides an overview of StegaStamps with example use cases and a condensed demonstration of in-the-wild results.

https://youtu.be/jpbRhOH3D9Y
This video is a compilation of multiple in-the-wild captures. The first set of clips visualizes the output bounding polygons along with the percentage of bits recovered correctly out of 100. We filter the output to only show detections where the bit accuracy is greater than 70 percent. We note that the messages are regularly recovered with greater than 90% accuracy when they are properly detected. The second set of clips demonstrates the used of BCH error correction [2] to robustly detect and correct recovered codes. The transmitted data consists of 56 message bits and 40 error correcting bits. When the accuracy is greater than 95% (fewer than 5 corrupted bits), the original 56-bit message can be recovered exactly. If too many bits are corrupted, the error correcting fails and we filter out the proposal. The video represents successfully decoded StegaStamps with green polygons. The decoded code is printed above the polygon. Note that for most real world applications, it is only necessary to recover the code in a single video frame to count it as successfully scanned.

3. Comparison Details

We compare our method to Baluja [1], HiDDeN [5], and LFM [3]. Baluja was designed to hide images within images, which differs from our task of hiding a bitstring within an image. To account for this, we convert our 100 bit message into a $10 \times 10$ grid of ones and zeros that is upscaled to the resolution of the cover image. During decoding we round the model output to 0 and 1 and take the mode within each upscaled block. As the original model was trained to hide natural images, we retrain the model from scratch to hide our bitstring grids.

HiDDeN was trained to hide 30 bit messages in $128 \times 128$ pixel images. We observed a significant drop in accuracy when we trained a model to hide 100 bit messages in $400 \times 400$ pixel images, therefore we report accuracy results on the $30 \times 30$ bit version.

LFM [3] was trained to encode 1024 bit messages as $4 \times 4$ pixel blocks in a $256 \times 256$ pixel image. To encode our 100 bit message, we allocated 9 blocks for each message bit (we therefore only use a $244 \times 244$ pixel subset of the image). We average and round the 9 block predictions to recover the message bit.

Each compared method encodes a different length message into a different size image. However, if we treat the mean bit recovery accuracy (first column in Table 1) as the crossover probability $p$ in a binary symmetric channel, we can use information theory to calculate the channel capacity

<table>
<thead>
<tr>
<th></th>
<th>Mean Acc. $\uparrow$</th>
<th>bits/MP $\uparrow$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baluja [1]</td>
<td>0.51</td>
<td>0.5</td>
</tr>
<tr>
<td>HiDDeN [5]</td>
<td>0.65</td>
<td>125</td>
</tr>
<tr>
<td>LFM [3] (printed)</td>
<td>0.61</td>
<td>287</td>
</tr>
<tr>
<td>LFM [3] (screen)</td>
<td>0.93</td>
<td>1109</td>
</tr>
<tr>
<td>None</td>
<td>0.49</td>
<td>0.1</td>
</tr>
<tr>
<td>Pixelwise</td>
<td>0.51</td>
<td>0.2</td>
</tr>
<tr>
<td>Spatial</td>
<td>0.89</td>
<td>318</td>
</tr>
<tr>
<td>All</td>
<td>0.99</td>
<td>571</td>
</tr>
</tbody>
</table>

Table 1: Quantitative comparison of other methods and our ablations. We show numbers in terms of fraction of bits correctly recovered (mean accuracy) as well as bits-per-megapixel (bits/MP). Higher is better for both metrics. The bits/MP metric normalizes the message length and image sizes between different methods. All methods except “LFM [3] (screen)” (cellphone camera/cellphone screen) are reported on the cellphone camera/consumer printer pipeline. We report LFM’s results in this additional case because it was explicitly designed for screen/camera transmission.
Figure 1: Additional examples of encoded images and their residuals.

$$C(p) = 1 - (-p \log_2 p - (1 - p) \log_2 (1 - p))$$  \hspace{1cm} (1)

If we divide $C(p)$ by the number of pixels $N_{\text{pix}}$ in the original image, we get the expected number of bits-per-pixel transmitted by that method. Multiplying $\frac{C(p)}{N_{\text{pix}}}$ by $10^6$ yields our bits-per-megapixel metric in the second column of Table 1.

