CITI Technical Report 01-11

Detecting Steganographic Content on the Internet

Niels Provos *Peter Honeyman*
{provos,honey}@citi.umich.edu

Abstract

Steganography is used to hide the occurrence of communication. Recent suggestions in US newspapers indicate that terrorists use steganography to communicate in secret with their accomplices. In particular, images on the Internet were mentioned as the communication medium. While the newspaper articles sounded very dire, none substantiated these rumors.

To determine whether there is steganographic content on the Internet, this paper presents a detection framework that includes tools to retrieve images from the world wide web and automatically detect whether they might contain steganographic content. To ascertain that hidden messages exist in images, the detection framework includes a distributed computing framework for launching dictionary attacks hosted on a cluster of loosely coupled workstations. We have analyzed two million images downloaded from eBay auctions but have not been able to find a single hidden message.

August 31, 2001

Center for Information Technology Integration
University of Michigan
535 West William Street
Ann Arbor, MI 48103-4943

Detecting Steganographic Content on the Internet*

Niels Provos Peter Honeyman
Center for Information Technology Integration
University of Michigan

1 Introduction

Steganography is the art and science of hiding the fact that communication is taking place. Steganographic systems can hide messages inside of images or other digital objects. To a casual observer inspecting these images, the messages are invisible.

In February 2000, *USA Today* reported that terrorists are using steganography to hide their communication from law enforcement [4]. According to them, messages are being hidden in images posted to Internet auction sides like eBay or Amazon. The article lacked any technical information that would allow a reader to verify these claims. Nonetheless, the article was echoed by a number of other news sources.

To assess the claim that steganographic content is regularly posted to the Internet, we need a way to detect steganographic content in images automatically. This paper presents a steganography detection framework that begins with a web crawler that downloads JPEG images from the Internet. Using statistical analysis, a subset of images likely to contain steganographic content is identified. The analysis is statistical, *i.e.* there is no guarantee that an identified image really contains a hidden message, so we also describe a distributed computing framework that launches a dictionary attack hosted on a cluster of loosely-coupled workstations to reveal any hidden content.

We discuss the results from analyzing two million images downloaded from eBay auctions. So far we have not been able to find a single message.

The remainder of this paper is organized as follows. In Section 2, we give a brief background of steganography in general. Section 3 explains how to hide information in JPEG [15] images. Section 4 presents statistical test capable of detecting steganographic content. In Section 5, we give an overview of existing steganographic systems and describe how to detect them. The detection framework is presented in Section 6. We discuss our results and related work in Sections 7 and 8. We conclude in Section 9.

*This research was supported in part by DARPA grant number F30602-99-1-0527.

2 Steganography Background

The term "Information Hiding" relates to both watermarking and steganography. Watermarking usually refers to methods that hide information in a data object so that the information is robust to modifications. That means, it should be impossible to remove a watermark without degrading the quality of the data object.

On the other hand, steganography refers to hidden information that is fragile. Modifications to the cover medium may destroy it.

Watermarking and steganography differ in another important way: while steganographic information must never be apparent to a viewer unaware of its presence, this feature is optional for a watermark.

The security of a classical steganographic system relies on the secrecy of the encoding system. Once the encoding system is known, the steganographic system is defeated. A famous example of a classical system is that of a Roman general who shaved the head of a slave and tattooed a hidden message on it. After the hair had grown back, the slave was sent to deliver the message [3]. While such a system might work once, the moment that it is known, it is simple to shave the heads of all people passing by to check for hidden messages.

Other encoding systems might use the last word in every sentence of a letter or the least significant bits in an image.

However, modern steganography should be detectable only if secret information is known, namely, a secret key. This is very similar to "Kerckhoffs' Principle" in cryptography, which holds that the security of a cryptographic system should rely only on the key material [5].

Because of their invasive nature, steganographic systems leave detectable traces within a medium's characteristics. This allows an eavesdropper to detect modified media, revealing that secret communication is taking place. Although the secret content is not exposed, its existence is revealed, which defeats the main purpose of steganography.

In general, the information hiding process consists of the following steps:

1. Identification of redundant bits in a cover medium. Redundant bits are those bits that can be modified without degrading the quality of the cover medium.

2. Selection of a subset of the redundant bits to be replaced with data from a secret message. The stego medium is created by replacing the selected redundant bits with message bits.

The modification of redundant bits can change the statistical properties of the cover medium. As a result, statistical analysis may reveal the hidden content [11, 16]. In Section 4, we explain in detail how this is possible.

3 Information Hiding in JPEG Images

JPEG images [15] are commonly used on Internet web sites. This section briefly explains the JPEG format and how it can be used for information hiding.

The JPEG image format uses a discrete cosine transform (DCT) to transform successive 8×8-pixel blocks of the image into 64 DCT coefficients each. The least-significant bits of the quantized DCT coefficients are used as redundant bits into which the hidden message is embedded.

In some image formats, *e.g.* GIF, the visual structure of an image exists to some degree in all bit-layers of the image. Steganographic systems that modify least-significant bits of these image formats are often susceptible to visual attacks [16].

This is not true for the JPEG format. The modification of a single DCT coefficient affects all 64 image pixels. For that reason, there are no known visual attacks against the JPEG image format.

