Carving Orphaned JPEG File Fragments

Article *in* IEEE Transactions on Information Forensics and Security · March 2015

DOI: 10.1109/TIFS.2015.2416685

2 authors, including:

Erkam Uzun
Georgia Institute of Technology
13 PUBLICATIONS **70** CITATIONS

Carving Orphaned JPEG File Fragments

Erkam Uzun, Hüsrev T. Sencar

Abstract

File carving techniques allow for recovery of files from storage devices in the absence of any file system metadata. When data are encoded and compressed, the current paradigm of carving requires the knowledge of the compression and encoding settings to succeed. In this work, we advance the state-of-the-art in JPEG file carving by introducing the ability to recover fragments of a JPEG file when the associated file header is missing. To realize this, we examined JPEG file headers of a large number of images collected from Flickr photo sharing site to identify their structural characteristics. Our carving approach utilizes this information in a new technique that performs two tasks. First, it decompresses the incomplete file data to obtain a spatial domain representation. Second, it determines the spatial domain parameters to produce a perceptually meaningful image. Recovery results on a variety of JPEG file fragments show that given the knowledge of Huffman code tables, our technique can very reliably identify the remaining decoder settings for all fragments of size 4 KiB or above. Although errors due to detection of image width, placement of image blocks, and color and brightness adjustments can occur, these errors reduce significantly when fragment sizes are larger than 32 KiB.

Index Terms

File Carving, JPEG, Huffman Synchronization

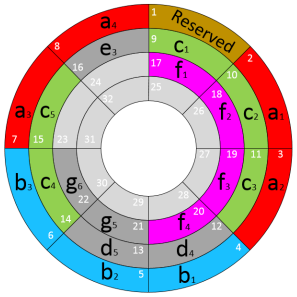
E. Uzun and H. T. Sencar are with TOBB University, Ankara, Turkey and New York University Abu Dhabi, UAE. e-mail: {*euzun, htsencar*}@etu.edu.tr

Carving Orphaned JPEG File Fragments

I. Introduction

With more and more data created and stored across countless devices, there is a growing need for techniques to recover data when the file system information for a device is corrupt or missing. An essential part of the response to this challenge is widely known as file carving as it refers to process of retrieving files and fragments of files from storage devices based on file format characteristics, rather than using file system metadata. Today, file carving tools play an important role in digital forensic investigations where carving for deleted files and salvaging of data from damaged and faulty media are common procedures.

All file systems store files by breaking them into fixed-size chunks called blocks, that vary in size from 0.5 KiB for the older systems up to 64 KiB for the latest high storage volumes. When a file is accessed, its data are retrieved in sequence from this list of blocks, which is often referred to as a file allocation table. Similarly, deletion of a file is typically realized by removing a file's entry from this table. Figure 1 depicts a layout of files distributed across blocks of a storage media and provides a simplified view of a file allocation table.



File Name	File Segments	Allocated
File a	2-3, 7-8	✓
File b	4-6	✓
File c	9-11, 14-15	✓
~~File d~~	~~9-13~~ (12-13)	X
~~File e~~	~~14-16~~ (16)	X
File f	17-20	✓
~~File g~~	~~17-22~~ (21-22)	X
Free	23-32	▪

Fig. 1. Distribution of data blocks on a storage volume and the corresponding file allocation table.

In the absence of a file allocation table, the ability to recover file data involves two main challenges. The first is due to inability to lay out file data blocks contiguously on the storage. When using conventional disk drives, repeated execution of file operations, like addition, deletion and modification of files, over time will yield to fragmentation of available free storage space. As a result, the newly generated files needs to be broken up into a number of fragments to fit in the available unallocated space. (This phenomenon is illustrated in Fig. 1 where files a

and c are both fragmented into two pieces.) Solid state drives, which are now in widespread use, are designed to emulate interface characteristics of hard disk drives and also behave like block devices. Therefore, they are also susceptible to file fragmentation. The most important implication of file fragmentation phenomenon is that, for many files, their recovery will require identification of file fragments that are potentially distributed across the storage.

The second challenge stems from the fact that data are typically stored in binary format to provide faster and flexible access to it, save storage space, achieve error correction and allow for encryption. All these require deployment of specialized interpreters (decoders) to handle file data. Thus, without decoding, a block of file data will reveal little or no information about the content of a file.

When compounded with the file fragmentation problem, binary coding of data renders many file fragments unrecoverable. Because of this difficulty, conventional file carvers utilize known file signatures, such as file headers and footers, and attempt to merge all the blocks in between them when carving encoded files [1]–[4]. Although this approach is successful in carving contiguously stored files (like files e and f), in other cases, it can only recover the first fragment of a fragmented file, *i.e.*, from the start of the file up to the point at which disruption has occurred, like the first two and three blocks of files e and f, respectively.

Fragmented file carving problem has received considerable research interest and a number of techniques have been proposed [5]–[9]. These techniques have primarily focused on recovery of fragmented JPEG images due to their higher prevalence in forensic investigations on digital media. In essence, these techniques try to identify fragmentation points in the recovered file data during JPEG decoding. A fragmentation point is deemed to be detected when the decoder encounters an error and declares a decoding failure or start generating erroneous data that cause abrupt changes in image content. These techniques then utilize disk geometry to search for the next fragment of a file by repetitively merging new data blocks with the correctly identified partial image data and validating the decoded data. With these techniques, if a fragment of a file is not available, the file can only be recovered up to the missing fragment.

In fact, state-of-the-art in file carving assumes that a file header is always present. Since the header marks the start of a file and includes all the encoding parameter settings needed for decoding, it is very critical for the recovery. Consequently, if a file is partially intact with its header deleted, existing techniques cannot recover any data even though the rest of the file data

may be intact. In most file systems, when a file is deleted no data are ever really deleted, just overwritten over time by newly saved files. Hence, on an active system, it is very likely that many fragments of previously deleted files, which may not necessarily include the file header, would have remained intact, like the fragments of files d, e and g which were overwritten by files c and f as shown in Fig 1.

In this paper, we address the challenging problem of recovering *orphaned JPEG file fragments*, arbitrary chunks of JPEG compressed image data without the matching encoding metadata. The underlying idea of our approach is to reconstruct a file header for decoding the orphaned file fragments. Since JPEG images are created mostly by digital cameras, we examined a large number of images publicly available on Flickr photo sharing website to obtain knowledge on encoding settings. (Several other studies analyzed JPEG file headers of Flickr images to determine image veracity [10], [11].) Our carving approach incorporates this information and proposes a new carving technique to both decompress incomplete file data and estimate the parameters necessary for correct rendering of the recovered image data.

In the next section, we provide an overview of JPEG encoding standard from this perspective and describe challenges in reconstructing a header for a given orphaned fragment. In Section III, we provide our findings on Flickr dataset and discuss how they can be utilized in tailoring a file header. Section IV describes in detail individual steps of our proposed file carving technique. Carving results obtained from our experiments and publicly available test disk images are given in Sec V. We conclude the paper with a discussion of the results and the implications of our work for practice of digital forensics.

