Identification and Recovery of JPEG Files with Missing Fragments

By

Husrev Sencar and Nasir Memon

From the proceedings of

The Digital Forensic Research Conference

DFRWS 2009 USA

Montreal, Canada (Aug 17th - 19th)

available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Identification and recovery of JPEG files with missing fragments

Husrev T. Sencar[*], *Nasir Memon*

TOBB University of Economics and Technology, Computer Engineering Department, Sogutozu Cad. No: 43, Ankara 06560, Turkey

Keywords:
File recovery
Fragmentation
JPEG/JFIF
Huffman code tables

A B S T R A C T

Recovery of fragmented files proves to be a challenging task for encoded files like JPEG. In this paper, we consider techniques for addressing two issues related to fragmented JPEG file recovery. First issue concerns more efficient identification of the next fragment of a file undergoing recovery. Second issue concerns the recovery of file fragments which cannot be linked to an existing image header or for which there is no available image header. Current file recovery approaches are not well suited to deal with these practical issues. In addressing these problems, we utilize JPEG file format specifications. More specifically, we propose a technique based on bit sequence matching to identify fragments created by the same Huffman code tables. We also address the construction of a *pseudo header* needed for recovery of stand-alone file fragments. Some experimental results are provided to support our claims.

1. Introduction

Ability to recover files from a storage volume without file system information involves several algorithmic and statistical challenges. These challenges are primarily a result of two main factors. The first and foremost is the inability to lay out file data contiguously on the storage. Essentially, systems store files by breaking them into fixed-size chunks of data called blocks that are arranged in some physical or logical sequential order. In conventional hard disk drives, a data block is located in multiple consecutive physical sectors of radially concentric tracks, and to achieve shorter access time file data is stored on consecutive blocks. However, over time as the files on a disk are added, deleted and changed in size file fragmentation unavoidably occurs as such modifications will externally fragment the free space and will leave only small clusters of unallocated blocks for storage of new files. By contrast, the newly emerging solid state drives have blocks of memory chips and have no sectors or tracks. With these drives the main concern is to increase the life of the drive by wearing out the chips at an even rate. Therefore a layer of logical remapping is needed to spread the writes among all chips evenly. As a result of the *wear-levelling*, all files are inherently physically fragmented, and in the case of an interval drive controller failure, most severe instance of file fragmentation occurs. The problem with fragmentation is that, as a result, pieces of fragmented files will be distributed all over the storage volume and, in the lack of any file system metadata, different pieces of a file cannot be trivially identified. Fig. 1 depicts a layout of files distributed across logical blocks of a storage media to illustrate file fragmentation.

The other factor is the considerable variety and complexity in the ways information can be coded into binary format. In most file formats, data is stored in encoded form to provide compression in size, error checking and correction, and security. This requires sophisticated encoders and decoders to interpret the file data. As a result, pieces of a file reveal little or no information about the content of the file. When combined

[*] *Corresponding author.*
E-mail address: htsencar@etu.edu.tr (H.T. Sencar).

1742-2876/$ – see front matter © 2009 Digital Forensic Research Workshop. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2009.06.007

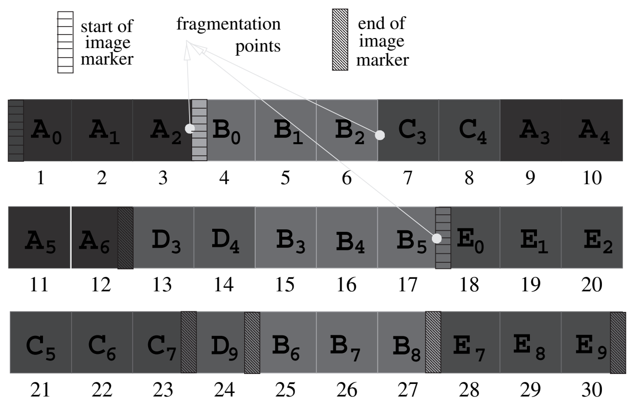

Fig. 1 – File A has been broken into 2 fragments spanning 7 blocks and file B into 3 fragments spanning 9 blocks. The file fragments are distributed across 30 logical blocks. In between the fragments of the two files are small fragments of files C, D and E.

with uncertainties concerning fragmentation level of files on a drive (like number of pieces that the file is fragmented into, size of each piece, and how the pieces of a file are scattered over the drive), inability to interpret stored information in individual data blocks greatly complicates the recovery of fragmented files.

Today, there are only a few techniques that allow recovery of fragmented files (Pal et al., 2003, 2008; Pal and Memon, 2006; Garfinkel, 2007). These techniques are proven to be very effective in recovering files that are not fragmented into many pieces and when those pieces are not placed far from each other on the storage. Nevertheless, these techniques have certain limitations that warrant further research. An inherent shortcoming of all current fragmented file recovery approaches is that recovery process is not error tolerant. That is, in cases where part of a file's data (which may include a single bit) is deleted or corrupted, the file can be recovered only up to the point of disruption in the data, and rest of the file data cannot be considered during recovery. Ultimately, if the header of the file is missing, then the file cannot be recovered at all even though rest of its data might be intact. Such disruptions arise most commonly when a file on the storage is deleted by only removing its file metadata and designating those data blocks (associated with the file) as unused space without actually erasing the data on them. In such cases, when a new file's data is written over blocks that contain deleted files, those files can only be recovered until the disruption in the file data.

The other limitation emerges from the difficulties in identifying start and end points of file fragments. In current approaches, the procedure used to identify the fragments of a file creates a computational bottleneck when the storage volume contains large amounts of data and when a file is fragmented into more than two pieces. In such cases, only single or few fragments of each file are rather easily identified and the remaining fragments have to be searched in large clusters of data blocks. To identify the remaining fragments, those data blocks have to be subjected to a comprehensive analysis as each cluster may contain a mix of fragments from many files as well as random data and unallocated blocks.

Focusing on recovery of fragmented JPEG images, in this work, we attempt to address two fundamental questions. First, how should the recovery be performed for files whose headers cannot be identified or not available. And second, given clusters of data blocks, how can one effectively and reliably determine whether or not pieces of a file are contained within these clusters. Crucially, the first question addresses the dependency on file header in bootstrapping the recovery process. The second one concerns with the speed at which recovery can be conducted. Our approach utilizes specifics of JPEG file format in its operation; however, the underlying ideas can be generalized to other similar file formats.

