Advanced Jpeg Carving

Michael I. Cohen
Data specialist
Australian Federal Police
Brisbane, Australia
scudette@gmail.com

ABSTRACT

Data carving is a digital forensic technique which aims to reconstitute a file from unstructured data sources with no knowledge of the previously stored file system. This paper presents an approach for the carving of JPEG files. Since JPEG is one of the most popular image formats in the storage and distribution of digital photographic imagery it is frequently of great interest for certain types of forensic investigations.

We apply the previously developed carving theory to the JPEG image format and develop an accurate, fully automated carver.

We present a novel technique for the suspension and resumption of the decoder, making it possible to carve JPEG images in an acceptable time.

Categories and Subject Descriptors

[**Information**]: Data Carving

General Terms

Data Carving, Digital Forensics, Data Recovery

Keywords

Data Carving, Digital Forensics, Data Recovery

1. INTRODUCTION

Carving is the process by which files are extracted from raw images without the use of filesystem allocation information. Carving techniques are important to the forensic investigator in order to recover deleted files without resorting to directory entries. This is becoming more important as certain types of modern filesystems wipe critical information needed to recover deleted files. (e.g. ext3 wipes the block pointers in the inode and indirect blocks, making file recovery difficult [1]).

Carving is also important for data recovery from images with damaged or incomplete filesystems. For example, a popular carving tool was originally written to recover photos from damaged memory cards[8].

The general usefulness of carving technology has spurred a great deal of research efforts recently. Most notable is the Digital Forensic Research Work-Shop's (DFRWS) forensic challenge for 2006 and 2007[2]. These challenges have posed extremely difficult images, encouraging the development of techniques and tools to advance the state of the art.

The DFRWS 2007 challenge required the development of automated solutions, and in particular [4] has developed a formal theory for carving. This paper explores how this theory can be applied for the carving of image files, and in particular JPEG files.

We present a fully automated JPEG carver which is capable of solving the DFRWS challenges. The DFRWS challenges generally do not reflect typical fragmentation as found in real filesystems, rather they are designed to simulate extreme carving scenarios. We present a statistical analysis of typical carving scenarios with a variety of different filesystems, and develop statistical fragmantation models for each file system. This model is then applied to the JPEG carver in order to increase its efficiency and speed of recovery with real world file systems. The carver is released under the GPL license as part of the PyFlag project[5].

2. THEORY OF CARVING

The detailed theory of carving is presented in [4, 6]. The basic definitions are repeated below to clarify the ensuing discussion.

2.1 Definitions

In the following discussion the term *file* refers to the file to be recovered (in our case the JPEG file) while the term *image* refers to the forensically acquired disk image the file is to be recovered from. The image is taken to contain contiguous bytes linearly representing the data obtained from the hard disk (as acquired with the dd program for example).

2.1.1 The mapping function

The process of carving can be defined as extracting (or copying) all the bytes belonging to a file from an image. That is to say, that there is a mapping between the bytes contained in the file to the bytes within the image itself. This mapping can be described as a mapping function.

An example of a typical mapping function is shown in Figure 1. The mapping function has a number of interesting properties:

- The slope of the function is always 1 (because there is

E-FORENSICS 2008, January 21-23, Adelaide, Australia

DOI 10.4108/e-forensics.2008.2643

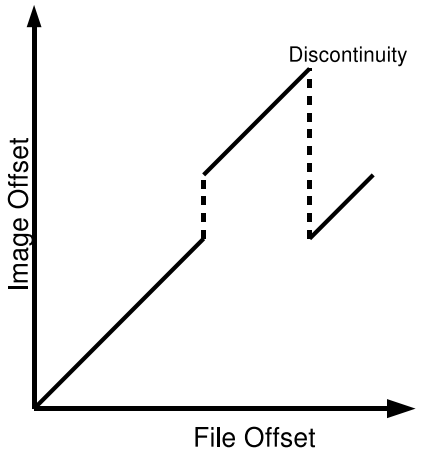

Figure 1: An example of a mapping function

a 1-1 mapping for the image with the file).

- There are a number of discontinuities in the function at various places.
- The function is invertible - i.e. there is at most one file offset for each value of the image offset (if such a value exists). In practice this means that once a sector from the image has been used it can not be used again.

The process of carving is therefore equivalent to estimating the mapping function.