II. JPEG Encoding and Orphaned File Fragment Carving Challenges

JPEG is the most widely adopted still image compression standard today despite having been superseded by a new standard (JPEG 2000). It is the default file format for most digital cameras, and it is supported natively by all contemporary browsers. The JPEG standard specifies JPEG Interchange Format (JIF) as its 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), which is a minimal format, and exchangeable image file format (ExIF), which additionally provides a rich set of metadata, are the two most common formats used for storing and transmitting JPEG

images [12], [13].

The JPEG standard defines four compression modes: lossless, baseline, progressive and hierarchical. In baseline mode, image data are stored sequentially as one top-to-bottom scan of the image, and this mode is required as a default capability in all implementations compliant with the JPEG standard. By contrast, the progressive mode allows progressive transmission of image data by performing multiple scans, and the hierarchical mode represents an image at multiple resolutions. Baseline mode is by far the most-often-used JPEG mode, and what is commonly referred to as JPEG is the baseline encoding.

JPEG (baseline) compression is simple and requires very low computational resources. JFIF specifies a standard color space called `YCbCr` for encoding color information, where the color component `Y` (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 the horizontal, vertical or both the horizontal and vertical directions. During compression `Y`, `Cb` and `Cr` components are processed independently.

JPEG image compression is essentially based on discrete cosine transform (DCT) coding with run-length and Huffman encoding [14]. Following the color conversion and subsampling, 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 zig-zag scanning within the transformed block and are passed through a lossless entropy coder to remove any further redundancy. For decoding, these steps are performed in reverse order.

JPEG file data contain two classes of segments. The marker segments contain general information about the image, camera settings and all the necessary information needed for decoding the image. Each segment is identified with a unique two-byte marker that begins with a `0xFF` byte followed by a byte indicating the type of the marker, length of the payload, and the payload itself. The entropy coded data segment contains the compressed image data. Since this segment is much larger in size, file carving operations are most likely to encounter fragments of entropy-coded data. Therefore, understanding of the structure of the encoded data is of utmost importance

in developing a file carving technique for orphaned JPEG file fragments.

Every JPEG image is actually stored as a series of compressed image tiles that are 8×8, 8×16, 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 crucially determined by the chroma subsampling. (Figure 2 illustrates the four chroma subsampling schemes utilized during JPEG encoding.) Each MCU is then encoded as a sequence of its luminance data followed by the chrominance data.

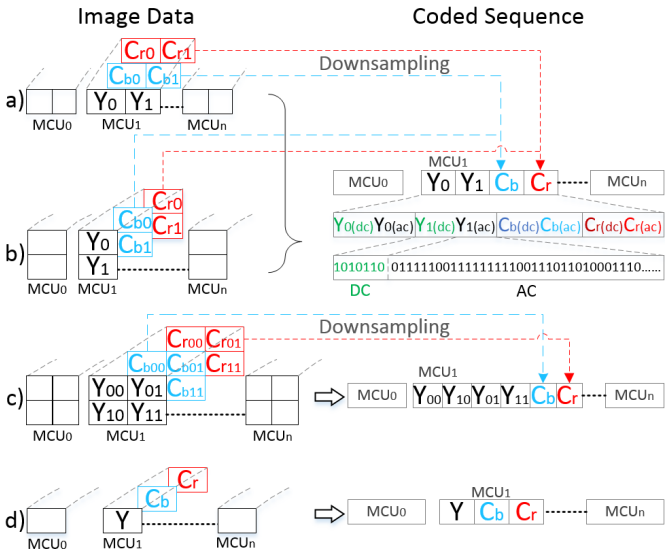


Fig. 2. Four different chroma subsampling schemes and how each MCU a) horizontal sampling, b) vertical sampling, c) horizontal and vertical sampling, and d) no-sampling.

Entropy coding is performed on a block-by-block basis by compressing each block in an MCU consecutively. The first step of entropy coding involves run-length coding of zeros in the zig-zag scanned block of (quantized transform) coefficients by replacing the runs of zeros by a single value and the count. Then the resulting run-length encoded coefficients are organized into a so called intermediate code, where the representation for each non-zero coefficient includes a run-length value, the number of bits in the non-zero coefficients, and the value of the coefficient. In the second step of entropy coding, the run-length and bit-length values associated with each coefficient in the intermediate code are mapped to symbols, and the resulting symbols are individually Huffman encoded. Huffman encoding essentially works by substituting more

frequently used symbols with shorter bit sequences, referred to as codewords, that are stored in a pre-defined code look-up table. Figure 7(a) illustrates the entropy coding steps for an 8×8 block of transformed and quantized coefficients. It must be noted that quantization operation is likely to introduce many zero-valued coefficients towards the end of zig-zag scan; therefore, an end of block codeword (`EOB`) is used to indicate that the rest of the coefficient matrix is all zeros.

One noteworthy detail is that each of the `Y`, `Cb` and `Cr` components are composed of two different types of coefficients, namely, `DC` and `AC` coefficients, where the `DC` coefficient corresponds to average intensity of an 8×8 component block while the 63 `AC` coefficients represent variations at different spatial frequencies, thereby providing information about the level of detail in the block. Due to the different statistical properties, `DC` and `AC` coefficients are encoded using different code tables. Similarly, luminance and color components use two different sets of Huffman code tables. During encoding of image blocks, MCU structure is preserved, and the encoded sequence always starts by coefficients of `Y` component(s) followed by `Cb` and `Cr` component coefficients. (The structure of the coded sequence for different types of MCUs is shown in Fig. 2.) The output bitstream of a JPEG compressed image is constructed by mixing codewords from four different Huffman code tables where each codeword is followed by the associated coefficient value.

Designing an automated orphaned file fragment carver requires solving two main problems. First, file fragment data need to be decompressed. This requires correctly identifying compression mode, compression parameters (*i.e.*, Huffman tables) and chroma subsampling configuration (*i.e.*, MCU structure). However, there are further complications that need to be taken into account. Deletion (or loss) of file data essentially means that decoding will have to start at some arbitrary bit in the compressed bitstream, which may not correspond to a (Huffman) codeword boundary. This is further exacerbated by the fact that JPEG encoder stuffs encoded bitstream with `0x00` bytes, which are meant to be ignored during decoding, after any time a `0xFF` byte is encountered to prevent interpretation of encoded data as marker codes by the decoder.

Once entropy decoding is successfully performed by correctly identifying the MCU structure and decoding the component coefficients, the second problem that needs to be overcome is correct rendering of the partial image data. This involves estimation of missing parameters like image width and quantization tables by utilizing the recovered data. Deviations from the original image width, combined with the shifts in image blocks due to missing image data, will result in

incorrect arrangement of the image blocks. Similarly, a mismatch in the used quantization table and the errors in the DC coefficients (which are incrementally coded with respect to preceeding blocks) will introduce brightness and color anomalies that must be repaired.

Obviously, entropy decoding of the file fragment data is more challenging as it may require trying large number of possible compression settings. Essentially, the complexity of this step will be mainly determined by the difficulty in identifying the correct set of Huffman tables. Although Huffman tables can be individually customized for each image, thereby making recovery extremely difficult if not impossible, it is very likely that digital cameras do not perform such optimizations but rather utilize default sets of tables determined by each manufacturer. Therefore, we will focus on the latter scenario where Huffman tables are not custom built for each image.