Next, we provide an overview of current fragmented file recovery approaches and give a more detailed description of our approach. Section 4 provides a brief description of JPEG image compression standard with special emphasis on its file format. The utilization of structural properties of JPEG files in identification of file fragments is described in Section 3. We focus on recovery of file fragments that cannot be linked to a file header in Section 5. Our conclusions are presented in Section 6.

2. Recovering JPEG files

JPEG is the most widely adopted still image compression standard. It is the default format saved by most digital cameras and camera phones. JPEG enjoys such popularity not because it provides the best (rate/distortion) performance and coding flexibility but rather because of its simplicity and low computational resource requirements. Below, we describe in more detail proposed fragmented JPEG file recovery approaches. Then, we will describe how these approaches can be further improved.

2.1. *Review of existing approaches*

Three techniques have been proposed for recovery of fragmented JPEG files (Pal and Memon, 2006; Garfinkel, 2007; Pal et al., 2008).[1] In their operation, these techniques primarily utilize detailed knowledge of the file system and the JPEG file format. The deployed procedure for recovery can be viewed to consist of three phases, despite a number of important differences between these techniques. A JPEG file contains an ordered sequence of markers, parameters, and entropy-encoded segments that are spread over multiple blocks. However, to be able to discriminate JPEG file data from non-JPEG data one can only rely on the markers. Therefore, in the first stage, all data blocks of the storage device are scanned for known file markers. Each marker is two bytes in length with the first byte always having 0xFF value and the second byte containing a code that specifies the marker type. Although there are many markers, few can be rather easily distinguished. Among these, the most important one is the start of image marker which will be located at the beginning of a block

[1] A detailed side-by-side comparison of these techniques can be found in (Pal et al., 2008).

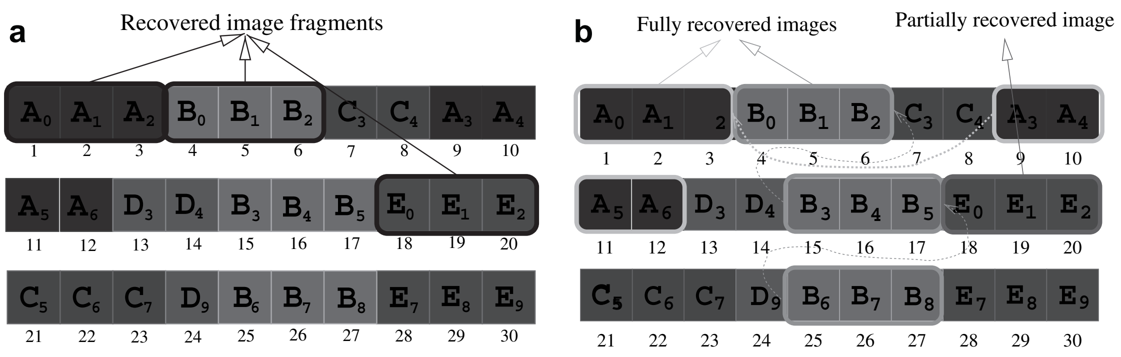

Fig. 2 – (a) Once all the start of image markers are identified, a fragmentation point detection technique will identify the first fragments of files A, B and E. (b) After all the remaining data blocks are scanned to identify the remaining fragments, files A and B will be fully recovered and E will only be partially recovered, as its middle piece is unavailable. No fragments of files C and D will be identified since their headers are not available.

and mark the beginning of a file. The end of file marker, which will be located in the last file block, and the restart markers, that will appear almost periodically to establish synchronization, are the other useful ones.

In the second phase, data extracted from the first file block, containing the start of image marker, is combined together with the data extracted from the consecutive data blocks. Every time a new data block is merged with the previously merged file blocks a decoding operation is performed on the resulting data to test whether a fragmentation point is reached. In Garfinkel (2007), a fragmentation point is deemed to be reached when an error occurs during decoding and the lastly merged block is discarded from the partially recovered file data. However, this procedure is not accurate since the decoder may quite easily confuse random block data for entropy-coded data. To compensate for this drawback, Pal et al. (2008) proposed an improved technique wherein each successful decoding operation is followed by a test to detect abrupt pixel intensity changes in the most recently merged regions. In this case, a fragmentation point is deemed to be reached when the test indicates that the change is significant. In essence, these methods exploit the fact that data blocks used for storing a file are allocated sequentially, as much as possible. If a file is not fragmented, this phase will result with the full recovery of the file. Considering the fragmented file system given in Fig. 1, Fig. 2-a shows the file fragments that can be recovered in the second phase.

In the last phase, other available pieces of a file are recovered. This actually requires identifying starting blocks of the remaining pieces of a file. Since a file can be fragmented at any point, file markers cannot be relied upon for the identification of starting blocks of file fragments and all unmerged blocks from the second phase have to be taken into consideration. More critically, a decision as to whether a block is a starting block can only be made using the above fragmentation point detection technique (which includes merging the block with the already identified file blocks, decoding and testing). This procedure is repeated until all available pieces of a file are identified. However, if a piece of a file is not available, the file will be recovered only up to the missing piece even though subsequent pieces may be present. Fig. 2-b displays the final result of the recovery process for the same system.

2.2. *Our approach*

In the light of current approaches to fragmented JPEG file recovery, we aim at improving the proposed techniques in the two following aspects.

1. After the first piece of a fragmented file is identified, see Fig. 2-a, in the third phase, the search for the starting block of the remaining pieces may prove too costly. This is due to the fact that decoding is a computationally intensive task and the number of unmerged blocks that needs to be considered may be too large. To alleviate this problem, we propose a method based on bit pattern matching to identify sequences of data blocks that are more likely to contain other pieces of the same file. That is, certain clusters of blocks will be prioritized in search for the starting block of a file fragment rather than performing a more random search.
2. The above summarized file recovery procedure starts by identifying all the image headers. This is rightfully so, because header part of an image contains necessary parameters needed for its decoding. Since the use of a decoder is at the core of current approaches two types of file fragments cannot be recovered. First is the *stand-alone fragments* whose headers are not available. Second is the *disrupted fragments* which cannot be linked to a header due to loss of data. Given this dependency on the header information and the use of a decoder, we investigate the possibility of recovering these two types of file fragments.

After describing JPEG file format, in the following sections, we will focus on JFIF-encoded baseline JPEG image files and discuss the details of our approach.