Figure 1 shows two images with a resolution of 800×600 and 24-bit color depth. The uncompressed original image has a size of almost 12 Mb, while the two JPEG images shown are about 0.3 Mb. The one to the left is unmodified. The one to the right contains the first chapter of Lewis Carroll's "The Hunting of the Snark." After compression, the chapter has a size of about $14,700$ bits. It is not possible for the human eye to find a visual difference between the two of them.

4 Statistical Analysis

Statistical tests can reveal if an image has been modified by steganography by testing whether an image's statistical properties deviate from a norm. Some tests are independent of the data format and just measure the entropy of the redundant data.

The simplest test measures the correlation towards one. A more sophisticated one is Ueli Maurer's "Universal Statistical Test for Random Bit Generators" [7]. We expect images with hidden data to have a higher entropy than those without.

These simple tests are not able to decide automatically if an image contains a hidden message. Westfeld and Pfitzmann have observed that embedding encrypted data into a GIF image changes the histogram of its color frequencies [16]. One property of encrypted data is that the one and the zero bit are equally likely. When using the least-significant bit method to embed encrypted data into an image that contains the color two more often than the color three, the color two is changed more often to the color three than the other way around. As a result, the difference in color frequency between two and three has been reduced by the embedding.

The same is true for JPEG images. Instead of measuring the color frequencies, we analyze the frequency of the DCT coefficients. Figure 2 shows an example where embedding a hidden messages causes noticeable differences to the DCT coefficient histogram.

We use a χ^2-test to determine whether an image shows distortion from embedding hidden data. Because the test uses only the stego medium, the expected distribution y_i^* for the χ^2-test has to be computed from the image. Let n_i be the frequency of DCT coefficients in the image. We assume that an image with hidden data embedded has similar frequency for adjacent DCT coefficients. As a result, we can take the arithmetic mean,

$$y_i^* = \frac{n_{2i} + n_{2i+1}}{2},$$

Figure 1: The image on the left is the unmodified original, but the image on the right has the first chapter of the "Hunting of the Snark" embedded into it. There are no visual differences to the human eye.

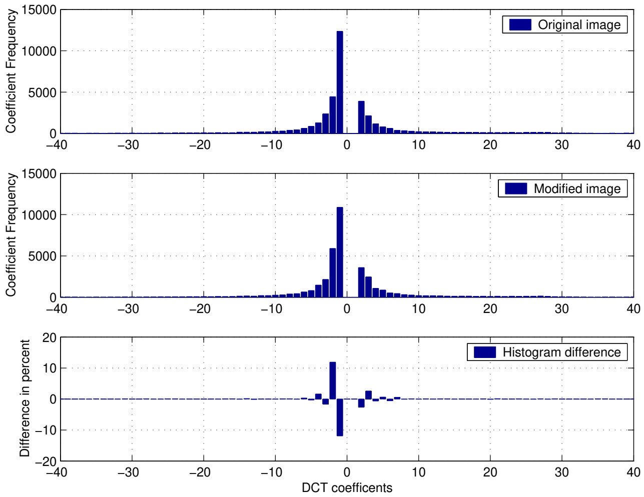

Figure 2: Embedding a hidden message causes noticeable changes to the histogram of DCT coefficients.

to determine the expected distribution. The expected distribution is compared against the observed distribution

$$y_i = n_{2i}.$$

The χ^2 value for the difference between the distributions is given as

$$\chi^2 = \sum_{i=1}^{\nu+1} \frac{(y_i - y_i^*)^2}{y_i^*},$$

where ν are the degrees of freedom, that is, the number of different categories in the histogram minus one.

The probability of embedding p is then given by the complement of the cumulative distribution function,

$$p = 1 - \int_0^{\chi^2} \frac{t^{(\nu-2)/2} e^{-t/2}}{2^{\nu/2}\Gamma(\nu/2)} dt,$$

where Γ is the Euler Gamma function.

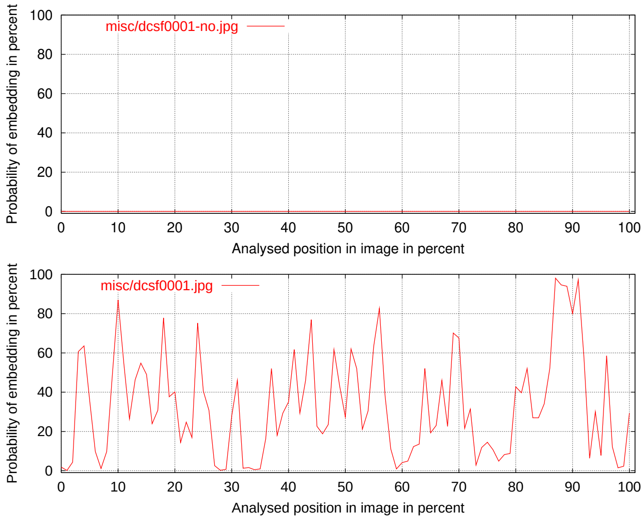

Figure 3: The probability of embedding calculated for different areas of an image. The upper graph shows the results for an unmodified image, the lower graph shows the results for an image with steganographic content.