The task of reconstructing a new header can be further alleviated. In practice, it is reasonable to assume that images in a storage 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. This information, when available, can be utilized to guess the missing information needed for decoding. Alternatively for the general case, one can examine a large set of images to determine what parameters and settings are commonly used by different camera makes and models. Focusing on this latter setting, in the next section, we present our findings on images obtained from Flickr photo sharing website and discuss how they can be utilized in tailoring a header to recover an orphaned file fragment.

III. JPEG Encoding Setting Statistics for Digital Cameras

To obtain reliable statistics on JPEG encoding settings, we first compiled a list of all digital camera makes and models that are attributed to photos uploaded to Flickr by using the available Flickr API methods. This yielded to 42 different manufacturers and 2653 different camera, mobile phone and tablet models. Using the 2653 model names as keywords, we searched Flickr image tags of more than 6 million publicly accessible Flickr user photos. To ensure encoding settings reflect those set by the capturing device, and not those potentially modified or reconstructed by an image editing software (which may re-encode images with a custom-built header information), we applied a set of filters to the searched image tags prior to downloading images. (It must be noted that Flickr API only provides access to part of the photo metadata and not the complete ExIF data.) These filtering rules were implemented similar to [10] as follows.

- Only images in JPEG format and tagged as 'original' were retained.
- Images not stored in ExIF file format and with no or very sparse metadata were eliminated.
- Images with less than three color channels were eliminated.
- Images with inconsistent 'modification' and 'original' dates were eliminated.
- Images in which software metadata tags (that records the name and version of the software or firmware of the device used to generate the image) contain one of the 300 entries (that indicate processing with photo editing tools and applications) were eliminated[1].
- All search entries that led to less than 25 images with the same make and model were eliminated.

Filtered search results provided us with $767,916$ images. These images are downloaded and the information embedded in the original ExIF headers are extracted. The resulting data are first analyzed to obtain compression mode characteristics. It is determined that 99.51% of the images are encoded in baseline mode while progressive mode and extended sequential mode (which refers to baseline mode with 12-bit sample resolution) are only used in encoding of 3762 and 3 images, respectively. No images were found to be encoded in lossless or hierarchical modes. This finding further establishes the widely held belief that baseline mode is the most common compression mode.

To further eliminate the images potentially modified by photo editing software, ExIF metadata of all the images are searched for the same 300 keywords and phrases. This reduced our database to $460,643$ images captured by all camera models. Further analysis have been performed on the retained images to extract statistics on Huffman tables, image dimensions, quantization tables, quality factors, image sizes, chroma subsampling schemes and frequency of restart markers.

Diversity of Huffman tables: Noting that a set of Huffman tables include four different tables for encoding `DC` and `AC` coefficients of luminance and chrominance components, we observed 5963 distinct sets of Huffman tables in our dataset. However, 98.61% of these images $(454,216)$ were determined to use the same set of tables, recommended by the JPEG standard [15]. The remaining 5962 Huffman table sets were utilized by 6425 images, indicating that some tables were potentially customized for the image content.

[1]See Appendix A for the list of keywords and phrases used for filtering out potentially modified images.

Utilized chroma subsampling schemes: Table II shows the frequencies with which each chroma subsampling scheme was used in our image set. Out of the four subsampling schemes, horizontal sampling has been the most common choice (84.2%) followed by the horizontal and vertical sampling scheme (10.7%). The other two schemes were less common but still used occasionally. In our image set, only 24 manufacturers demonstrated all four subsampling schemes, while the rest of the manufacturers did not exhibit the use of the no-sampling scheme.

Diversity of image dimensions: The resolutions of images in our dataset varied from 0.018 megapixel to over 53 megapixels. After eliminating image dimensions that appeared only once in our image set (which might be seen as an indicator of image modification), we identified 1331 distinct dimensions in total for all chroma subsampling schemes as summarized in Table I. In the table, min_w, max_w, min_h and max_h are used to denote the minimum and maximum widths and heights in pixels while min_r and max_r denote minimum and maximum image resolutions in megapixels.

TABLE I
DIVERSITY OF IMAGE DIMENSIONS IN PIXELS

Subsamp. Scheme	min_w	min_h	min_r (mpix)	max_w	max_h	max_r (mpix)	# of distinct dimensions
Horizontal	160	120	0.018	12416	9420	53.785	646
Vertical	240	320	0.073	3744	5616	20.052	124
Hor.&Ver.	160	120	0.018	10800	7728	36.613	630
No-Samp	315	337	0.142	5572	5699	20.370	239

Presence of restart markers: Restart markers are the only type of marker that may appear embedded in the entropy coded segment. They are used as a means for detecting and recovering bitstream errors and are inserted sequentially after a preset number of MCUs, specified in the header. Restart markers are always aligned on a byte boundary and help the decoder compute the number of skipped blocks with respect to the previous markers and determine where in the image the decoding should resume. Although insertion of restart markers is optional, their presence helps decrease complexity of carving significantly as the data following a marker will always correspond to the start of a new MCU. In fact, use of restart markers have already been

considered in the context of file carving [9], [16], [17]. In our image set, however, we found that only 11.88% of the images ($53,966$ images) used restart markers. (Table II shows a breakdown of the frequency of restart markers for each chroma subsampling scheme.) This finding shows that the carving method cannot rely on the presence of restart markers.

TABLE II
USAGE OF CHROMA SUBSAMPLING SCHEMES AND RESTART MARKERS

Subsampling Scheme	Distribution of Photos	Photos with Restart Markers
Horizontal	84.24% (382,600)	11.92% (45,591)
Vertical	04.13% (18,780)	00.01% (2)
Hor.&Ver.	10.71% (48,645)	16.91% (8,225)
No-Sampling	00.92% (4,191)	03.53% (148)

Image Quality: JPEG achieves most of its compression through quantization of transform coefficients. This is realized by use of a quantization table, an 8×8 matrix of integers that specify the quantization value for each DCT transform coefficient. (Typically, two different quantization tables are used, one for the luminance component and the other for chrominance components.) The strength of compression is usually expressed in terms of quality factor (QF), a number that varies between 1 and 100 and corresponds to use of scaled versions of standard quantization tables depending on the designated QF. It must be noted, however, that manufacturers may also use custom quantization tables that are not based upon the standard. In those cases, we estimate the quality factor of images by deploying the publicly available JPEGsnoop utility [18]. Figure 3 shows the distribution of estimated QFs of all the images. Although 6091 distinct quantization tables were seen to be used, since 96% of all images were found to be above QF 80, carving can be performed using a few default quantization tables without a significant impact on the quality of the recovered file fragment [9].

Overall, encoder setting characteristics obtained from this large set of images most importantly reveal that a very significant portion of images captured by digital cameras perform JPEG compression in the baseline mode and use a particular set of Huffman tables for entropy coding. Next, we describe the details of our technique that utilizes this information for decompressing a partially intact JPEG bitstream and determining all the necessary parameters needed to render

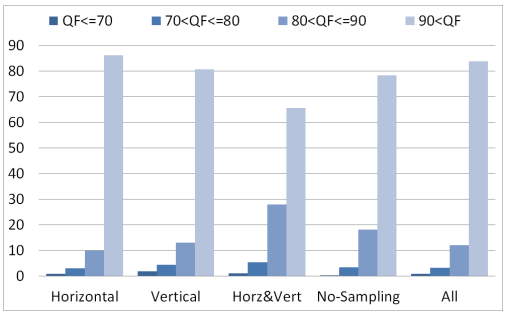


Fig. 3. Estimated compression quality factor (QF) values from the images.

the corresponding segment of an image.