2.1.2 The discriminator

The Discriminator is a process which rates the validity of the mapping function in comparison to other mapping functions.

The discriminator can tell the difference between a recovered file which is corrupt and one which is likely to be correct. A good discriminator is able to detect corruption in the recovered file with a great level of certainty, and in particular localise the corruption within the file - i.e. identify which sections of the file are correct and which are not.

In the literature this component is often referred to as a *Validator*[7]. We use the term *discriminator* to emphasise that the discriminator not only returns a valid/invalid verdict, but also helps to shape the choice of future mapping functions. In this way we can discriminate in favour of better mapping functions over functions which are still valid but less optimal.

The discriminator's job is essentially to detect when the carved file is inconsistent with its expected internal structure. If the file type has little internal structure, this task is very difficult. The discriminator may resort to using statistical techniques or semantic techniques (to identify human languages) to try to detect corruption in less structured files.

2.1.3 Mapping function generator

A Mapping function generator is a component which generates new mapping functions to be tested by the discriminator.

The generator may use information from the discriminator itself as well as external means to influence its choice of new functions. For example, external information such as knowledge of blocks already allocated to files (as obtained for example from a filesystem analysis) can be used to discount certain sectors from the mapping function to be tested.

An efficient discriminator can be used to drive the choice of new mapping functions by isolating the parts of the file which are deemed correct, and trying different sectors for those parts of the file which are corrupted. The function generator can then use this information to select better mapping functions. The function generator may have a number of different discriminators available to it and may use different discriminators under different situations.

Ideally a discriminator is able to verify the file sequentially sector by sector and immediately detect the point of fragmentation accurately. In that case the function generator can simply piece new sectors to the fragmented region until a match is made. In practice however, a corrupted sector is often only detected once the next few sectors are decoded (e.g. when decompressing data). In any case, the discriminator needs to provide as accurate an estimate of the corrupted sector as possible.

The process of carving can be viewed as a mathematical discrete optimisation problem - the goal is to find the mapping function which minimizes the error rate in the discriminator. A purely combinatorial optimization algorithm is typically infeasible due to the large number of possibilities[10]. However, there may be a number of assumptions that can be used to reduce the total number of possible mapping functions, and assist the generator in optimising the mapping function.

2.2 Simplifying assumptions

The first kind of assumption may arise from analysing the internal file structure and thereby identifying constraints upon the mapping function itself. The constraints may be positive or negative constraints:

2.2.1 Positive constraint

Positive constraints are deductions which identify a likely point on the mapping function. This likely point is referred to as an *identified point*.

A positive constraint might actually deduce a number of possible points on the function - only one of which belongs to the true function. A good positive constraint reduces the number of possibilities to just one or very few such points, and possibly estimate the probability of occurrence.

For example, an identified file header places a positive constraint on the first sector of the file. In some file formats, positive constraints may be identified throughout the file.

2.2.2 Negative constraint

Negative constraints are deductions which exclude certain points from the mapping function.

A negative constraint can be defined as an observation that a certain sector does not belong within the sequence observed. This might occur, for example, if the sector clearly does not exhibit the characteristics required from this file type (e.g. sector with binary bytes following a HTML sector etc). These constrains may be detected by running statistical classification algorithms over the disk and excluding potential sectors based on their statistical characteristics.