4. Architecture Details

Network architectures for our encoder (Table 3) and decoder (Table 4). Our detector uses the BiSeNet [4] architecture.

<table>
<thead>
<tr>
<th></th>
<th>PSNR ↑</th>
<th>SSIM ↑</th>
<th>LPIPS ↓</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baluja [1]</td>
<td>24.61</td>
<td>0.926</td>
<td>0.256</td>
</tr>
<tr>
<td>HiDDeN [5] (native)</td>
<td>31.07</td>
<td>0.940</td>
<td>0.070</td>
</tr>
<tr>
<td>HiDDeN [5]</td>
<td>24.55</td>
<td>0.775</td>
<td>0.202</td>
</tr>
<tr>
<td>LFM [3]</td>
<td>20.89</td>
<td>0.910</td>
<td>0.315</td>
</tr>
<tr>
<td>Ours</td>
<td>27.25</td>
<td>0.927</td>
<td>0.194</td>
</tr>
</tbody>
</table>

Table 2: Quantitative comparison of encoded image quality, indicating how well hidden the message is. For HiDDeN [5] we show both the metrics for the original lower resolution (native $128 \times 128$) and upsampling to our compared resolution of $400 \times 400$ with bicubic interpolation. At full resolution, our method produces an encoded image most similar to the original in all metrics.
### Table 3: Our encoder network architecture.

<table>
<thead>
<tr>
<th>Layer</th>
<th>k</th>
<th>s</th>
<th>chns</th>
<th>in</th>
<th>out</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>inputs</td>
<td></td>
<td></td>
<td>6</td>
<td>1</td>
<td>1</td>
<td>image + secret</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>1</td>
<td>6/32</td>
<td>1</td>
<td>1</td>
<td>inputs</td>
</tr>
<tr>
<td>conv2</td>
<td>3</td>
<td>2</td>
<td>32/32</td>
<td>1</td>
<td>2</td>
<td>conv1</td>
</tr>
<tr>
<td>conv3</td>
<td>3</td>
<td>2</td>
<td>32/64</td>
<td>2</td>
<td>4</td>
<td>conv2</td>
</tr>
<tr>
<td>conv4</td>
<td>3</td>
<td>2</td>
<td>64/128</td>
<td>4</td>
<td>8</td>
<td>conv3</td>
</tr>
<tr>
<td>conv5</td>
<td>3</td>
<td>2</td>
<td>128/256</td>
<td>8</td>
<td>16</td>
<td>conv4</td>
</tr>
<tr>
<td>up6</td>
<td>2</td>
<td>1</td>
<td>256/128</td>
<td>16</td>
<td>8</td>
<td>upsample(conv5)</td>
</tr>
<tr>
<td>conv6</td>
<td>3</td>
<td>1</td>
<td>256/128</td>
<td>8</td>
<td>8</td>
<td>conv4 + up6</td>
</tr>
<tr>
<td>up7</td>
<td>2</td>
<td>1</td>
<td>128/64</td>
<td>8</td>
<td>4</td>
<td>upsample(conv6)</td>
</tr>
<tr>
<td>conv7</td>
<td>3</td>
<td>1</td>
<td>128/64</td>
<td>4</td>
<td>4</td>
<td>conv3 + up7</td>
</tr>
<tr>
<td>up8</td>
<td>2</td>
<td>1</td>
<td>64/32</td>
<td>4</td>
<td>2</td>
<td>upsample(conv7)</td>
</tr>
<tr>
<td>conv8</td>
<td>3</td>
<td>1</td>
<td>64/32</td>
<td>2</td>
<td>2</td>
<td>conv2 + up8</td>
</tr>
<tr>
<td>up9</td>
<td>2</td>
<td>1</td>
<td>32/32</td>
<td>2</td>
<td>1</td>
<td>upsample(conv8)</td>
</tr>
<tr>
<td>conv9</td>
<td>3</td>
<td>1</td>
<td>70/32</td>
<td>1</td>
<td>1</td>
<td>conv1 + up9 + inputs</td>
</tr>
<tr>
<td>conv10</td>
<td>3</td>
<td>1</td>
<td>32/32</td>
<td>1</td>
<td>1</td>
<td>conv9</td>
</tr>
<tr>
<td>residual</td>
<td>1</td>
<td>1</td>
<td>32/3</td>
<td>1</td>
<td>1</td>
<td>conv10</td>
</tr>
</tbody>
</table>