IV. Orphaned JPEG File Fragment Carver

Given a fragment of a JPEG baseline coded image, our carving technique follows the process flow diagram shown in Figs. 4 and 5. For the general case, we assume that the file fragment includes no header data or markers (like start of scan, restart or end of image markers), which when present will only make the carving process easier. The first step of the carving is the detection of the byte boundary in the partial bitstream. Although carving will begin at the start of the bitstream, byte boundary detection is necessary to distinguish stuffed bytes from the coded image data. After all possible byte boundary positions are determined, a set of Huffman tables are selected for decoding. In the current setting, although we only consider the standard set of Huffman tables due to their prevalence, one also needs take into account the fact that there may be other sets of tables specific to certain camera makes/models and editing software.

Using the selected set of Huffman tables, our technique tries to decode the data assuming three possible MCU structures. (Note that horizontal and vertical subsampling schemes only concern the layout of luminance blocks and will both yield to the `YYCbCr` MCU structure.) This may potentially lead to three coefficient arrays, each associated with one of the MCU structures, and the correct chroma subsampling scheme and layout is identified through a variety of statistical measures. If errors are encountered and decoding fails for all three MCU structures, decoding steps are repeated considering either another possible byte position or a new set of Huffman tables, until successful decoding is achieved.

Entropy decoded DCT coefficients are then dequantized using the quantization table that was encountered most frequently (in our Flickr image set) for the identified subsampling scheme

and are translated to spatial domain. This is followed by analysis of the recovered image blocks to determine the image width, to detect offset from the leftmost position on the image, and to adjust the brightness level of the image. The resulting image fragment is finally assessed for whether it exhibits the natural image statistics before it is considered successfully recovered. If not, we assume the incorrect carving might be due to byte boundary detection, identified MCU structure or the choice of Huffman tables, and the algorithm is run again assuming a different decoding setting.

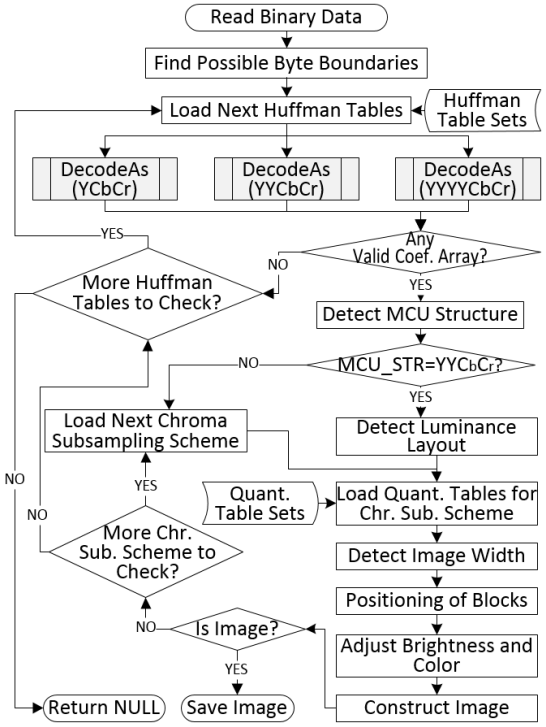


Fig. 4. The process flow diagram of the proposed file carving technique.

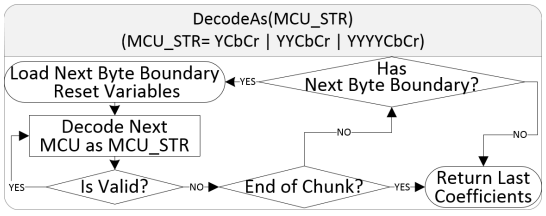


Fig. 5. Detailed view of the 'DecodeAs' sub-process defined in Fig. 4.

A. *Decoding the Bitstream*

Decoding an encoded bitstream at an arbitrary bit location requires determining two things: first what MCU component is being decoded (*i.e.*, which Huffman table to use for decoding), and secondly alignment of the codewords. At the worst, this latter task would require decoding of the data repetitively, each time discarding some bits, until the decoder synchronizes to the bitstream and the data are decoded without any failure. There are two potential causes of a decoding failure.

The first concerns encountering an invalid codeword. Huffman codes are examples of complete prefix codes [19], [20], which means that any binary sequence can be decoded as if it was the encoding of an intermediate code string. Figure 6 provides the Huffman code tree corresponding to luminance `DC` component of the most commonly used Huffman table by digital camera manufactures. This binary tree, however, is not complete as the last leaf node that corresponds to the codeword `111111111` is not assigned to an intermediate code, making it an invalid codeword. (Similarly, for both luminance and chrominance `AC` Huffman code trees, 16-bit all-ones codewords and for the chrominance `DC` tree 11-bit all-ones codeword are invalid.) Therefore, the probability of encountering an invalid codeword during decoding is small, but nonetheless non-zero (*e.g.*, any sequence up to 9-bits will be decoded successfully with probability 0.998 in the above case). The second type of failure that might commonly occur is coefficient overflow, which happens when the number of decoded coefficients in an 8×8 block is found to be more than 64.

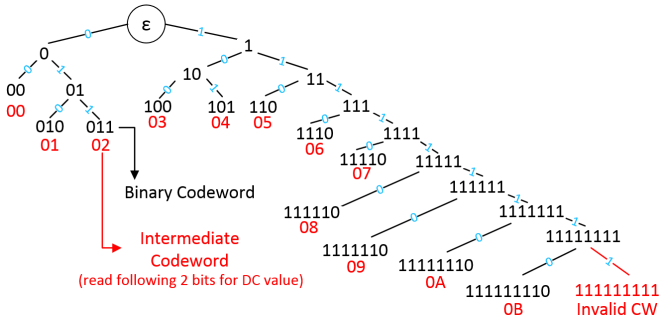


Fig. 6. Huffman code tree used for encoding `Y` – `DC` coefficients from the JPEG standard.

Byte stuffing at the encoder adds a further level of complication to the whole decoding task. When the bitstream is not byte aligned, it won't be possible to discriminate between stuffed

bytes and the encoded data during decoding. Therefore, although decoding will start at the start of the bitstream, byte boundaries need to be detected to discard non-encoded data. (It must be emphasized here that in the coded data segment only restart markers are byte aligned.) The details concerning how we detect byte boundary and establish synchronization with the bitstream are provided below.

1) Byte Boundary Detection: Since `0xFF` byte identifies the first byte of all JPEG markers, when it is encountered in the encoded data, the decoder may interpret it as the beginning of a restart marker (the only marker allowed to appear in the coded data segment that will increment from `0xFFD0` to `0xFFD7`). To avoid this confusion, the standard requires that when the encoded data yield the `0xFF` byte value at a byte boundary, it must be followed by a `0x00` byte. This indicates decoder that the preceding `0xFF` byte is in fact data, and the `0x00` byte is to be disregarded [15].