2.2.3 Projection line

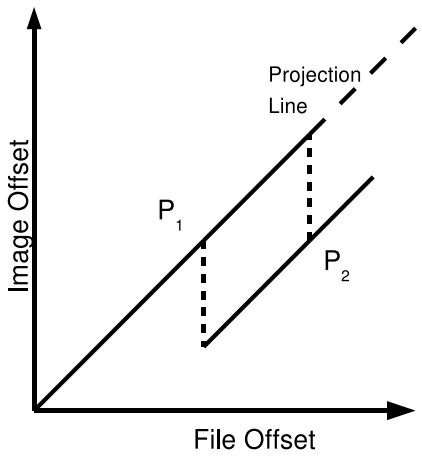

Figure 2: An example of an impossible mapping function

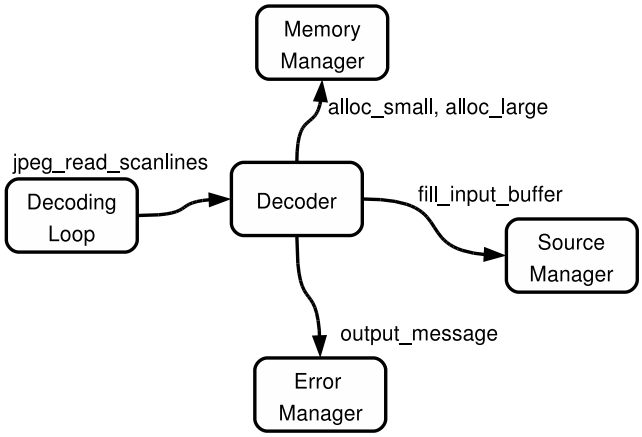

Figure 3: An overview of libjpeg architecture

Consider the mapping function with identified points P_1 and P_2 shown in Figure 2. The projection line is a line of unit slope which passes through P_1. The second identified point P_2 lies below the projection line. Assuming only first order fragmentation, it is impossible to interpolate a valid mapping function between P_1 and P_2, because the function is not invertible[1].

This property of mapping functions can be used in many cases to immediately spot situations where first order fragmentation is impossible, and an exhaustive search is required.

3. THE JPEG FILE FORMAT

JPEG is the most common file format for storage and distribution of photographic imagery. This standard is defined in detail in [9], but here we discuss those aspects of the format which are directly relevant to the construction of carvers.

A JPEG file is divided into a series of markers representing different sections within the file. Although the markers do contain length parameters for the size of the headers, compressed data appears between the headers and has no length parameter. In particular the SOS (Start of Scan) marker header is followed by the ECS (Entropy Coded Segment) which does not have its length specified in advance. In addition, the ECS section is typically the largest section of the file, representing the majority of data in the JPEG file.

Unlike the examples presented by [6] of the PDF and ZIP file formats, the JPEG file format is less structured. It is impossible to know in advance the file offset of most markers. This is mainly due to the fact that some sections are unsized. This makes it difficult to find a sufficient number of positively identified points. Unfortunately this implies that JPEG carvers usually need to resort to exhaustive searches.

3.1 The JPEG Discriminator

[1]Mapping functions must be invertible because there is a 1-1 relation between file offsets and image offsets. The above example has regions where image data is used twice in the file

As mentioned in 2.1.2, a good discriminator will provide a good fix on the location of the discontinuities within the file. For the JPEG discriminator, we need to identify the sector offset of the discontinuity as accurately as possible.

Our discriminator uses the libjpeg decoder[11]. This library is extremely well designed and supports a number of operating modes which are useful in different situations. The strength of the library is that its possible to replace parts of the decoder to achieve different goals.

Figure 3 illustrates how libjpeg is used:

- A decoder loop retrieves each scan line using the *jpeg_read_scanlines* function.
- The main JPEG decoder then relies on its subsystems to decode the required scan lines and return them. Only complete scan lines are returned.
- When the decoder requires more data it calls *fill_input_buffer* from the *source manager*. This allows the source manager to provide compressed data in small chunks.
- The decoder uses the *memory manager* to allocate memory for its internal operation.
- If an error is detected by the decoder, the decoder calls the *error manager* to display the message using its *output_message* function.

In our case we want to detect errors as soon as they occur. We therefore create our own error manager to monitor the error count from the decoder. Our source manager then provides the decoder with one sector at the time and watches the error count in order to detect the exact sector which causes an error.

Unfortunately the jpeg decoder itself is often unable to detect errors soon enough after they occur. This makes the detection of bad sectors unreliable.

Figure 4 illustrates an example of an image containing a major corruption but without causing libjpeg errors. Although the corruption can easily be detected visually, the decoder is unable to find errors within the compressed stream. On closer examination of the corrupted region, as illustrated