The kernel size, s is the stride, chns is the number of input and output channels for each layer, in and out are the accumulated stride for the input and output of each layer. input denotes the input of each layer with + meaning concatenation and “upsample” performing $2 \times$ nearest neighbor upsampling. A ReLU is applied after each layer except the last.

### Table 4: Our decoder network architecture.

<table>
<thead>
<tr>
<th>Layer</th>
<th>k</th>
<th>s</th>
<th>chns</th>
<th>in</th>
<th>out</th>
<th>input</th>
</tr>
</thead>
<tbody>
<tr>
<td>conv1</td>
<td>3</td>
<td>2</td>
<td>3/32</td>
<td>1</td>
<td>2</td>
<td>image</td>
</tr>
<tr>
<td>conv2</td>
<td>3</td>
<td>2</td>
<td>32/64</td>
<td>2</td>
<td>4</td>
<td>conv1</td>
</tr>
<tr>
<td>conv3</td>
<td>3</td>
<td>2</td>
<td>64/128</td>
<td>4</td>
<td>8</td>
<td>conv2</td>
</tr>
<tr>
<td>fc0</td>
<td></td>
<td></td>
<td>320000</td>
<td>4</td>
<td>4</td>
<td>flatten(conv3)</td>
</tr>
<tr>
<td>fc1</td>
<td></td>
<td></td>
<td>320000</td>
<td>8</td>
<td>8</td>
<td>fc0</td>
</tr>
<tr>
<td>fc2</td>
<td></td>
<td></td>
<td>128/6</td>
<td>16</td>
<td>16</td>
<td>fc1</td>
</tr>
<tr>
<td>image_warped</td>
<td>1</td>
<td>1</td>
<td>32/3</td>
<td>1</td>
<td>1</td>
<td>transf(image, fc2)</td>
</tr>
<tr>
<td>conv1</td>
<td>3</td>
<td>2</td>
<td>3/32</td>
<td>1</td>
<td>2</td>
<td>image_warped</td>
</tr>
<tr>
<td>conv2</td>
<td>3</td>
<td>1</td>
<td>32/32</td>
<td>2</td>
<td>2</td>
<td>conv1</td>
</tr>
<tr>
<td>conv3</td>
<td>3</td>
<td>2</td>
<td>32/64</td>
<td>2</td>
<td>4</td>
<td>conv2</td>
</tr>
<tr>
<td>conv4</td>
<td>3</td>
<td>1</td>
<td>64/64</td>
<td>4</td>
<td>4</td>
<td>conv3</td>
</tr>
<tr>
<td>conv5</td>
<td>3</td>
<td>2</td>
<td>64/64</td>
<td>4</td>
<td>4</td>
<td>conv4</td>
</tr>
<tr>
<td>conv6</td>
<td>3</td>
<td>2</td>
<td>64/128</td>
<td>8</td>
<td>16</td>
<td>conv5</td>
</tr>
<tr>
<td>conv7</td>
<td>3</td>
<td>2</td>
<td>128/128</td>
<td>16</td>
<td>32</td>
<td>conv6</td>
</tr>
<tr>
<td>fc0</td>
<td></td>
<td></td>
<td>20000</td>
<td>16</td>
<td>16</td>
<td>flatten(conv7)</td>
</tr>
<tr>
<td>fc1</td>
<td></td>
<td></td>
<td>20000</td>
<td>512</td>
<td>512</td>
<td>fc0</td>
</tr>
<tr>
<td>secret</td>
<td></td>
<td></td>
<td>512/100</td>
<td>1</td>
<td>1</td>
<td>fc1</td>
</tr>
</tbody>
</table>

The code and pretrained networks can be found at https://github.com/tancik/StegaStamp.

### References