We can compute the probability of embedding for different parts of an image. The selection depends on what steganographic system we try to detect. For an image that does not contain any hidden information, we expect the probability of embedding to be zero everywhere. Figure 3 shows the embedding probability for an image without steganographic content and for an image that has content hidden in it.

5 Steganographic Systems in Use

In this section, we present several steganographic systems that embed hidden messages into JPEG images. We show that the statistical distortions depend on the steganographic system that inserted the message into the image. Because the distortions are characteristic for each system, we develop signatures that allow us to identify which system has been used.

There are three popular steganographic systems available on the Internet that hide information in JPEG images:

- JSteg, JSteg-Shell
- JPHide
- OutGuess

All of these systems use some form of least-significant bit embedding and are detectable by statistical analysis except the latest release of OutGuess [9]. In the following, we present the specific characteristics of these systems and show how to detect them.

5.1 JSteg and JSteg-Shell

JSteg is an addition by Derek Upham to the Independent JPEG Group's JPEG Software library. The DCT coefficients are modified continuously from the beginning of the image. JSteg does not support encryption and has no random bit selection.

The data of the message is prepended with a variable size header. The first five bits of the header express the size of the length field in bits. The following bits contain the length field that expresses the size of the embedded content.

Figure 4 shows the result of the χ^2-test for an image that contains information hidden with JSteg. In this case, the first chapter of "The Hunting of the Snark" has been bzip2 compressed before the embedding. The low probability at the beginning of the graph is caused by the dictionary at the beginning of a bzip2 compressed file. The dictionary does not look like encrypted data and is not detected by the test.

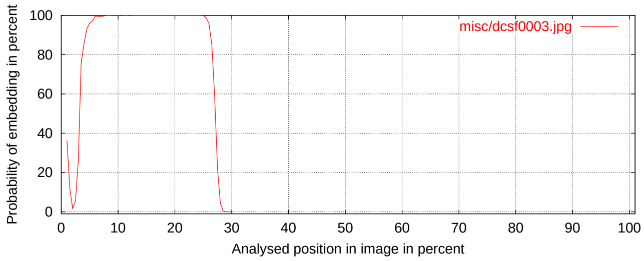

Figure 4: An image containing a message hidden with JSteg shows a high probability of embedding at the beginning of the image. It flattens to zero, when the test reaches the unmodified part of the DCT coefficients.

JSteg-Shell is a Windows user interface to JSteg. It has been developed by Korejwa and supports encryption and compression of the content before embedding the data with JSteg. JSteg-Shell uses the stream cipher RC4 [13] for encryption. However, the RC4 key space is restricted to 40 bits.

When encryption is being employed, we expect the probability of embedding to be high at the beginning of the image. There should be no exception.

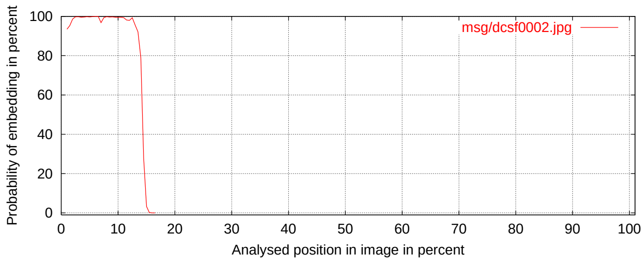

Figure 5: Using JSteg-Shell with RC4 encryption causes the probability of embedding to be high for all embedded data.

An example of JSteg-Shell is shown in Figure 5. Just observing the graph allows us to determine the size of the embedded message. We show later how this can help to improve the automatic detection of steganographic content.

5.2 JPHide

JPHide is a steganographic system by Allan Latham. There are two versions: 0.3 and 0.5. Version 0.5 supports additional compression of the hidden message. As a result, they use slightly different headers to store embedding information. Before the content is embedded, it is Blowfish [12] encrypted with a user-supplied pass phrase.

Because the DCT coefficients are not selected continuously from the beginning, JPHide is slightly more difficult to detect. The program uses a fixed table that determines which coefficient to modify next. The coefficients are selected by the table in such a way that coefficients that are likely to be numerically high are used first. A pseudo-random number generator determines if coefficients are skipped. The probability of skipping bits depends on the length of the hidden message and how many bits have been embedded already.

JPHide not only modifies the least-significant bits of the DCT coefficients, it can also switch to a mode where the second-least-significant bits are modified.

Figure 6: JPHide has a signature similar to JSteg. The major difference is the order in which the DCT coefficients are modified.

Figure 6 shows the probability of embedding for an image containing information hidden with JPHide. Because JPHide can skip DCT coefficients, the probability is not as high as with JSteg.

5.3 Outguess

OutGuess is a steganographic system available as UNIX source code. There are two released versions: OutGuess 0.13b, which is vulnerable to statistical analysis, and OutGuess 0.2, which includes the ability to preserve statistical properties [11] and can not be detected by the statistical tests used in this paper.

OutGuess is different from the systems described in the previous sections in that its chooses the DCT coefficients with a pseudo-random number generator. A user-supplied pass phrase initializes a stream cipher and a pseudo-random number generator, both based on RC4. The stream cipher is used to encrypt the content.