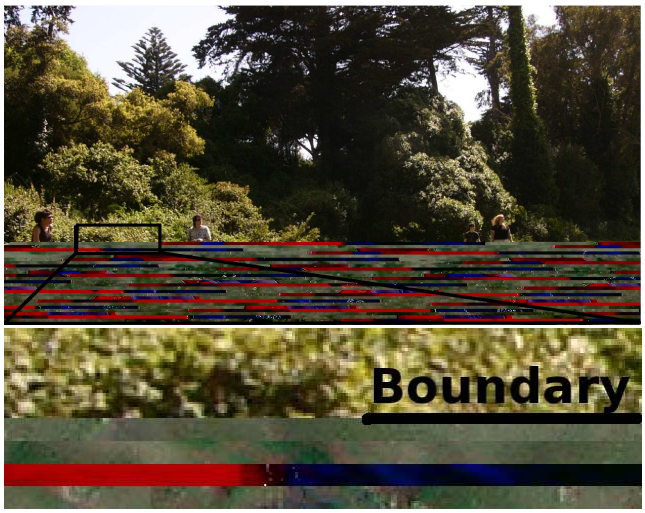

Figure 4: An example of an image corruption which is not detectable by libjpegs error detection. This image was taken from the 2007 DFRWS header offset 90377 sectors. The inset illustrates the boundary between the scan lines which are correct and those which are corrupted.

by the inset, we can see that there is a clear edge between the two regions based on the different textures between the corrupted region and the correctly decoded region.

Our carver uses an edge detection technique to estimate the error levels in addition to libjpegs internal error detection mechanisms. This allows us to be more accurate with detecting possible errors. The presence of an edge is estimated by calculating the estimate:

$$E(y) = \sum_{x=0}^{width} \left| f_{x,y} - \frac{f_{x,y+1} + f_{x,y-1}}{2} \right| \quad (1)$$

Where $E(y)$ is the estimate of scan line y, and $f_{x,y}$ is the component value at scan line y and pixel x. The estimate is thus merely an integral of the difference between the actual component value of a line and the expected value derived by averaging the line above and below it. For JPEG photographic images component values tend to vary smoothly at the pixel level. In deriving estimate we therefore apply a linear approximation to the vertical component of the component values.

An example of this estimate for all scan lines is shown in Figure 5.

Sometimes the bad sector will cause a corruption in the middle of a scan line. This makes it difficult to detect because the decoder will typically consume a number of sectors in order to complete the line before returning the full scan line to us for edge detection. Our carver solves this problem by inserting an end marker (Hex code *0xFFD9*) code immediately after the sector of interest. This causes the decoder to flush all compressed data and return a partial line. We are then able to determine how many horizontal pixels were decoded and restrict our edge detection to the difference between the decoded frame resulting from decoding subsequent sectors.

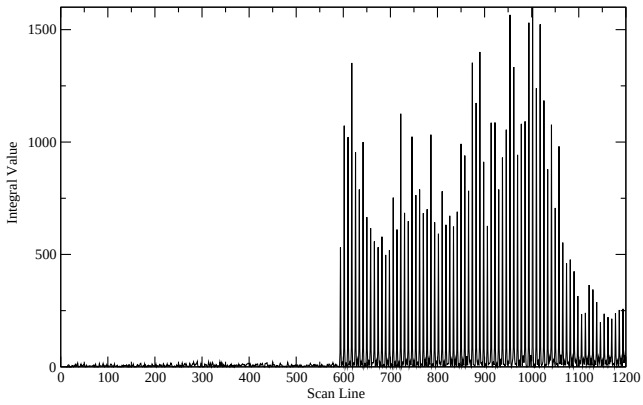

Figure 5: A graph of the value of the estimate Eq. 1 calculated for each scan line of the image shown in Figure 4. As can be seen at scan line 594 the integral rises dramatically with severe oscillations from then on.

Figure 6: An example of an image corruption which can be revealed by decoding sectors one at the time. This image (of a dog) was taken from the 2007 DFRWS at header offset 87716 sectors. The top part is shown after decoding 243 sectors, while the bottom part was after decoding 244 sectors. The black bands illustrate the region over which the Estimate (Eq. 1) was taken. This region is restricted to the boundary between the smaller image and the image obtained by decoding more data.

The estimate (Eq. 1) is then calculated for the difference between the image resulting from decoding the previous sector and this sector.

Figure 6 illustrates this process. The top part of this figure shows the image resulting from decoding only 243 sectors of the image (and artificially inserting an End Of Image (EOI) marker). The bottom part of this image shows the decoded result of decoding 244 sectors. The edge detection algorithm is then applied to the difference between the two frames and indicates an error.