3. JPEG file format

JPEG is a standard developed for lossy compression of images, and it essentially specifies a procedure for transforming images into a stream of bytes and vice versa (Miano, 1999). The JPEG Standard defines four compression modes: lossless, sequential (baseline), progressive and hierarchical. In sequential mode, a JPEG file is stored as one top-to-bottom scan of the image. By contrast, the progressive mode allows progressive transmission of images by performing multiple scans and the hierarchical mode represents an image at multiple resolutions. Sequential mode is required as a default capability ant it is by far the most-often-used JPEG mode.

JPEG image compression is essentially based on discrete cosine transform (DCT) coding with run-length and Huffman encoding. Typical JPEG encoders have four main steps. First, the input image is divided into non-overlapping blocks of size 8×8 pixels. Then, two-dimensional DCT is applied to each of the blocks to obtain transform coefficients in frequency domain. The resulting coefficients are quantized to discard the least important information. Finally, quantized transform coefficients are reordered via zigzag scanning within the transformed block and are passed through a lossless entropy coder to remove any further redundancy. For decoding, the above described steps are performed in reverse order.

The JPEG standard specifies *JPEG Interchange Format* (JIF) as the file format. However, since many of the options in the standard are not commonly used and due to some details that are left unspecified by the standard, several simple but JIF compliant standards have emerged. *JPEG File Interchange Format* (JFIF) and an extended variant of it, which provides additional supplementary information, produced by digital cameras known as *Exchangeable image file format* (ExIF) are the two most common formats used for storing and transmitting JPEG images.

JFIF specifies a standard color space called YCbCr for encoding color information. This color model consists of three color components: Y, Cb and Cr. The Y component (luminance) approximates the brightness information while the Cb and Cr chroma components approximate the color information. To take advantage of the sensitivity of human eye to changes in brightness as compared to color differences, chroma components can be subsampled to have different resolutions. When deployed chroma subsampling reduces the chrominance resolution by a factor of two in either horizontal/vertical direction or in both horizontal and vertical directions. During compression, Y, Cb and Cr components are partitioned into 8×8 blocks and processed separately.

From file recovery point of view, the most important aspect of JPEG is the storage format used in encapsulation of images bytes in the storage media. Every JPEG image is actually stored as a series of compressed image tiles that are usually 8×8, 16×8 or 16×16 pixels in size. The term minimum coded unit (MCU) is used to refer to each of these tiles. Each MCU consists of a certain number of blocks from each color component. The number of blocks per color component in an MCU is essentially determined by the chroma subsampling rate. If no subsampling is performed each MCU represents an area of 8×8 pixels and consists of Y, Cb and Cr blocks. When the horizontal chroma resolution is halved, the MCU represents an area of 16×8 pixels. As a result of subsampling in the chroma information, one MCU is composed of four blocks: two Y blocks, a Cb block and a Cr block. When the chroma resolution is halved in both horizontal and vertical directions, the MCU represents an area of 16×16 pixels and consists of four Y blocks, a Cb block and a Cr block. Each MCU is then encoded as a sequence of its luminance data followed by the chrominance data.

The image that underwent color-space transformation, chroma subsampling, block partitioning, transform coding and quantization has to be entropy coded before it can be stored as a JPEG file. Entropy coding starts by reordering the coefficients of the block in zigzag scan order. As a result, non-zero coefficients will be grouped together and a large runs of zero coefficients will be produced at the end of each scan to which run-length coding will be applied. This run-length coded coefficient values are further encoded with Huffman coding. This is realized by assigning a code word to each coefficient value from a set of Huffman code tables.

Finally, several markers are intermixed with entropy coded data to indicate the start and end of different data fields and to specify parameters, that are commonly referred to as header data and include image details, compression mode, quantization tables, Huffman tables, and other relevant information needed for decoding.

4. Use of Huffman code tables in identification of file fragments

One of the main difficulty faced during recovery of fragmented files is when the file data is encoded. This severely impedes the ability to identify all file fragments. In the case of JPEG files, only the first fragment containing the header part of a file can be easily identified. On a high capacity storage volume, this leaves large clusters of data blocks, between the detected fragments. Each of these clusters may contain other file fragments, random data and unallocated blocks, and the data stored in these blocks needs to be examined for identification of remaining file fragments.

In coping with this problem, two of the proposed techniques have taken similar approaches. Both of the methods are built upon the observations that files fragmented into greater than three fragments are very rare and the gap between the fragments of a file is likely to be small (Garfinkel, 2007). Considering bifragmented files, Garfinkel (2007) identifies two fragments of a file as two contiguous sets of data blocks randomly carved out from the range of blocks that mark the start and end of the file. An exhaustive search continues until identifying two fragments which when combined together can be decoded successfully. Alternatively, in Pal et al. (2008), it is assumed that the starting point of the next fragment will be within next 5000 blocks of the most recently identified fragmentation point. Therefore during recovery, the method merges each block with the already identified file blocks and attempts to decode the resulting data. A matching metric is also utilized to ensure that each successful decoding preserves the continuity at pixel boundary created by the merging of blocks.

In both methods, repeated application of decoding is the key enabler in identifying the starting point of the next file fragment. However, decoding is a computationally intensive task, and when deployed in recovery of files that are fragmented into many pieces and with large gaps of blocks between its fragments, above methods become computationally infeasible. This is not only because many more blocks have to be decoded in recovering each file, but also because the same set of blocks will be unnecessarily decoded during recovery of many other files. To alleviate this problem and to extend the capabilities of current techniques to handle severely fragmented files, we propose to utilize structural properties of individual files to identify data blocks whose structural properties are similar to those of the identified blocks.

JPEG file data contains two classes of segments: the entropy coded segments, which is typically much larger in size and contain the coded image data, and the marker segments, which contain all the information needed for decoding the entropy coded data. Entropy coding is used to further compress the quantized coefficients by efficiently encoding the most likely occurring values by shorter binary sequences. The Huffman look-up tables used to encode are a part of JPEG header. Ideally, for better compression, each code table has to be tailored specifically for each image file; however, this is rarely the case. When combined with the limitations of Huffman coding (as compared to Arithmetic coding) this essentially means Huffman coded sequences will not be sufficiently random. It also implies that two entropy coded sequences using different Huffman tables can be distinguished from each other. This fact forms the basis of our approach.

More formally, the underlying idea is the following. To distinguish an entropy-coded data, generated by a given Huffman code table, from another, we propose to use the occurrence frequencies of certain bit patterns constructed according to JPEG file format. This can be viewed as a simplified decoder but without the sensitivity to structural properties of the file. Consider a number of data block clusters and a partially recovered file whose next fragment will be searched in those clusters. Rather than trying to decode all the blocks, a number of bit patterns will be generated and searched in each of these clusters to identify the most likely cluster(s) that may contain the next fragment. The starting block of the next fragment will then be identified through decoding as performed in Garfinkel (2007) and Pal et al. (2008).