Because the modifications are distributed randomly over the DCT coefficients, the χ^2-test can not be applied on a continuously increasing sample of the image. Instead, we slide the position where we take the samples across the image.

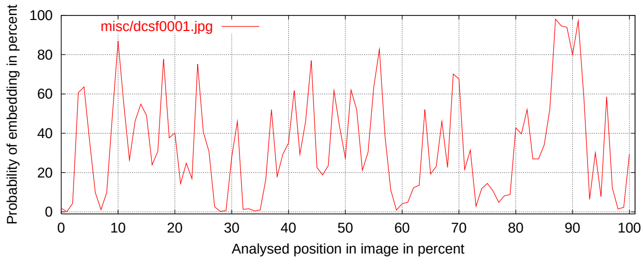

Figure 7: OutGuess 0.13b is more difficult to detect. Due to the random selection of bits, there is no clear signature.

For OutGuess 0.13b, we do not find any clear signatures. Figure 7 shows the probability of embedding for a sample image. The spikes indicate areas in the image where modifications to coefficients cause departures from the expected DCT coefficient frequency.

6 Detection Framework

In the previous section, we presented detection signatures that allow us to find hidden messages and determine which steganographic system was used to embed them. In the next section, we present "Stegdetect," an automated utility to analyse JPEG images for steganographic content.

6.1 Stegdetect

Stegdetect detects images that have content hidden with JSteg, JPHide and OutGuess 0.13b. For each system that we want to detect, we select the DCT coefficients in the order that they are modified and apply a χ^2-test.

```
misc/0003-wonder-2.jpg : jphide(*)
misc/dscf0001.jpg : outguess(old)(***)
misc/dscf0002.jpg : negative
misc/dscf0003.jpg : jsteg(***)
```

Figure 8: The output from Stegdetect contains an estimate of the detection confidence.

The output from Stegdetect lists the steganographic systems found in each image or "negative" if no steganographic content could be detected. Stegdetect expresses the level of confidence of the detection with one to three stars. Figure 8 shows some sample output.

6.1.1 JSteg Detection

Detection of content hidden with JSteg is similar to the approach outlined by Westfeld and Pfitzmann [16].

JSteg does not modify the DCT coefficients zero and one. For that reason, they are ignored in the χ^2-test. We sample the DCT coefficients starting from the beginning of the image and compute the probability of embedding. This process is repeated with increasing sample size until all DCT coefficients are contained in the sample. As a performance optimization, we stop computing the probability of embedding once it falls below a certain threshold.

To improve the detection accuracy, we estimate the size of the hidden content from the calculated graph and compare it with the size stored in the JSteg embedding header as described in Section 5.1.

6.1.2 JPHide Detection

Because JPHide modifies the DCT coefficients in a fixed order determined by a table, we rearrange the coefficients in that order before computing the probability of embedding. However, there are two exceptions that influence the detection.

JPHide modifies the DCT coefficients -1, 0 and 1 in a special way. As a result, the modifications to these coefficients can not be detected by the χ^2-test. However, simply ignoring these coefficients still allows us to detect content embedded with JPHide. We also ignore modifications to the second-least-significant bits, which are not as frequent as modifications to the least-significant bits.

Similar to JSteg, we stop computing the probability of embedding once it falls below a certain threshold.

6.1.3 OutGuess Detection

Detecting content embedded with OutGuess 0.13b is complicated by the fact that the coefficients are selected pseudo-randomly, there is no fixed order in which to apply the χ^2-test. However, Provos has shown that the χ^2-test can be extended to detect content hidden with OutGuess 0.13b [11].

Instead of increasing the sample size and applying the test at a constant position, we use a constant sample size but slide the position where the samples are taken over the entire range of the image.

The test starts at the beginning of the image, and the position is incremented by one percent for every application of the χ^2-test. The extended test does not react to an unmodified image, but detects the embedding in some areas of the stego image.

To find an appropriate sample size, we choose an expected distribution for the extended χ^2-test that should cause a negative test result. Instead of calculating the arithmetic mean of coefficients and their adjacent ones, we take the arithmetic mean of two unrelated coefficients,

$$y_i^* = \frac{n_{2i-1} + n_{2i}}{2}.$$

A binary search on the sample size is used to find a value for which the extended χ^2-test does not show a correlation to the expected distribution derived from unrelated coefficients.

6.1.4 Stegdetect Performance

In this Section, we analyse the performance of Stegdetect on a 333 MHz Celeron processor by measuring the time it takes to process a few hundred JPEG files. The result is the average number of kilobytes that can be processed per second (KBps).

We test the performance separately for each steganographic system, and then measure the performance for all tests in concert.

Test	Speed
JSteg	356 KBps
JPHide	200 KBps
OutGuess 0.13b	227 KBps
All tests	127 KBps

Figure 9: Stegdetect performance on a 333 MHz Celeron processor.

The results are displayed in Figure 9. As expected, the JSteg test is the fastest and detection of JPHide and OutGuess 0.13b are about the same speed.

Given the results for the separate tests, we would expect the combined speed for all tests to be about 80 KBps. However, the speed is higher because the tests for JPHide and Outguess are skipped if JSteg has been detected.