3.2 The mapping function generator

The mapping function generator is a component of the carver which is responsible for choosing possible mapping functions to be tested by the discriminator. In our case the generator is focused on resolving each discontinuity separately. The discriminator is asked to evaluate the validity of the mapping function only up to a point shortly after the discontinuity. This allows each discontinuity to be resolved

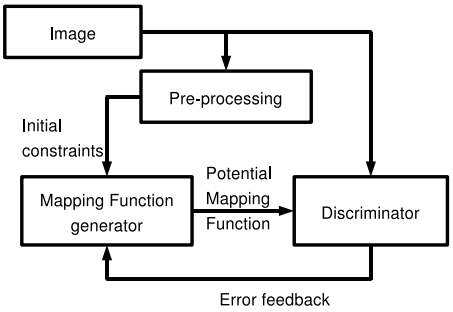

Figure 7: Overview of Carving algorithms

in turn until the complete image is recovered.

3.3 Carving Algorithm

The overall algorithm is illustrated in Figure 7:

- The image is preprocessed to locate JPEG headers. This positively identifies a single point on the mapping function.
- The mapping function is decoded a sector at the time to establish the point at which errors begin to occur. This provides a narrow range on the file offset where a discontinuity may exist.
- A function generator is used to choose possible sectors to append at the identified discontinuity and the discriminator is asked to assess the resultant image if decoded a few sectors past the discontinuity.
- When a sector was found which does not result in errors, the algorithm goes back to identifying the next discontinuity.

3.4 Fragmentation model

Fragmentation occurs in hard disks as a result of filesystem allocation strategy. These strategies are typically designed to optimise some characteristic of the filesystem (e.g. faster access speed, better storage efficiency etc) and are not deliberately designed to make recovery difficult. It is often possible to make assumptions about the type of fragmentation present based on information about the file system which was previously on the disk and its typical allocation strategies.

A fragmentation model is a set of assumptions derived from an observation of how fragmentation occurs in practice. This may depend on filesystem characteristics or other simplifying assumptions.

The model can be applied to the generator by reducing the number of mapping functions which need to be examined, or having the mapping functions tested in likely order of occurrence.

For example the DFRWS challenge [3] states the following assumptions:

- Files begin on sector boundaries.
- Fragmentation can only occur on sector boundaries.
- Sectors are 512 bytes in size.

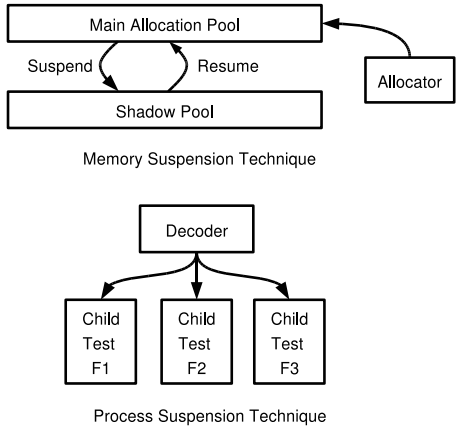

Figure 8: Decoder suspension techniques. Top: Memory based suspension. Bottom: Process based suspension.

We also assume that fragments are more likely to be localised. That is, to say that sectors close to the file header are more likely to match than sectors further away. This assumption is common in real filesystems which attempt to keep file fragments physically close on the disk. For example [7] shows that in the wild most bifragmented files were separated by a reasonably small number of sectors. The same reference also shows that modern filesystems such as Ext2 and NTFS contain files which are fragmented on a filesystem blocksize of 4kb even larger. This allows for more accurate determination of the exact block of corruption by the discriminator because there is more data within the incorrect block to cause more of the image to be decoded incorrectly - thereby increasing the value of the estimate.

4. OPTIMIZATIONS

Although the carver presented in this work is very effective in determining the correct mapping function, it still needs to perform an exhaustive search. For each sector tested, the decoder is used to decode the full image from the start until a particular point in the image (a little after the discontinuity identified). This is very inefficient because for subsequent decoding operations, most of the image is exactly the same, differing only in its final sectors.

Ideally the decoder could be suspended at the point just prior to the position where the discontinuity can be found. The decoder can then be used to decode the region in question and rewind back to the suspension point without needing to re-decode the file from the start. Unfortunately, libjpeg is not designed in this way and does not provide means to suspend the decoder. We devise two novel approaches of suspending the decoder without needing to modify the source code of the decoder itself.

The following two techniques are illustrated in Figure 8.