The ability to identify clusters of data blocks that are similar to an encoded file in terms of the way they are entropy coded offers two advantages. First, rather than randomly decoding many blocks to identify a starting block of a fragment only the most likely ones determined based on bit sequence matching will be decoded. Second, even if a few bytes of a file are missing decoding based fragment identification will not identify the incomplete fragment and fragments following it. In such cases, bit sequence matching can be deployed to identify likely fragments of a file. Below we describe the construction of bit patterns to be used during matching and provide some results.

4.1. *Bit pattern construction*

The premise of our approach is that if the occurrence frequency of a given n-bit pattern in an m-bit sequence one should expect to encounter roughly $\frac{m}{2^n}$ matches (assuming very large m). However, if the m-bit sequence is not random, then for certain n-bit patterns we should expect a bias which will reflect as a deviation from the expected number of random matches. Constructing bit patterns that can be detected in a file more times than expected through random chance requires that file have a structure. Since JPEG uses 4 Huffman code tables (two for luminance coefficients and two for chrominance coefficients) which are most generally not optimized to match statistical properties of the data, resulting JPEG bit sequences will have a structure. In order to capture the degree of non-randomness, frequency characteristics of bit patterns constructed from Huffman code words can be used. It should be noted that the two other components of a JPEG file, i.e., markers and parameters, cannot be used for this purpose. Parameters are typically at the header part and they are easily identified at earlier stages of recovery and global file markers can only be used distinguish JPEG files from other files.

Another reason for JPEG files having a structure is that they are generated by repetitively coding MCUs, which are the fundamental building blocks of all JPEG files (Hass). Each MCU correspond to a number of 8×8 luminance (Y) and chrominance (Cb and Cr) blocks of the image which is determined by the rate of chroma sampling. To encode each individually compressed color component a different Huffman code table is used. One notable detail is that each color component is composed of two different types of coefficients, namely, DC and AC coefficients, and due to their statistical properties different code tables are used in their coding. During encoding of image blocks, MCU structure is preserved, and it is ensured that encoding sequence always starts by Y-DC and Y-AC components followed by DC and AC coefficients of Cb and Cr components. Fig. 3 displays the MCU coding sequence for three rates of chroma subsampling.

The arrangement of coefficients in Fig. 3 shows in what order code words selected from the four Huffman look-up tables will appear in the final bit sequence. Moreover, to prevent ambiguities when differentiating code words from different fields, end of block (EOB) codes are available. However, potential use of EOB codes in constructing bit patterns is limited. For DC coefficients they are usually not needed and for AC coefficients they have to be used only when the sequence of AC coefficients ends prematurely because many of them are all zeros. Luckily, the latter condition holds most of the time. Another consideration is the length of the chosen bit patterns. Since shorter bit patterns will yield many random matches, even if there is a bias due to structure of the file that might not be noticeable (as JPEG bit sequences are still more random than not). Therefore, larger bit patterns have to be selected.

4.2. *Results*

We performed a set of experiments to test the premise that bit patterns constructed from a given Huffman table can be used to identify files created using the same Huffman table. Bit patterns are generated in two different forms using the Huffman tables extracted from baseline JPEG images with no chroma subsampling. That is, each MCU consists of only 3 blocks, e.g., Y, Cb and Cr blocks.

In the first case, we selected the EOB code that follows AC coefficients of the luminance component (Y-AC EOB). For this purpose, we considered three images with different Huffman tables in which Y-AC EOB code word are defined as ‘1010’, ‘1000’ and ‘11000’. Then, the number of occurrence of each code word in all three files are computed. We observed for all bit patterns that measured frequencies, computed as number

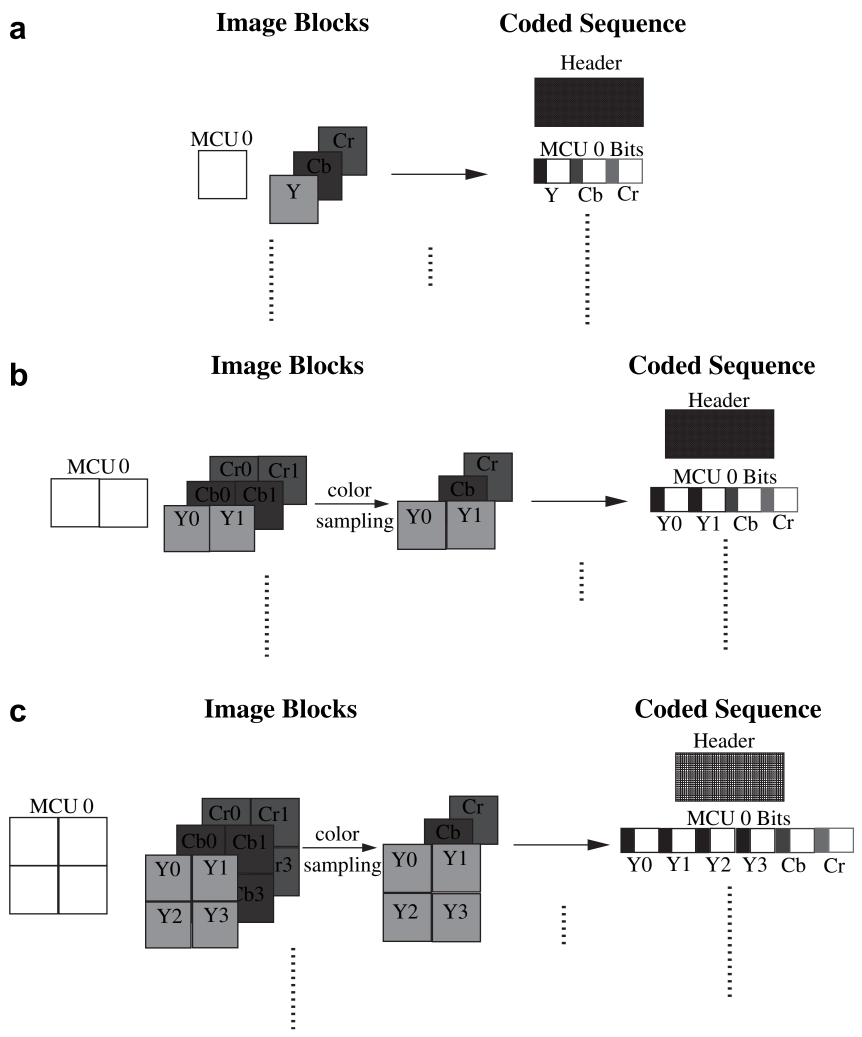

Fig. 3 – Encoding of MCUs under different rates of chroma subsampling. (a) No chroma subsampling is performed. (b) Horizontal frequency is halved. (c) Both vertical and horizontal frequencies are halved.

of matches divided by the size of the file, closely follows the expected values, i.e., $1/2^4$ and $1/2^5$, in all cases. This result is in line with our intuition that for short patterns the bias, i.e., deviation from expected value, will be too small to be observed.