To calibrate the detection sensitivity of Stegdetect, we tested it on about $1,500$ images taken with a Fuji

MX-1700 digital camera. The results are shown in Figure 10. For images of this quality, we do not find any false positives.

The percentage of false negatives depends on the steganographic system and the size of the embedded message. The smaller the message, the harder it is to detect by statistical means. Stegdetect is very reliable in finding images that have content embedded with JSteg. For our sample images, we found only around 2% false negatives. For JPHide, between 15% and 60% were false negatives. JPHide 0.5 is more difficult to detect because it compresses the content before embedding. The rate of false negatives for OutGuess 0.13b is around 60%. The false negative rate is quite high. However, this is preferable to a high false positive rate, as we will explain in the next Section.

Test	False Negatives
JSteg	2%
JPHide	15% – 60%
OutGuess 0.13b	60%

Figure 10: Percentage of false negatives for a set of sample images.

6.2 Finding Images

Now that we can automatically test for steganographic content, we are ready to search for images that might have hidden messages embedded. The obvious locations to look for images are web sites on the Internet. A web crawler that finds JPEG images can supply Stegdetect with enough data.

Unfortunately, there were no open-source, image capable web crawlers available when we started our research, so we added the capability to save images to existing web crawlers, like larbin or the web consortium's web robot. However, none of them were stable enough to crawl large web sites reliably.

So we wrote "Crawl", a simple but efficient web crawler that saves JPEG images it encounters on web pages. Using "libevent" [10], a library for asynchronous event notification, Crawl is implemented in fewer than 5,000 lines of C source code.

Crawl performs a depth-first search and has the following features:

- Images and web pages can be matched against regular expressions. A match can be used to include or exclude web pages in the search.
- Minimum and maximum image size can be specified. This allows us to exclude images that are too small to contain hidden messages. We restricted our search to images that were larger than 20 KByte but smaller than 400 KByte.
- DNS requests are synchronous but cached. Synchronous DNS queries can be a major performance penalty because they cause the crawler to block and not to make progress on any other outstanding network connections. The effects are mitigated by caching positive and negative query results.

```
HEAD http://img.andale.com/635/monitor_lo.jpg
HEAD http://img.andale.com/635/hi.jpg
GET http://www.cities.com/a_ports/graphone.jpg
GET http://img.andale.com/635/scope_lo.jpg
Terminated with 3479 saved urls.
448684 GET for body 2861924 Kbytes
436084 HEAD for header 271287 Kbytes
   9.172 Requests/sec
```

Figure 11: The output from Crawl is used as input for Stegdetect.

At this writing, we have downloaded more than two million images linked to eBay auctions. To automate the detection, Crawl uses "stdout" to report successfully retrieved images; see Figure 11.

Because Stegdetect can accept images from "stdin", we connect Crawl to Stegdetect via a pipe to automate the detection process. After processing the two million images with Stegdetect, we find that over 1% of all images seem to contain hidden content. JPHide is detected the most; see Figure 12.

Test	False Positives
JSteg	0.003%
JPHide	1%
OutGuess 0.13b	0.1%

Figure 12: Percentage of (false) positives for images obtained from the Internet.

Most of these are likely to be false positives. Axelsson applies the "Base-Rate Fallacy" to intrusion detection systems and shows that a high percentage of false positives has a significant impact on the efficiency of such a system [1]. The situation is very similar for Stegdetect. It is safe to assume, that the percentage of images containing steganographic content is low in comparison to the percentage of false positives. As a result, the "true positive" rate, *i.e.* the probability that an image detected by Stegdetect really has steganographic content, is influenced mostly by the false positive rate.

We notice that there are special classes of images for which Stegdetect falsely indicates hidden content. An example of a false positive is shown in Figure 13. Stegdetect indicates that content has been hidden by JSteg. However, when analyzing the probability of embedding displayed next to the drawing, we do not see a plateau at the beginning, as we would expect had encrypted data been embedded.

We find similar false positives when trying to detect content hidden with OutGuess. Images with monotone backgrounds like the painting in Figure 14 are more likely to be false positive. When analyzing the graph, we see only a few high probability spikes. If there were hidden content, we would expect to find more areas in the image where the extended χ^2-test shows a positive result.

That Stegdetect finds so many images that seem to have content hidden with JPHide does not indicate that there are many images that really contain hidden content. Instead, it means that the detection functions for JPHide need to be improved to be more accurate. Furthermore, many images downloaded from the Internet are of very low quality, while the images that were used to calibrate Stegdetect are of higher quality, because they come directly from a digital camera.

6.3 Verifying Hidden Content

The statistical tests used to find steganographic content in images indicate nothing more than a likelihood that content has been embedded. Because of that, Stegdetect can not guarantee the existence of a hidden message.

To verify that the detected images have hidden content, it is necessary to launch a dictionary attack against the JPEG files. Stegbreak does just that for content hidden with JSteg-Shell, JPHide or Outguess 0.13b.

Because all the presented steganographic systems hide content based on a user supplied password, an attacker can try to guess the password to determine what content has been hidden. Instead of trying all possible passwords, it is much faster to try only words from dictionary, *i.e.* a dictionary attack [8].

For a dictionary attack to work, it is necessary that the user of the steganographic system selects a weak password, *i.e.* he selects the password from a small subset of the full password space.