4.1 Memory suspension

One can think of the decoder as a finite state machine. The future evolution of the decoder is fully determined by its current state and future input. Since libjpeg is a reentrant library, it follows that no state information can be stored within static library variables. All state information

is stored by memory allocated to the specific instance of the decoder.

As shown in Figure 3, libjpeg relies on a memory manager subsystem to allocate internal memory through a specialised allocator. We introduce a special memory manager which allocates memory from a single primary pool (Figure 8).

The pool can be copied on demand to a shadow pool in order to form a snapshot of the decoder's memory without needing to understand the specifics of how the decoder operates. This memory snapshot in addition to the file offsets of any file descriptors embodies the full decoder state.

This can be used to restore the decoder to a known state by restoring the pool from its snapshot in the shadow pool. In addition to the memory, we need to also rewind file descriptors to their saved offsets.

The overall result of this is that a snapshot is created just before the identified discontinuity, and then this snapshot is resumed with each new sector tested. The decoder then decodes just the new region of the image from the point of the snapshot on, rather than decoding the entire image from the start.

Because using the memory suspension technique we have a well defined concept of the decoder state (namely the primary allocation pool contents and file descriptor offsets), this state can easily be shared by different machines or processes on the same machine. This approach allows us to spread the testing loads between many CPUs and machines in a cluster, thereby taking advantage of any parallelism inherent with the carving process.

4.2 Forking/Process suspension technique

The forking/process suspension technique is much simpler to implement than the memory suspension technique described above. The basic technique relies on the decoder process forking at the suspension point, sending its child process to continue decoding the test function. The parent then waits for the child to exit with an exit code which informs the parent about the result of the test (A pass/fail exit code).

The technique works with any decoder which decodes the file linearly (e.g. MPEG, Zlib, etc), without knowledge of the internal decoder state.

The process suspension technique allows the decoder process to fork several times simultaneously, and thereby have several child processes test a number of functions simultaneously. This approach takes advantage of multiple processors on the system. Unlike the memory suspension technique, however, its difficult to share decoder states between different systems since the decoder state is effectively represented by the process's memory image and can not be easily extracted.

5. CONCLUSIONS

Data carving is a very promising forensic technology. This paper examined the JPEG file format and the development of efficient and automated carvers for it. A discriminator was presented which used edge detection and texture mapping operations to determined errors in the decoded image. The discriminator was based on libjpeg and operates by examining errors in the decoded image as each sector is processed. Finally an optimization was presented which relies on being able to store and resume the state of the decoder at arbitrary points without knowledge of the exact operation of the decoder.

Although this paper concentrated on the JPEG file format, the techniques described can be applied equally well to all image formats. In particular the detection of discontinuities based on image properties such as textures and edges can be applied to any image formats (or indeed video formats).

6. REFERENCES

[1] B. Carrier. *File System Forensic Analysis*. Addison Wesley Professional, March 2005.

[2] B. Carrier, E. Casey, and W. Venema. DFRWS 2006 forensics challenge, URL `http://dfrws.org/2006/challenge/` 2006.

[3] B. Carrier, E. Casey, and W. Venema. DFRWS 2007 forensics challenge, URL `http://dfrws.org/2007/challenge/` July 9 2007.

[4] M. Cohen. Advanced carving - DFRWS submission, URL `http://sandbox.dfrws.org/2007/cohen/` 2007.

[5] M. Cohen and D. Collett. Python Forensic Log Analysis GUI (PyFlag), URL `http://www.pyflag.net/` 2005.

[6] M. I. Cohen. Advanced carving techniques. *Digital Investigation*, 2007. In Press.

[7] S. L. Gafinkel. Carving contiguous and fragmented files with fast object validation. *Digital Investigation*, 4(Supplement 1):2–12, September 2007.

[8] C. Grenier. Photorec, URL `http://www.cgsecurity.org/wiki/PhotoRec` 2007.

[9] Joint Photographic Experts Group. JPEG standard (JPEG ISO/IEC 10918-1 ITU-T Recommendation T.81), URL `http://www.w3.org/Graphics/JPEG/itu-t81.pdf` 1992.

[10] A. Schrijver. A course in combinatorial optimization, URL `http://homepages.cwi.nl/~lex/files/dict.pdf` February 1 2006.

[11] The Independent JPEG Group. Libjpeg 6b, URL `http://www.ijg.org` March 1998.