In identifying the byte boundary, we exploit the fact that markers that take values in the ranges of `0xFF01` to `0xFFCF` and `0xFFD8` to `0xFFFF` cannot appear in the encoded data segment. Hence, any occurrence of these marker patterns in the encoded bit sequence must be due to coded data and at a non-byte boundary location. Hence, we obtain all indices that indicate the location of these marker patterns in the bit sequence and eliminate those position indices as potential byte boundaries. (Note that this is realized by evaluating index values in modulo 8, as there only 8 possibilities for a byte boundary.) Our empirical results show that for data blocks of size 4 KiB or larger, byte boundary can be identified with 100% accuracy.

2) Achieving Synchronization: A decoder that starts at some arbitrary bit in the encoded stream is said to achieve synchronization when it generates the first intermediate code from a codeword correctly. Many variable-length codes, including Huffman codes, possess a self-synchronization property that makes them resilient to errors like insertion, deletion or substitution of bits [21], [22]. That is, given a bitstream composed of only Huffman codewords, errors in the bitstream will cause incorrect parsing of the codewords at the beginning but the decoder will eventually resynchronize. Synchronization behavior of Huffman codes have been extensively studied and various results are obtained on the existence and determination of synchronizing strings (a string that when encountered immediately resynchronizes the decoder) [19], [20], [22] and on the computation of average synchronization delay [23], [24]. The self-synchronization property of Huffman codes have been utilized by [25] for recovering PNG images, where encoded data is

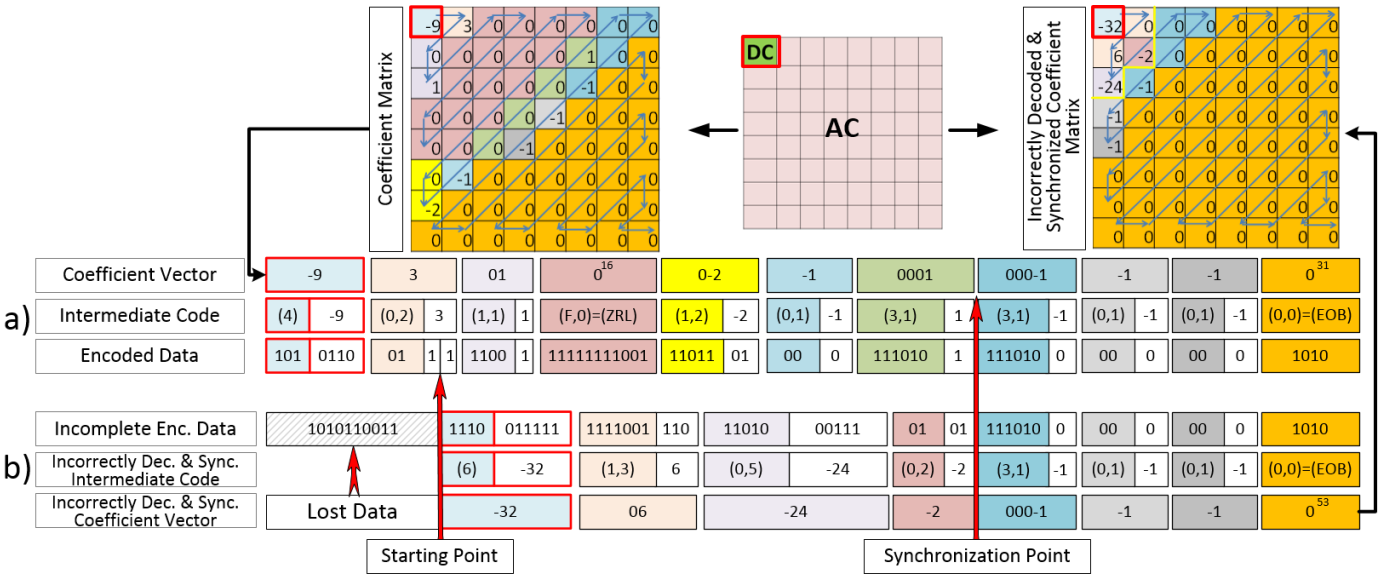


Fig. 7. (a) Huffman encoding of a block of quantized transform coefficients. (b) Synchronization of JPEG decoder to the encoded bitstream obtained when its first 10 bits are missing. It must be noted that both DC and AC coefficients are encoded and decoded using the recommended Huffman tables in the JPEG standard [15]. Huffman code tree for the Y-DC coefficients is given in Fig. 6).

essentially a sequence of Huffman codewords.

The issue of synchronization gets more complicated in the JPEG coded bitstream as codewords from different Huffman tables are mixed together with coefficient values. Therefore, the earlier theoretical results do not directly apply, and their extension to JPEG encoding is not trivial. This problem has so far only been initially explored in the context of parallel decoding of JPEG images [26]. This work essentially observed that the property of Huffman codes to resynchronize seems to hold true even when codewords are mixed with coefficient values. Figure 7 presents an example of the synchronization phenomenon considering JPEG encoding. In the example, first 10 bits of the encoded bitstream corresponding to the 8×8 block of coefficients shown in Figure 7(a) are lost, thereby yielding the incomplete bitstream given in the first row of Figure 7(b). It can be seen that after decoding four codeword and value pairs, which resulted with five coefficients, decoder achieved synchronization with the encoder. Despite synchronization, however, 27 coefficients encoded in the first portion of the original bitstream (prior to synchronization point) cannot be recovered, and, therefore, the resulting block data are incorrect. The `EOB` codeword encountered at the end of the block (which fills the rest of the block with zero valued coefficients) ensures

that correct decoding will start at the next block.

The fact that chroma subsampling scheme is not known when carving an orphaned file fragment adds another level of complication to synchronization. This is because a mismatch between the actual and assumed type of MCU structures will eventually lead to loss of synchronization even though it might have been already achieved. (Assume data are encoded in the `YCbCr` structure and synchronization has been established while decoding the `Y` block, considering any other MCU structure, *e.g.*, `YYCbCr` or `YYYYCbCr`, as the correct one will result with an attempt to decode the following `Cb` block data with an incorrect Huffman code table.)

Under any synchronization setting, however, the ability to correctly decode blocks primarily depend on the presence of `EOB` codewords. Otherwise, block boundaries could not be identified and the decoder will drift in and out of synchronization. Results on the prevalence of `EOB` codewords obtained on $454,216$ JPEG baseline images reveal that 94.11% of all the coded 8×8 blocks in these images terminated with an `EOB` codeword. Table III provides the statistics on the frequency of occurrence of `EOB` codeword for each color component considering different chroma subsampling schemes.

TABLE III
PREVALANCE OF `EOB` CODEWORD IN CODED IMAGE BLOCKS (%)

Subsampling Scheme	Y	Cb	Cr
Horizontal	96.96	99.67	99.70
Vertical	97.00	99.87	99.88
Hor.&Ver.	92.89	97.00	97.18
No-Sampling	72.70	88.14	88.28

Although `EOB` codewords are encountered quite frequently, it is still possible for the decoder to synchronize in a block that ends without the `EOB` codeword (*i.e.*, when the last `AC` coefficient of a block is non-zero valued). In that case, decoder is very likely to continue decoding across consecutive blocks until either the decoded coefficients overflow the 8×8 block or an invalid codeword is encountered, both of which will indicate that synchronization is lost. Due to this possibility, in the occurrence of a coefficient overflow or an invalid codeword, our technique discards all the data decoded up until that point and restarts decoding a new block expecting synchronization will be re-established and an `EOB` codeword will be encountered at the block

boundary.