In the second case, we aimed at constructing longer bit patterns by choosing code words from three of the four Huffman tables denoted by Y-DC, Y-AC, CbCr-DC, and CbCr-AC. Following Fig. 3, the pattern is generated as a sequential combination of Y-AC EOB code, a CbCr-DC code word, and a CbCr-AC code word. For this purpose, a set of three larger sized images are used. We constructed all possible 9- to 16-bit patterns that confirm to above arrangement of codewords by appropriately selecting codewords from the extracted CbCr-DC and CbCr-AC Huffman tables. Then, the occurrence frequencies of all generated bit patterns in all the image files are computed. Results show that as the length of bit pattern increases from 9–16 the ability of the bit patterns to identify the file with the same Huffman table increases, and, in fact, above 14-bit each file is identified correctly with an observable bias in the measured frequencies. Tables 1 and 2 provide the measured frequencies for 15-bit and 16-bit patterns in the three images.

It must be noted that the measured values are obtained as the average of all possible bit-patterns in that length. The reason we expect to observe an increase in the measured frequency as compared to the expected frequency (i.e., $1/2^n$ for an n-bit pattern) is that the code words that make up the pattern are indeed used in encoding of the file which make them more likely to match with the searched pattern as compared to

Table 1 – Averaged occurrence frequencies of all 15-bit patterns in images A, B and C. Patterns are generated from three different sets of Huffman tables, i.e., H_A, H_B and H_C. Each row provides the frequencies of three sets of patterns in the same image.

	H_A patterns	H_B patterns	H_C patterns
Image A	3.07E−05	2.38E−05	2.36E−05
Image B	2.89E−5	3.39E−05	3.25E−05
Image C	2.88E−05	2.54E−05	3.19E−05

Table 2 – Averaged occurrence frequencies all of 16-bit patterns in images A, B and C. Patterns are generated from three different sets of Huffman tables, i.e., H_A, H_B and H_C. Each row provides the frequencies of three sets of patterns in the same image.

	H_A patterns	H_B patterns	H_C patterns
Image A	1.46E−05	5.87E−06	1.29E−05
Image B	9.93E−06	1.5E−05	1.23E−05
Image C	1.2E−05	1.25E−05	1.49E−05

a random pattern. (Had we averaged over all 2^n possible n-bit patterns, we would not see an increase.) To more reliably quantify the significance of the bias, large scale experiments have to be conducted and statistical tests have to be performed.

A related issue is the diversity of Huffman tables used in encoding of images. If all images were to be encoded with the same set of Huffman tables, then the described approach here would not offer any advantage. However, many digital cameras are known to use their own Huffman tables and image editing software tools typically generate Huffman tables optimized for a given image. Therefore, even if this approach may not be used in identifying fragments of individual images (as many images may use the same set of tables), it will, at worst, provide a differentiation of fragments at a class level.

5. Recovery of fragmented files with missing data

In this section, we address the problem of file recovery under two scenarios. First concerns file fragments for which there is no available file header, and the second concerns file fragments that follow a missing piece of a file. In Fig 2-b, two fragments of file C falls into former category, and the second fragment of file E (spanned over blocks 28–30) into the latter case. Both cases, however, pose a challenge to current techniques. Recovery of JPEG files (fragmented or not) requires that an image header be present. This is because all the necessary information needed by a decoder to interpret a JPEG file is stored in the file header. Therefore, fragments that cannot be linked to a known header are not considered for recovery. Moreover, since a decoding has to follow the structure defined by JPEG file format, any disturbance or corruption of the file structure will prevent the decoding of file data. Hence, disruption of the continuity of the file data will cause decoding errors, and fragments that are beyond the disruption point will not be able to recovered.

Essentially, the problem of recovering disrupted file fragments is a special case of the more general headerless file recovery problem. But, since restart markers defined by the JPEG standard can be utilized in addressing the former problem, they will be discussed separately.

5.1. *Recovery of disrupted fragments*

In the JPEG standard, restart markers are provided as a means for detection and recovery after bitstream errors. There are eight unique restart markers and each is represented by a two byte code (0xFFD0–0xFFD7). They are the only type of marker that may appear embedded in the entropy-coded segment;

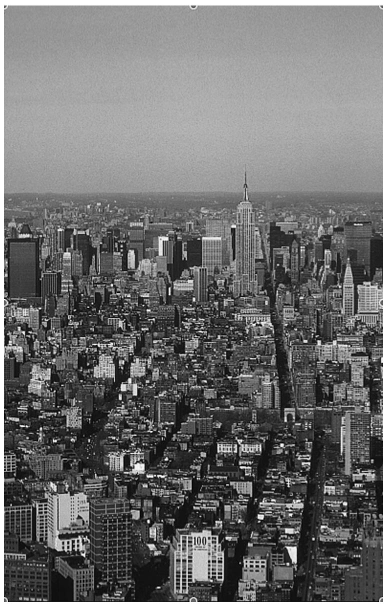
Fig. 4 – Original JPEG file.

therefore, they can be directly searched in the file data. Restart markers are inserted periodically in the data and they repeat in sequence from 0 to 7, as indicated by the value of the marker code. The number of MCUs between the markers has to be defined in the (DRI marker segment of the) file header. Although insertion of restart markers is optional, they are generally used in coding of large sized images.

In JPEG files, DC coefficients of all color components are encoded as difference values rather than as absolute values. When a restart marker is hit, this DC difference is reset to zero and the bitstream is synchronized to a byte boundary. In other words, the runs of MCUs between restart markers can be independently decoded. Also, since restart markers are placed in sequence, in the case of a bitstream error decoder can compute the number of skipped MCUs with respect to the previous marker and determine where in the image the decoding should resume.

