Ada Feng, Drew Shippey, and Ely Merenstein
MA340/CP341: Cryptography: Theory and Practice
Professors Ben Ylvisaker and Stefan Erickson
5/10/19

Steganography, Watermarking, and Fingerprinting

Historically, steganography has involved hiding a message in a seemingly innocuous item to transmit the message without revealing its presence. Once a message is discovered, there is nothing stopping the adversary from learning the message (as opposed to classical cryptography, in which that the adversary knows that there is an encrypted message but must put effort into revealing it). Gustavus Simmons (1983) describes one theoretical application of steganography — the "Prisoner's Problem" — wherein two prisoners, Alice and Bob, are colluding to formulate a prison break, and they are passing information hidden in regular objects. If the guard ever catches them, they will be put into solitary confinement and doomed, therefore, it is in the prisoners' interests to decide carefully how much of each object is modified such that they can slip in as much information as possible without making the modifications obvious (1983). Similar decisions must be made in digital steganography—how many bits of an image or audio track can be modified before the alteration is plain to see or hear?

Steganography has recently attracted much interest as a means to protect media copyrights or to circumvent government censorship of private messages. Digital artworks are often compressed when transmitted across the internet, and most customers are capable of making modifications to them, so robustness has become the new focus of steganography. Therefore, it is necessary to modify more bits in order to withstand added "noise," and it is important to determine to what degree one can still retrieve data after the message-embedded objects are modified. Cryptographers also have developed methods of watermarking and fingerprinting to invisibly mark the digital works either with the same marks for all copies or with distinct marks to track leakage identity.

Our programming work demonstrates experimentation with several of these matters. One of the runnable programs we've submitted is found in the file "Steganography.py." This program implements basic image steganography: there's functionality to encode a chosen message into a chosen image, and there's functionality to decode the message stored in a chosen image. In addition, this program implements an added security feature that prompts what we believe to be an interesting discussion. This feature consists of a pseudorandom sequence of integers between 0 and 2 (inclusive) that represent the colors red, green, and blue. For each bit of the message to be encoded, the next pseudorandom value determines which color value of the pixel to be modified will actually be modified. The added security (and necessary determinism of the algorithm) involves the seed used for the random number generator. The user must set the seed to be used (they may choose any number) prior to running an encoding or decoding algorithm. Using the same seed to encode and decode enables one to successfully decode the message stored in an image; keeping the seed secret, meanwhile, prevents even attackers who are aware of the presence of steganography from decoding the message.

The "interesting discussion" that comes of this feature is at least two-fold. For one thing, using a secret value (and therefore assuming an attacker is aware of the transmission of a message) trivializes the whole use of steganography, since there are other encryption methods that are widely accepted as more secure for applications in which attackers are aware of the transmission of a message. Furthermore, randomly deciding which one of every three values to change means that two thirds of the image file, which could encode message bits, is not being used. Therefore, it's fair to say that such a security feature would likely never be used for steganography in actual practice (though this could, doubtless, be debated further). Instead, we provide it as a simply experimental matter in our process of learning about steganography.

To run Steganography.py, ensure that you've saved any images you'll want to use for encoding/decoding in the same directory as the program (of course, the encoding algorithm outputs an image file to be used for decoding into that directory).

In Steganography.py, the algorithm was designed to make changes unrecognizable to the human eye. In Distortion.py, meanwhile, we allow the user to experiment with exactly how an image is modified with much more flexibility. In this program, the computer takes care of generating a random message that will use all of the available pixels for encoding. The user then selects which color values to modify in the image, as well as how many of the least significant bits of each color value to modify. As expected, the modifications get exponentially more recognizable as the user increases the number of bits to be modified; 3 bits (a maximum change of 1/32 of the color value) is essentially not recognizable; 4 bits (maximum 1/16 change) creates a visible but somewhat subtle difference; 8 bits (maximum complete change) leaves the initial image completely unrecognizable — the pixel values simply represent the randomly-generated message, encoded in color.

To run Distortion.py, ensure that an image file called "cat.png" is stored in the same directory as the program. Note: this program may take a while to complete, but it does indeed complete; it should be done in less than sixty seconds.

Finally, in fingerprint.py, we've implemented a Fingerprinting algorithm. The user is able to issue a copy of the cat image and if that copy is leaked, the user can download that copy and input it into our program to find out who leaked it. It is a pretty naive program that depends on the people leaking the program not knowing the copy is fingerprinted, or how it is fingerprinted, since otherwise it is easy to just change the pixels containing the identifying message, and the user wouldn't be able to map the identity correctly. We used a hash value of the order of the copy issued so that we won't easily have two values that are the same that map to different identities, and so that if the leaker actively tries to find if the copy had been fingerprinted, they will just find the hash values that look innocuous.

To run fingerprint.py, ensure that an image file called "cat.png" is stored in the same directory as the program.