To determine how fast the decoder is able to achieve synchronization and starts decoding correctly, we conducted tests on 400 randomly selected images obtained from our image set. This set of images comprises four equal-sized subsets of images based on the chroma subsampling scheme used in their coding. A random chunk of data (including JPEG header portion) is deleted from each file to ensure they start at an arbitrary point in the original bitstream. Resulting data are then decoded by asserting the above error conditions. Table IV shows the average delay measured in terms of the number of bits, codewords, coefficients and MCUs for each chroma subsampling scheme until correct decoding starts.

The results show that all of the orphaned JPEG file fragments could be decoded correctly by only discarding up to three MCUs worth of image block data. For horizontal and vertical subsampling schemes, it requires around 50 decoded codewords to achieve this. In the no-sampling case, despite the lower number of components, this number is slightly higher, which can be attributed to less frequent use of `EOB` codewords, *i.e.*, indicating that a low- or mid-level compression is being applied. (See the last row of Table III.) For the case of horizontal and vertical sampling, however, there is a significant increase in decoding delay. This is primarily so because the decoder has to synchronize multiple times with the bitstream before it can align itself with the ordering of components (*i.e.,* when synchronization occurs at a `Y` component, decoder could not identify which of the four `Y` components it is decoding and eventually loses synchronization).

TABLE IV
SYNCHRONIZATION DELAY STATISTICS

Subsampling	Average Number of Discarded Data in			
Scheme	Bits	Codewords	Coeffs.	MCUs
Horizontal	292.18	54.22	386.03	2.18
Vertical	238.13	43.62	371.89	2.11
Hor.&Ver.	999.21	182.18	1199.7	3.69
No-Sampling	367.89	67.44	406.52	2.76

B. Identifying the MCU Structure

To determine the employed subsampling scheme, we decode the orphaned JPEG file fragment assuming three MCU structures. The premise of our identification approach relies on the finding that decoding with the correct MCU structure will yield to fast synchronization and correct recovery of block data. On the contrary, when there is a mismatch in the assumed and actual MCU structures, decoder won't be able to maintain synchronization, and, as a result, it will either encounter an invalid codeword or generate block data that do not exhibit the characteristics of quantized DCT coefficients. In our approach, we utilize these two factors to identify the MCU structure of the JPEG encoded data.

To test its effectiveness, we determine how frequently one encounters an invalid codeword when the assumed MCU structure is incorrect. Experiments are performed on varying sizes of data chunks obtained from the dataset of 400 images described above. Figure 8 shows the corresponding probabilities. These results show that as the size of data chunks gets larger than 16 KiB, the likelihood of all incorrect MCU structures yielding an invalid codeword increases to above 90%. However for data chunks of 8 KiB or lower, which seem to be inadequate to encounter an invalid codeword for most cases, correct identification cannot be made as there is more than one possibility for the correct MCU structure. In those cases, resulting coefficient arrays are examined to determine whether their statistical properties resemble those of block-DCT coefficients.

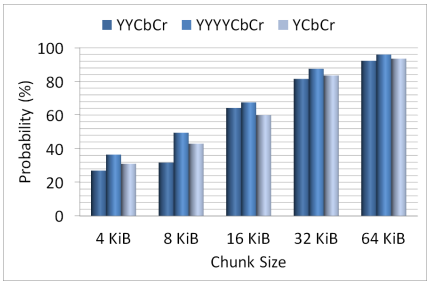


Fig. 8. Probability of encountering an invalid codeword while decoding data with all the MCU structures other than the correct one.

We utilize two defining characteristic of 8×8 block-DCT coefficients for this purpose. First, most DCT coefficients beyond the upper-left corner have typically zero or very small values. Second, since `DC` coefficient values are encoded as a difference from the `DC` value of the

previous block, they won't take arbitrarily high values. We consider five features to capture these characteristics and identify the MCU structure accordingly. These include the means of absolute values of AC coefficients ($f_{\mathbb{E}[|AC|]}$), DC coefficients ($f_{\mathbb{E}[|DC|]}$) and all coefficients combined ($f_{\mathbb{E}[|All|]}$) in addition to the mean ($f_{\mathbb{E}[var(block)]}$) and maximum ($f_{\max[var(block)]}$) of 8×8 block level variances. It is expected that the coefficient array associated with the correct MCU structure will yield the minimum values for all these features. To evaluate the identification performance with these features, tests are performed exclusively in cases where MCU structure could not be identified on the basis of encountering an invalid codeword. As Fig. 9 shows when the decisions of the five features are fused using a majority voting scheme, even for 4 KiB data chunks the identification accuracy reaches 96.75%.

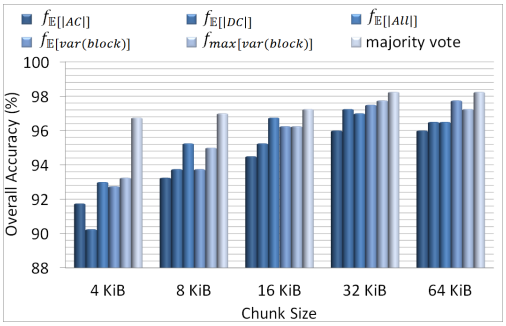


Fig. 9. Accuracies for MCU structure identification based on both encountering an invalid codeword and statistical characterization of 8×8 block-DCT coefficients.

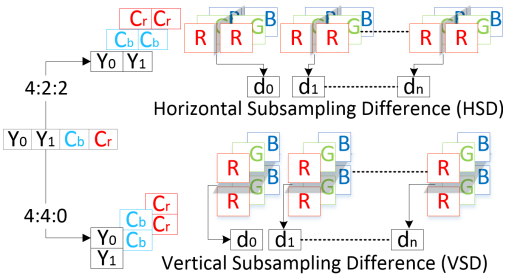


Fig. 10. Luminance layout detection method for YYCbCr MCU structure.

Since the YYCbCr structure may correspond to either horizontal or vertical subsampling schemes (see Fig. 10), the layout of the luminance blocks also needs to be determined. To discriminate between the two layouts, we compute the differences between adjacent block edges in each MCU as illustrated in Fig. 10. Considering both layouts, data are first transformed into RGB color

space and differences between the coefficients across horizontally or vertically adjacent RGB blocks are computed. Average values of each color component are then combined together to produce a single value which is then stored in a vector associated with that layout. For images with sufficient content variation, measured difference should be lesser when blocks are laid out correctly. To verify this, tests are performed on 200 randomly selected images with the `YYCbCr` MCU structure where the layout that yields the lowest average difference is selected as the underlying sampling scheme. Table V shows that an overall accuracy of 94.8% can be achieved in identifying the luminance layout for varying data chunk sizes.

TABLE V
ACCURACY FOR LUMINANCE LAYOUT DETECTION

Fragment size	4 KiB	8 KiB	16 KiB	32 KiB	64 KiB
Hor. Samling (%)	100	100	100	100	100
Ver. Sampling (%)	90	91	91	93	93

C. Detecting Image Width

Once the MCU structure is identified, the next step is to determine the image width. Since JPEG encodes image blocks row by row from left to right, there will be strong correlations between spatially adjacent blocks. However, for blocks that wrap over two rows (such that one block is at the end of a row and the other at the beginning of the next row), this spatial relation will weaken significantly. To exploit this, we arrange all the recovered MCUs in a single row and evaluate the degree of smoothness in the content to identify periodic disruptions that will indicate the start of a new image row.