Key attacks on cryptographic systems often have the benefit that properties of the underlying plaintext are known to the attacker. Given these properties, it is possible to verify statistically if the correct decryption key has been found [14]. All the steganographic systems presented in this paper embed header information in addition to a message into the images. The header information contains, among other things, the length of the hidden message. We can use this information to verify the correctness of the guessed password.

6.3.1 JPHide Header Information

Length bits 23-16	Length bits 15-8	Length bits 7-0	IV 1
IV 2	IV 3	IV 4	IV 5

Figure 15: Header information for JPHide 0.3.

JPHide 0.3 embeds a 64-bit header. The first 24 bits include the length of the hidden message in bytes. The other 40 bits are obtained from encrypting the first eight DCT coefficients with Blowfish. The Blowfish key schedule is initialized with the guessed password. JPHide takes the first eight DCT coefficients, reduces them modulo 256 and then concatenates to get a 64-bit block. This block is then encrypted, and the first 3 bytes are overwritten with the length information. The result is stored as header in the image; see Figure 15.

The dictionary attack uses the 40-bit IV as a verifier. Additionally, we can check if the encoded length fits in the image.

Compressed length bits 23-0			Mode
Orig. Len. bits 23-16	IV 1	IV 2	IV 3
Orig. Len. bits 15-8	Compressed length bits 15-0		Orig. Len bits 7-0
IV 4	IV 5	IV 6	IV 7

Figure 16: Header information for JPHide 0.5.

The header for JPHide 0.5 is twice as long as for JPHide 0.3; because JPHide 0.5 compresses the message before embedding, the header contains both the compressed and the original length of the message. With the increased header length, we get a 56-bit verifier. The IV is obtained by encrypting the first 16 DCT coefficients, and is then overwritten with the length information. In addition, to the 56 bits, we also get 16 more bits to verify our guess because parts of the compressed length have been duplicated in the header; see Figure 16.

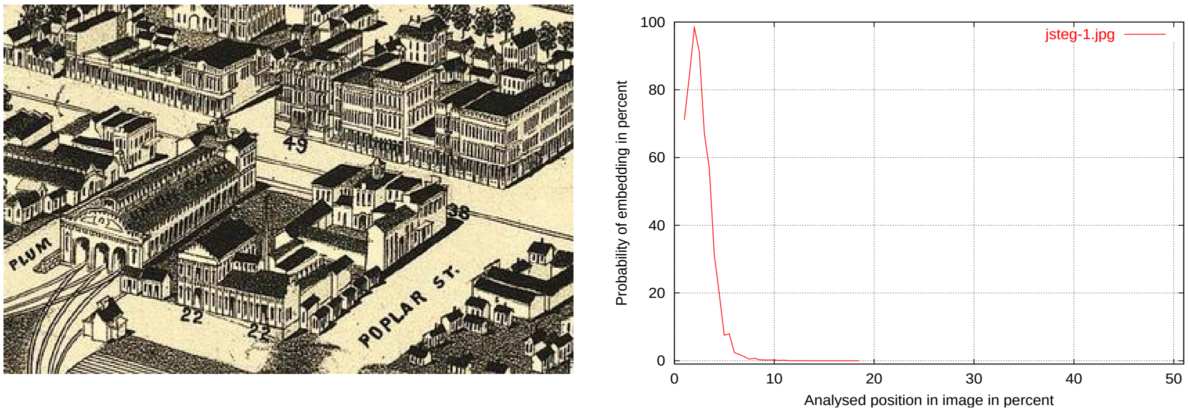

Figure 13: Stegdetect indicates that this drawings seems to have content hidden with JSteg.

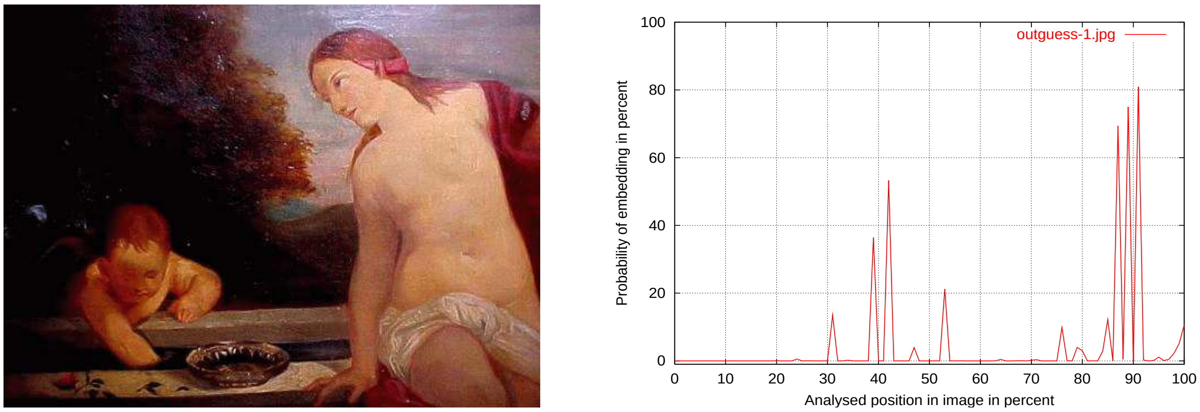

Figure 14: Stegdetect indicates that this painting seems to have content hidden with Outguess 0.13b.