These properties make restart markers potentially very useful in recovering disrupted fragments. These fragments can be quite reliably identified due to unique restart marker codes appearing periodically. However, the main problem that remains to be addressed is the identification of the file header. To accomplish this, one can utilize the approach described in Section 3. Since header information for all the partially recovered files are available, one can generate appropriate bit patterns and search for fragments that are more likely to be generated using the same Huffman code tables. Then, starting from the first restart marker on, disrupted fragment can be decoded using one of the headers from those files or can be merged to the first fragment of those files and then decoded. In any case, decoding will succeed only for the matching file.

To assess the potential use of restart markers in recovering disrupted fragments, we simulated different fragmentation scenarios. For this purpose, random chunks of data are erased from the tail, center, and both header and tail parts of the original JPEG file displayed in Fig. 4. In the bitstreams

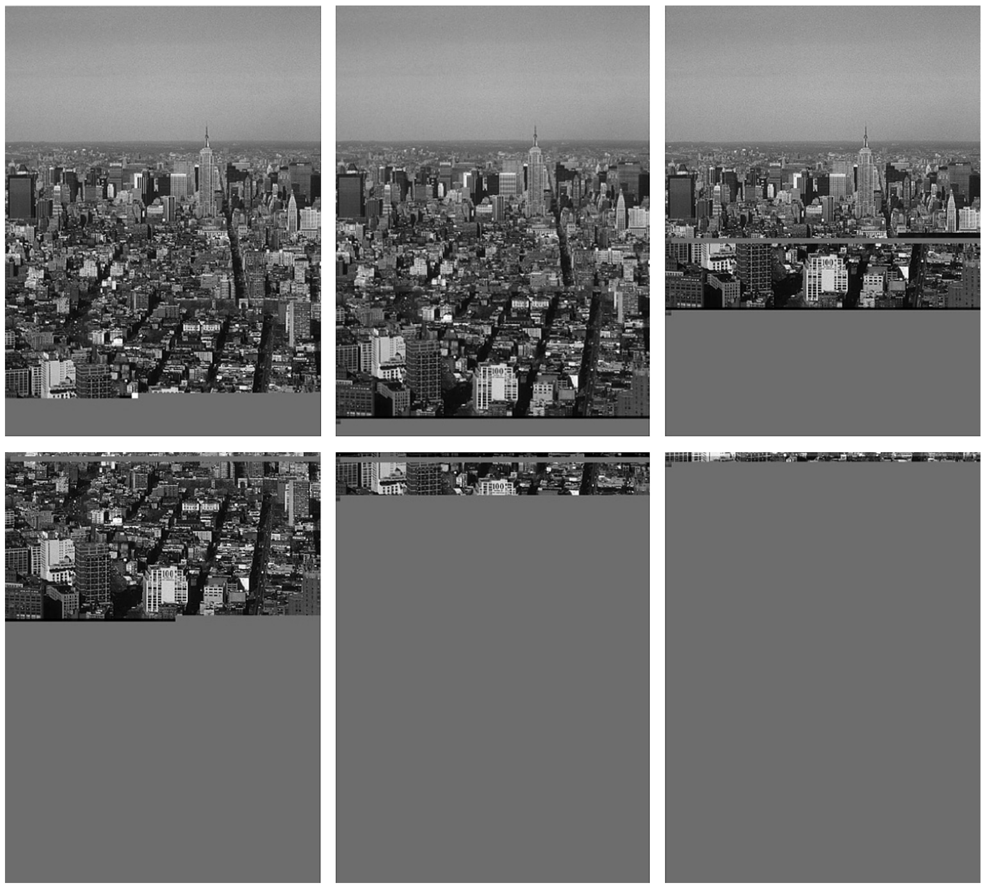
Fig. 5 – Recovered files after erasure of random amounts of data from tail (upper left), center (upper center and right), and both header and tail parts (lower row) of the original image.

corresponding to deleted files, the restart markers are searched. After identifying any of the seven restart markers, all the bits prior to marker position are discarded and resulting data is merged with the first part of the file or with the header extracted from the original JPEG file and decoded. Recovered files are displayed in Fig. 5. It can be seen that fragments of the original file can be successfully recovered. It should be noted that because the stored image size in the header is not modified, in all cases images appear in the right size, but the content is shifted.

5.2. *Recovery of stand-alone fragments by use of pseudo headers*

Obviously without a valid header, a JPEG file or a part of it cannot be decoded. Given this fact, in this section, we pose the question of what information one will need to reconstruct a pseudo header that can be utilized in decoding of a stand-alone file fragment. The information that can be inferred by analysis of encoded file data will not be sufficient to reconstruct a file header. Our premise is that image files stored on a recovery medium will be interrelated to some extent. This relation may exist because images may have been captured by the same camera, edited by the same software tools, or downloaded from the same Web pages. All these factors induce different levels of shared information among the neighboring files in terms of their encoding properties which may vary from image quality settings to specifications of the encoder. Therefore, in essence, we will investigate the possible use of encoding related information from recovered files in recovery of stand-alone fragments.

Considering only baseline JPEG/JFIF images, the most common JPEG encoding method used by most digital cameras and on the Web, the information needed to encode/decode an image can be categorized into four types. These are:

1. the width and height of the image specified in number of pixels;
2. the 8×8 quantization tables used during compression;
3. the number of color components and type of chroma subsampling used in composition of MCUs; and
4. the Huffman code tables.

Decoder essentially needs image size so that the number of MCUs can be computed and the image blocks obtained by decoding of each of the MCUs can be laid out at their proper locations on the image. Since the encoded values are not the quantized values, but the associated quantizer bin values, quantization tables are needed to perform de-quantization prior to inverse-DCT transformation. The composition of