Our technique utilizes several correlation and clustering based measures to detect image width. These methods essentially measure block-level discontinuities at vertical or horizontal MCU boundaries. As illustrated in Fig. 11, we first compute the average differences between adjacent pixels (in RGB color space) across vertical block boundaries of all adjacently positioned MCUs and store the resulting values in a block difference array (BDA). We then compute the autocorrelation of BDA and calculate the periodicity of peak values in the autocorrelation function (denoted by $w_{R(BDA)}$). The measured period value is the image width in 8×8 blocks. We repeat the same procedure by only considering differences of `DC` coefficients (which is an average of all

transform coefficients in a block) to generate a DC difference array (DCDA) and computing its periodicity ($w_{R(DCDA)}$). Figures 12(a) and 12(b) show examples of the periodicity in BDA and DCDA arrays. We also applied 2-means clustering to values in BDA to distinguish difference values due to neighboring blocks on the same row from the ones due to blocks on subsequent rows. When the difference between the indices of values in the latter cluster are computed, the most frequently encountered index-difference value ($w_{2-Means(BDA)}$) is likely to be the image width in terms of the number 8×8 blocks.

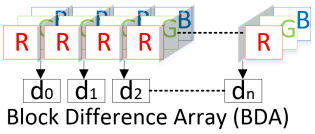


Fig. 11. Computation of block difference array, where each difference value d is computed across vertical block boundaries in adjacent MCUs.

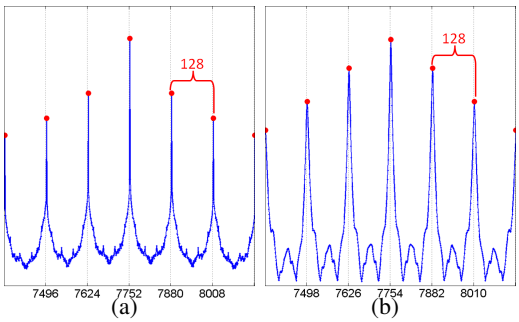


Fig. 12. Peaks in autocorrelation function of (a) BDA and (b) DCDA. A peak periodicity of 128 corresponds to an image width of $128 \times 8 = 1024$ pixels.

To also utilize continuity between spatially adjacent image blocks in the vertical direction, we designate a mask comprising a number of MCUs in RGB color space. This mask is then shifted over the entire row of MCUs by computing the difference between the pixels across the boundary of horizontally adjacent blocks in the mask and the row of MCUs as illustrated in Fig. 13. The mask is shifted block by block and the resulting averaged difference value is stored in a mask difference array (MDA). When correct alignment between the mask and data is attained, the computed difference is expected to be the smallest, and the shift corresponding to the minimum difference in the MDA is deemed to be the width of the image. We used two different masks in detecting the image width. The first mask included the first 20 blocks of the

recovered MCUs and the second mask included the whole MCU array itself. (The two estimated widths using the two masks will be referred to by $w_{MDA-20Bl}$ and $w_{MDA-All}$.)

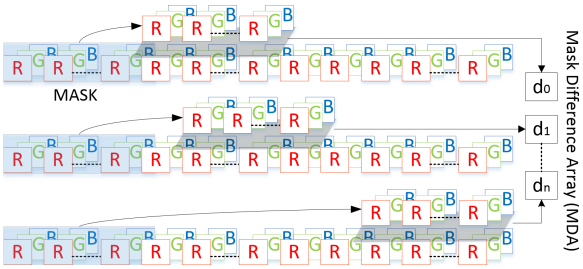


Fig. 13. Computation of mask difference array (MDA). Each difference value d is computed across horizontal block boundaries in adjacent blocks of the mask and the data.

The image width values detected by each measure are then combined together to give a single width value using simple majority voting. If no width value is in majority or all the measures provided different values, then images are constructed at those width values and the one that yields the minimum spatial discontinuity along the boundary of all 8×8 blocks is selected as the final image width.

Tests are performed on 400 images that have 12 different width values that vary between 400 to 1024 pixels and compressed using different chroma subsampling schemes. Figure 14 presents the accuracy for each measure and their fusion in detecting image width, considering different fragment sizes. These results show that for 4 KiB sized fragments, the detection accuracy remained around 70% as the number of rows of blocks that can be recovered from the data is not sufficient to obtain reliable statistics. When the size of encoded data grows from 8 KiB to 64 KiB, however, image width can be detected with an overall accuracy of 95.13%.

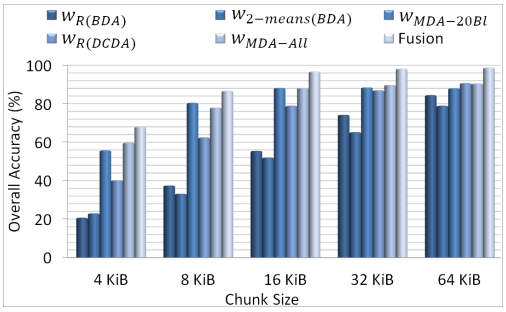


Fig. 14. Accuracy of image width detection for the five measures an their fusion.

D. Positioning of Blocks

Since image data is only partially available, the actual horizontal position of the recovered image blocks in the image cannot be known. When image width is correctly detected, incorrect positioning of the recovered blocks will only cause a circular shift in the image content, where the rightmost edge of the image will appear in some other location in the image, as demonstrated in Fig. 18. In most images, this will introduce a very sharp discontinuity that can be detected by computing the cumulative sum of the differences between boundary pixels of all 8×8 blocks adjacent in the vertical direction. This can be captured by computing the cumulative vertical distance array (CDVA) depicted in Fig. 15 and determining the columns of blocks that yield the highest discontinuity values. However, this measure will fail when the variation in image content is low or the content itself has such strong discontinuities. Experiments conducted on 400 randomly selected images by manually removing an arbitrary number of blocks from each image 100 times show that correct positioning of the blocks can be achieved with 91.79% success rate.

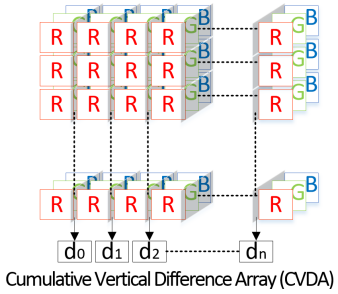


Fig. 15. Computation of cumulative vertical difference array (CVDA) to identify block positions.

E. Adjusting Brightness and Color

JPEG encodes `DC` coefficients differentially with respect to the preceding blocks; therefore, a decoded `DC` coefficient value of an intermediate block will be a difference value rather than its actual value. Since it is not possible to guess the missing base value, we assume it to be zero and decode the incremental `DC` coefficients accordingly. To compensate for the missing base value, we perform histogram stretching on the resulting image and shift the pixel values to fill the entire range of values. Although this procedure may sometimes lead to construction of images with incorrect brightness and color levels, in most cases, it leads to images with acceptable quality.

