svgsteg.py

svgsteg.py implements steganographic embedding and decoding with SVG, an XML based vector image format published by the W3C. A typical SVG file consists of a series of tags representing objects like text, curved paths, polygons, and gradients. For example, a blue 300x100 rectangle in SVG would be:

```
<rect width="300" height="100" style="fill:rgb(0,0,255)>
```

Though SVG images can be hand coded, it is much more common for vector graphics tools like Adobe Illustrator and Inkscape to output them automatically. I made a couple of images myself in Illustrator to see what the typical output looked like. One thing I immediately noticed was how precise the numbers in the tags were compared to the tutorial examples at w3schools. For instance, the following tag describes the soft shadow under Tux in the image below.

```
<path transform="matrix(1.4177,0,0,0.414745,-38.7944,222.194)"
d="M 670.88202 1166.6423 A 203.89551 186.63016 0 1 1
263.091,1166.6423 A 203.89551 186.63016 0 1 1 670.88202
1166.6423 z" id="path175" style="fill:url
(#radialGradient1399);stroke:none;stroke-width:1pt;stroke-
linecap:butt;stroke-linejoin:miter" />
```

After playing around with the numbers in this tag's "d" attribute, it became clear that the floating point values were far more precise than necessary. I could change the lowest digit of any path point without visibly affecting the rendered image.

Based on this observation, I used the least significant digits (LSDs) in the floating point numbers in <path>, <linearGradient>, and <radialGradient> tags as a steganographic embedding medium. These tags were chosen by trial and error. To represent a 1, svgsteg.py changes the LSD to a random odd number; to represent a 0, a random even number. The randomness in these numbers exists to prevent the algorithm from introducing any obvious patterns like "everything ends in a 0 or a 1".

To randomly spread the message through the image, I implemented a variant of the permutative straddling Westfeld describes in the F5 algorithm. Instead of DCT coefficients, svgsteg.py shuffles a list of embedding "slots" representing all of the floating point numbers in the image marked safe for embedding. Like F5, the shuffling permutation is derived from a user supplied stego-key.

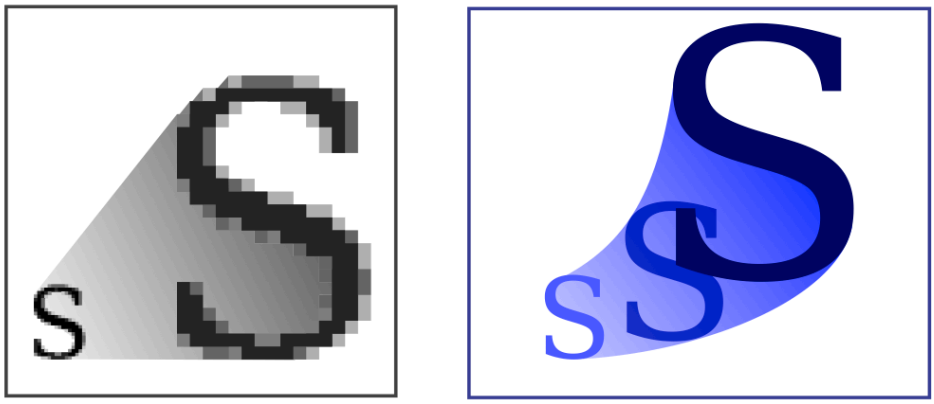
svgsteg.py has 3 different usage modes:

```
svgsteg.py -embed msg.txt cover.svg stegokey
svgsteg.py -extract stego-object.svg stegokey
svgsteg.py -capacity cover.svg
```

Embed mode embeds msg.txt into cover.svg with key stegokey, then dumps the resulting XML document to standard output. Extract mode extracts a message embedded in stego-obj.svg with key stegokey. Capacity mode informs the user how many ASCII characters can be embedded in cover.svg.

After embedding a variety of messages in SVGs of varying complexity, I came to the following conclusions:

- SVG steganography has excellent image quality. There was no visible distortion in the stego-object.
- Embedding capacity was not impressive. The following, rather complex image from Wikipedia could only hold 858 embedded characters:



As far as I've been able to find, no research has been done on the statistical properties of SVG tags. For short messages, SVG embedding may very well be a secure form of steganography.