Another difference between version 0.3 and 0.5 is a change in key schedule computation. In version 0.5, the Blowfish key schedule depends on the first eight DCT coefficients. As a result, the Blowfish key schedule has to be recomputed for images that differ in those DCT coefficients. This causes a marked slowdown in Stegbreak.

6.3.2 JSteg-Shell Header Information

JSteg-Shell is very simple. Because, it is just a user interface to JSteg, it does not encrypt the length of the embedded message. Instead it adds a signature at the end of the message. The signature is either "korejwa", "cMk4" or "cMk5".

We get at least 32 bits of certainty that we guessed the right password. However, because the key size is restricted to 40 bits, it is feasible to search the whole key space instead of using a dictionary attack.

6.3.3 OutGuess Header Information

Dictionary attacks against OutGuess seem to be infeasible, because we lack information to verify the password guess. OutGuess stores a 32-bit header in front of the embedded message. The header contains a 16-bit seed and 16 bits containing the length of the following message in bytes. We can use only the length to verify our password guess, because the seed can be an arbitrary number. While it is possible to restrain the acceptable seed or include a minimum length check in the password verification, there are still many keys that pass the verification.

As an additional check, Stegbreak retrieves 512 bytes of the encrypted message and checks the retrieved bytes for randomness. The simplest and fastest check is to count the number of zero and one bits. If there are close to 50% one bits then the data seems likely to be random. We further increase the accuracy by applying some simple statistical tests to the data. However, 512 bytes of data is not enough for a thorough test. For a large dictionary, we still find too many candidate passwords, making a dictionary attack infeasible.

6.3.4 Stegbreak Performance

System	Speed
JPHide	15,000 words/s
OutGuess 0.13b	47,000 words/s
JSteg	112,000 words/s

Figure 17: Stegbreak performance on a 1200 MHz Pentium III.

We measure the performance of Stegbreak on a 1200 MHz Pentium III. The results are shown in Figure 17. For JPHide, we can check about 15,000 words per second. A test run with 300 images and a dictionary of about 577,000 words takes ten days to complete. Stegbreak is slow because it has to check for both versions of JPHide. When checking for version 0.5, the Blowfish key schedule needs to be recomputed for almost every image.

Stegbreak is faster for OutGuess: it can check about 47,000 words per second. However, as explained above, with a large dictionary the tool finds candidate passwords for every image.

For JSteg-Shell, we can check about 112,000 words per second. This is fast enough to run a dictionary attack on a single computer. However, because the key space is restricted to 40 bits, it makes more sense to do a brute-force search of the whole key space. The key space is reduced to 40 bits in such a way that effectively only 35 bits are used. On a 1200 MHz Pentium III, a brute-force key search of the 35-bit key space completes within four days.

6.4 Distributed Dictionary Attack

As we have seen, Stegbreak is too slow to run a dictionary attack against JPHide on a single computer. However, because dictionary attack is inherently parallel, it is possible to distribute the dictionary attack to a number of workstations.

Such a distributed computing framework should work on a cluster of loosely-couple workstations that fulfills the following requirements:

- The setup and maintenance of jobs should be simple.
- It should be portable to many operating systems, so that we can use as many different computer systems as possible.
- All communication should be encrypted and authenticated.
- The system should not require "root" privileges for installation.

Because such a system was not available as open-source, we developed "Disconcert."

Disconcert uses libevent for asynchronous event notification and "libio", a library especially developed for use with disconcert. Libio abstracts communication into data sources and data sinks. A data source is connected to a data sink via multiple filters. Using this abstraction, encryption and authentication just become filters. Disconcert has fewer than 7,000 lines of source code.

In the following, we explain a few essential commands that Disconcert supports:

- The **init** command transfers files to selected clients. It is used to copy Stegbreak, word lists and image files to the remote computers.
- The **job** command sets up various parameters for a specific job. This includes the number of work units that should be completed and the command line to be executed on the client machines.
- The **run** command is used to start remote execution of a job. Disconcert sets the "nice" level for these jobs to ten, so that they do not irritate the users of the workstation.

Clients send the exit status of a terminated process to the server to indicate if a work unit has been completed successfully or not. To communicate password guesses or other messages to the server, "stdout" and "stderr" are redirected to files on the server.

If a client loses its connection to the server, all communication is buffered until the client can reconnect. If a client does not reconnect within a certain time frame, the server reassigns the work unit of that client to another machine. The disconcert framework also supports multiple jobs at the same time.

At this writing, Stegbreak is running on sixty clients, ten of them at the Center for Information Technology Integration and fifty on other machines at the University of Michigan.

To prevent transmission of objectionable content (such as pornographic images) to the clients, Stegbreak can extract the information from the JPEG images that is relevant to a dictionary attack and save it as a separate file. For JPHide, the dictionary attack requires only about 512 bytes to verify a password guess. Another benefit of this is a reduction of network traffic.

Stegbreak has very low I/O and memory requirements and is hardly noticeable when running in the background.