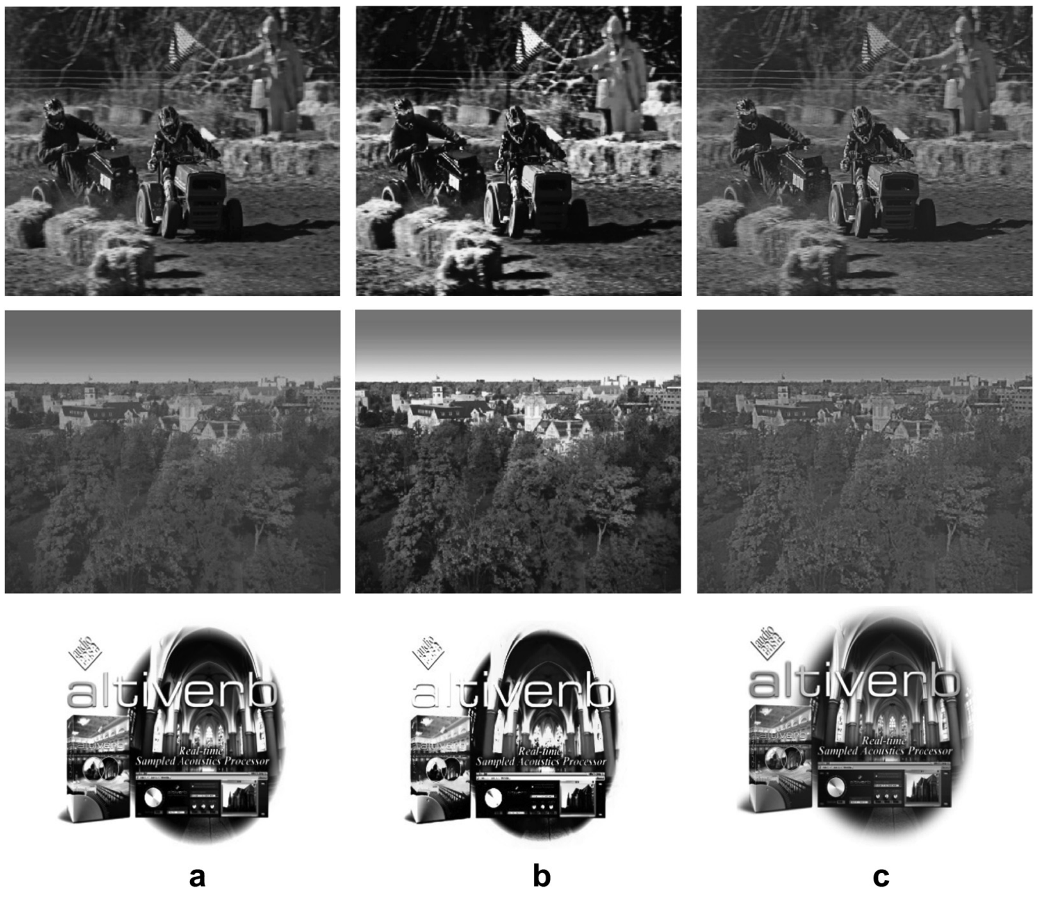

Fig. 6 – (a) Images compressed using the quantization tables obtained from the image in the first row. (b) Images compressed using the quantization tables obtained from the image in the middle row. (c) Images compressed using the quantization tables obtained from the image in the third row.

MCUs provide decoder with the ability to decide in what order Y, Cb and Cr components will be decoded. Decoding is implemented as a simple table look-up operation where the decoder looks up a code word in the appropriate Huffman code tables to determine the actual bit representation, i.e., Y-DC, Y-AC, CbCr-DC and CbCr-AC.

Another issue that needs to be discussed is the exactness requirements on this four type of information. Essentially, first two types of information are not critical elements for successful decoding. If the selected image size do not match with the actual one, image blocks will appear shifted in the resulting image. Due to nature of these shifts the resulting image might not be smooth and recognizable, but it will nevertheless not be filled with random data.

Similarly, compression information primarily impacts the visual quality of the resulting image, and a mismatch in the used tables will only result with a visually degraded image. Fig. 6 displays a decoding scenario where three images are decoded also using quantization tables from the two other images. It can be seen that despite the degradation, in all cases, important image details are preserved. Therefore, even if image size and quantization tables do not match exactly with the actual ones as long as MCU composition and Huffman tables match recovery can be accomplished.

To test the validity of this approach, we performed the following experiments. In the first one, we consider three gray-scale images with two different sizes and different quantization tables but with the same Huffman tables. The images will be referred to as A, B and C and are displayed in Fig. 7. Fig. 8 displays decoding results for the case when file A data is decoded using headers of images B and C. In both cases decoding is performed successfully. However, due to mismatch between the sizes of A and B, decoder placed blocks by shifting their locations. It must be noted that since sizes of A and C are the same, decoding A with C's header will succeed. Fig. 9 shows another decoding scenario where file A's data and C's data are combined and decoded using a header from an image with much larger size, but still using the same Huffman tables. In the decoded image, it can be seen that A's data is tiled across the larger sized image, but C's data caused loss of synchronization and it is not decoded properly. Since there were no restart markers, the synchronization could be established until the end of data.

The composition of MCUs and specification of the Huffman tables are the most critical aspects of successful decoding. A mismatch in MCU size and Huffman tables will cause an immediate decoding error that can only be recovered by providing the correct values. Three most typical subsampling

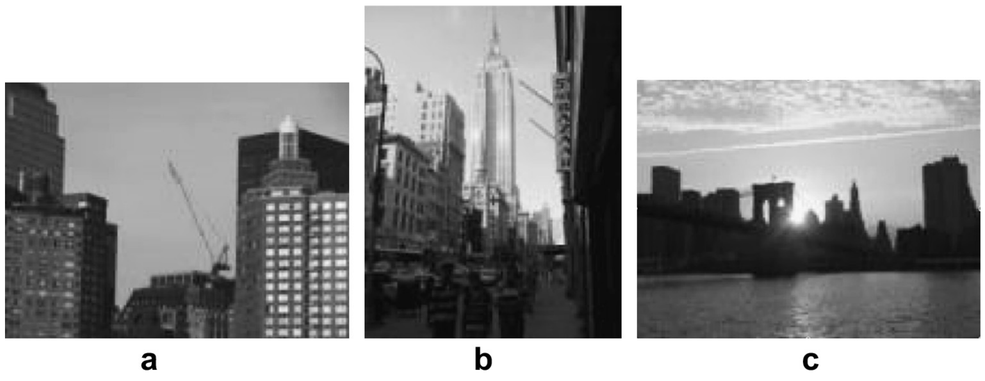

Fig. 7 – Images (a) A with size 160 × 120 pixels, (b) B with size 120 × 160 pixels, and (c) C with size 160 × 120 pixels are generated using the same Huffman tables.

configurations lead to use of MCUs with 8 × 8, 8 × 16 and 16 × 16 pixels in size. In coding of gray-scale images since there is only Y component the default MCU size is 8 × 8 pixels. In color images, however, to achieve better compression, MCU size is most generally set to 16 × 16 pixels. Therefore, as long as Huffman tables are known, MCU size can be easily determined by trying all possible configurations.