The total performance when trying to find content hidden by JPHide is about 200,000 words per second. This is 15 times faster than running on a single 1200 MHz Pentium III. The slowest client contributed 471 words per second to the job, the fastest 12,504 words per second. The average performance of a workstation is around 3,900 words per second.

7 Discussion

At this writing, Crawl has downloaded over two million images from eBay auctions. For these images, Stegdetect indicates that about 17,000 seem to have steganographic content. Of these 17,000 images, 15,000 supposedly have content hidden by JPHide. All 15,000 images have been processed by Stegbreak.

While Stegbreak has been running on a cluster of 60 machines, it is still too slow to process all images that Stegdetect finds. We hope that we will have access to more and better machines in the future.

To verify the correctness of all participating clients, we insert tracer images into every Stegbreak job. As expected the dictionary attack finds the correct passwords for these images. However, so far we have not found a single genuine hidden message. We offer three possible explanations to support our results:

- There is no significant use of steganography on the Internet.
- Nobody uses steganographic systems that we can find.
- All users of steganographic systems carefully choose passwords that are not susceptible to dictionary attacks.

Even if the majority of passwords used to hide content were strong, there would be a small percentage of weak passwords, *e.g.* a study conducted by Klein found nearly 25% of all passwords vulnerable [6]. Weak passwords are susceptible to a dictionary attack and we should have been able to find them. Similarly, even if most of the steganographic systems used to hide messages were undetectable by our methods, we still should find some images with hidden messages from detectable systems. The most likely explanation is that there is no significant use of steganography on the Internet.

The popular press claims that steganographic messages are hidden in images on eBay, Amazon and on "pornographic bulletin boards." So far, we have looked only at images obtained from eBay. Very soon, we will examine content from USENET image groups. Given the high number of false positive images that we found, we also plan to improve the accuracy of Stegdetect.

8 Related Work

Fridrich et al. analyze the security of steganographic systems that embed information in the LSB of color images [2]. They find that the number of pairs of "very close" colors increases when hidden messages have been embedded. While they are able to detect steganographic content, they are not able to differentiate between steganographic systems.

9 Conclusion

Steganography can be used for hidden communication. There are widely reported rumors that images on auction sites contain hidden messages. To verify these claims, we developed new techniques and software to find hidden messages on the Internet:

- Stegdetect allows us to automatically detect steganographic content in JPEG images.
- Crawl is an efficient web crawler that saves JPEG images from web pages that it encounters.
- Stegbreak launches dictionary attacks against steganographic systems to test whether content is indeed hidden in an image.
- Disconcert is a distributed computing framework for a cluster of loosely-coupled workstations used to distribute the dictionary attacks.

Even though we analyzed two million images that we obtained from eBay auctions, we are unable to report finding a single hidden message.

All software is freely available as source code and can be downloaded from `www.outguess.org` and `www.citi.umich.edu/u/provos/`.

10 Acknowledgments

We thank Patrick McDaniel, Jose Nazario and Therese Pasquesi for careful reviews and suggestions. We also thank Mark Giuffrida for providing computing resources.

References

[1] Stefan Axelsson. The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 1–7, November 1999.

[2] Jiri Fridrich, Rui Du, and Meng Long. Steganalysis of LSB Encoding in Color Images. In *Proceedings of the IEEE International Conference on Multimedia and Expo*, August 2000.

[3] F. Johnson and S. Jajodia. Exploring steganography: Seeing the unseen. *IEEE Computer Magazine*, 31(2):26–34, February 1998.

[4] Jack Kelley. Terror groups hide behind Web encryption. USA Today, Feburary 2001. `http://www.usatoday.com/life/cyber/tech/2001-02-05-binladen.htm`.

[5] A. Kerckhoffs. La cryptographie militaire. *Journal des Sciences Militaires*, Feburary 1883.

[6] Daniel Klein. Foiling the Cracker: A Survey of, and Improvements to, Password Security. In *Proceedings of the 2nd USENIX Security Workshop*, pages 5–14, August 1990.

[7] Ueli M. Maurer. A Universal Statistical Test for Random Bit Generators. *Journal of Cryptology*, 5(2):89–105, 1992.

[8] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, Boca Raton, 1996.

[9] Niels Provos. OutGuess - Universal Steganography. `http://www.outguess.org/`, August 1998.

[10] Niels Provos. Libevent - An Asynchronous Event Notification Library. `http://www.monkey.org/~provos/libevent/`, November 2000.

[11] Niels Provos. Defending Against Statistical Steganalysis. In *Proceedings of the 10th USENIX Security Symposium*, pages 323–335, August 2001.

[12] Bruce Schneier. Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish). In *Fast Software Encryption, Cambridge Security Workshop Proceedings*, pages 191–204. Springer-Verlag, December 1993.

[13] RSA Data Security. The RC4 Encryption Algorithm, March 1992.

[14] D. Wagner and S. Bellovin. A Programmable Plaintext Recognizer, 1994.

[15] G. W. Wallace. The JPEG Still Picture Compression Standard. *Communications of the ACM*, 34(4):30–44, April 1991.

[16] Andreas Westfeld and Andreas Pfitzmann. Attacks on Steganographic Systems. In *Proceedings of Information Hiding - Third International Workshop*. Springer Verlag, September 1999.