To exploit the MCU composition of an image during recovery, we performed an experiment similar to described in Section 4.2. Chunks of data were removed from the header and tail parts of the image given in Fig. 4 such that remaining data started from the middle part of the first MCU following a restart marker. In Section 4.2, it was shown that if the data at the beginning of the disrupted fragment is discarded until the first restart marker, it can be decoded successfully. In this case, however, we tried to identify an MCU boundary, rather than a restart marker. Noting that an MCU starts with a code word from Y-DC Huffman table, we tried to identify the boundary by first searching for a likely Y-AC EOB code in the beginning of the file, which will be at most 3 bytes away from the MCU boundary. Then, the file is attempted to be decoded by changing the starting bit positions. Fig. 10 shows two versions of the recovered file using the above search procedure.

As a result, it can be asserted that the ability to recover stand-alone file fragments reduces to ability to determine the underlying Huffman tables. If the Huffman tables are customized for the image then recovery is bound to fail. However, many encoders use precomputed tables or the suggested tables by the JPEG standard. When combined with the approach described in Section 3, this makes it possible to identify whether or not the stand-alone fragment is generated by a set of Huffman tables of other recovered images or by any of the known Huffman tables.

6. Conclusion

In this paper, we proposed an approach to improve the performance of existing fragmented file recovery techniques. Our contributions are two-fold. First, we proposed a method to identify the next fragment of a file more efficient than random decoding approach. Second, we demonstrated methods for

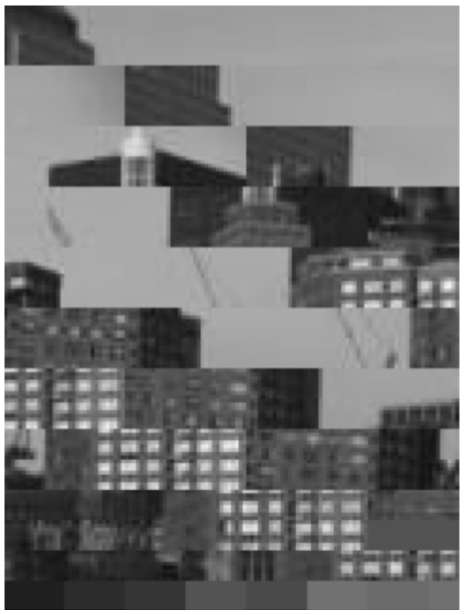

Fig. 8 – Result for decoding image A with image B header.

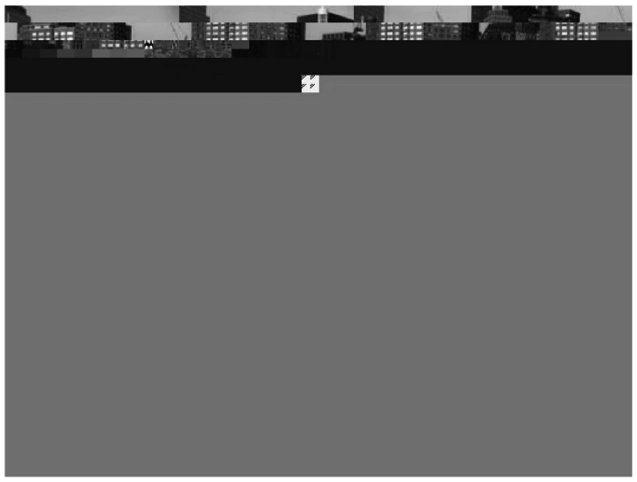

Fig. 9 – Result for decoding File A and C data (merged together) using a header of a larger image.

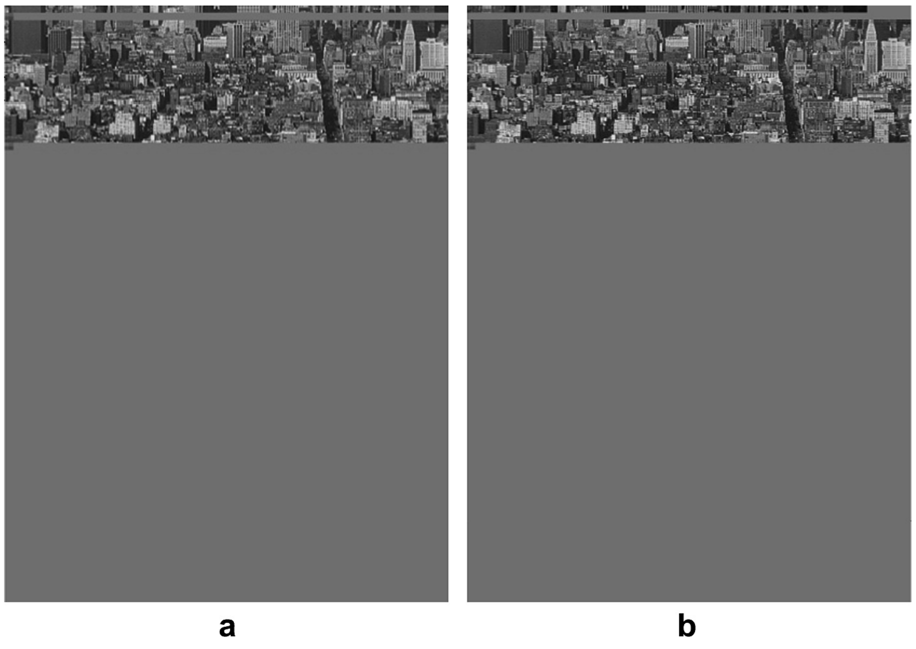

Fig. 10 – Recovered images from a disrupted file fragment through adjusting MCU boundary. Image in (b) is one MCU shorter than image in (a).

recovering disrupted file fragments and investigated the possibility of constructing a pseudo header for recovery of stand-alone file fragments. Additional work will be done on a more diverse data set to establish the generality of results.

We finally thank Chander Mohan, a senior student at IIT Kanpur, India, for helping us with the experiments during his summer internship.

REFERENCES

Garfinkel S. Carving contiguous and fragmented files with fast object validation. In: Proceedings of the 2007 digital forensics research workshop, DFRWS, Pittsburgh, PA, August 2007.

Hass C. Designing a JPEG decoder http://www.impulseadventure.com/.

Miano J. Compressed image file formats: JPEG, PNG, GIF, XBM, BMP. ACM Press; 1999.

Pal A, Shanmugasundaram K, Memon N. Reassembling image fragments. In: Proceedings ICASSP, Honk Kong, April 2003.

Pal A, Memon N. Automated reassembly of file fragmented images using greedy algorithms. In: IEEE transactions on image processing, February 2006, pp. 385–93.

Pal A, Sencar HT, Memon N. Detecting file fragmentation point using sequential hypothesis testing. In: Proceedings of the 2007 digital forensics research workshop, DFRWS, Florida, August 2008.