F. Verifying Natural Image Statistics

As the final step of our algorithm, we verify whether or not the recovered image fragment reflects the statistical properties of natural images. For this, we utilize the regularities in multi-scale representation of images [27]. Essentially, we build a model for natural images using these characteristics and then show how incorrectly carved image fragments deviate from this model. To realize this, quadratic mirror filters (QMF) are first used to decompose the image into three resolution levels in the wavelet transform domain and then first- and higher-order statistics are computed from the coefficients in each sub-band of all orientations, scales and color channels. These statistics include cumulants of the marginal distributions of the sub-band coefficients (like the mean, variance, skewness and kurtosis) as well as those of the distributions associated with linear prediction errors of the coefficient magnitudes. Resulting 72 features are used in conjunction with an SVM classifier to binary classify images as either correctly or incorrectly decoded.

Since the errors concerning image width, block placement, and color and brightness adjustments will in most cases produce perceptually meaningful images (at least, in the context of file carving), tests were performed to determine the technique's ability to discriminate images decoded with incorrect MCU structures. The classifier was trained using 400 photos and 400 incorrectly decoded versions by assuming an incorrect MCU structure (100 images from each subsampling scheme). Testing was performed on a set of 1200 unseen file fragments 400 of which were correctly decoded while the rest included incorrectly decoded ones. The file fragments used for training and testing the classifier were obtained randomly from images captured by 178 different camera models in a non-overlapping manner.

Table VI provides the classification results for different subsampling schemes. In this table, the diagonal elements show the accuracy in identifying a correctly decoded image as a natural image and off-diagonal elements are the accuracies for designating an incorrectly decoded image as a non-natural image. Results show that correctly decoded images are classified as natural images with an overall accuracy of 96.50% while incorrectly decoded ones are classified as non-natural images with an accuracy of 95.69%.

TABLE VI

BINARY CLASSIFICATION OF RECOVERED JPEG FRAGMENTS DECODED IN CORRECT AND INCORRECT CHROMA SUBSAMPLING SCHEMES.

	Decoded			
Actual	Horizontal	Vertical	Hor.&Ver.	No-Sampling
Horizontal	**94.34**	83.33	96.67	100.0
Vertical	70.00	**96.67**	98.33	100.0
Hor.&Ver.	100.0	100.0	**95.00**	100.0
No-Sampling	100.0	100.0	100.0	**100.0**

V. EXPERIMENTAL RESULTS

In this section, we provide results on overall success of our technique in recovering orphaned JPEG file fragments on two different datasets.

A. Custom Dataset

Our first dataset includes 4000 file fragments of four different sizes: 8 KiB, 16 KiB, 32 KiB and 64 KiB. These fragments were sampled in a non-overlapping manner from 1000 images that were encoded with four chroma subsampling schemes at 221 different pixel resolutions. The widths of the images varied from 1026 to 12416 pixels with a total resolution ranging from 0.102 megapixel to 12.13 megapixels. Results showing the success of the individual steps of the recovery algorithm are given in Fig. 16. Each stacked column in this figure presents the accuracy in identifying key encoding parameters and the rate of errors stemming from each step of the algorithm for different fragment sizes and subsampling schemes.

It can be deduced from these results that MCU structure can be very accurately detected. The error in MCU structure detection has been highest for 8 KiB file fragments with vertical sampling, where carving failed on 3 images due to incorrect MCU structure identification and on 9 images due to incorrect layout detection. For 8 KiB sized fragments, image widths were correctly detected only with an average accuracy of 52% across all sampling schemes, because for most fragments recovered data were not enough to fill the whole width of an image. (Table VII provides the average MCU counts in a single row that spans the full width of original images in the test dataset and the average MCU counts recovered from 8 KiB fragments with incorrectly detected widths for each chroma sampling scheme.) Although width detection also failed on

some of the fragments with size above 16 KiB, those partially recovered images still presented visually meaningful information. Figure 17 shows examples of incorrectly decoded images due to a mismatch in the estimated and actual image widths.

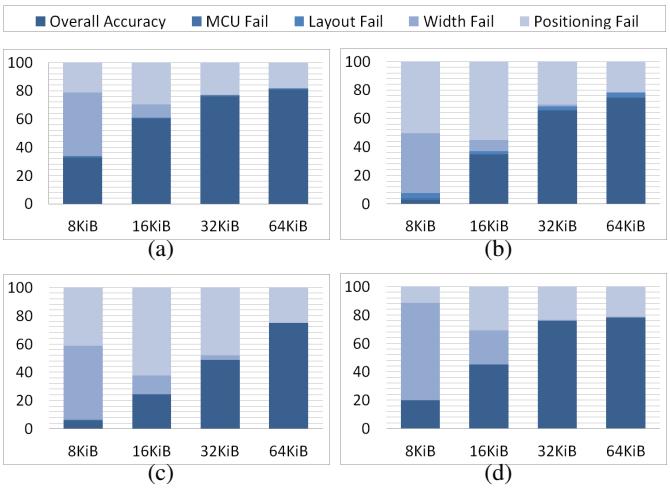


Fig. 16. a) horizontal sampling, b) vertical sampling, c) horizontal and vertical sampling, d) no-sampling

TABLE VII

AVERAGE NUMBER OF MCUS IN A FULL ROW OF ORIGINAL IMAGES AND IN RECOVERED 8 KIB SIZED FRAGMENTS WITH INCORRECT WIDTH

Subsamp. Scheme	Horizontal	Vertical	Hor.&Ver.	No-Sampling
Original Images	401	552	336	993
File Fragments	155	307	170	394

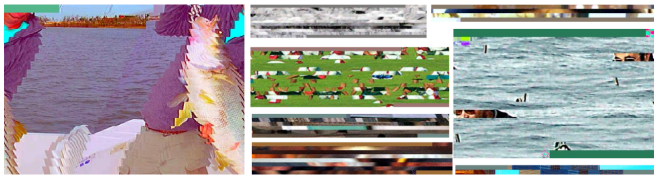


Fig. 17. Recovered images with incorrectly detected widths.

In the above results, the single most common cause of recovery errors is the positioning of the image blocks. For the 8 KiB and 64 KiB file fragments, which on the average have 47 pixels

and 160 pixels height, this alone decreased the overall recovery accuracy by 62% and 11%, respectively. We must note here that a positioning error is declared to have occurred when the actual offset of the first correctly recovered block (from the leftmost side of the image) is not among the top five offsets corresponding to the five highest elements of the CVDA array. Our results show that correct positioning of blocks can be achieved on average in 1.24 trials for all fragment sizes and sampling schemes. That is to say, in majority of the cases, correct offset can be detected either in the first or second trial or it cannot be reliably determined. Our experiments also show that in order to correctly position recovered blocks, on average, recovered fragment data should be more than 160 pixels in height and have more than 6400 MCUs. Table VIII shows the average heights and MCUs of recovered fragments for each chroma subsampling scheme and fragment size. In any case, this type of error is the most innocuous of all as the context of the recovered image would still be apparent to a human analyst. Figure 18 shows some of the incorrectly decoded images due to errors in block positioning.

TABLE VIII

AVERAGE NUMBER OF MCUS IN RECOVERED FRAGMENTS AND THEIR HEIGTHS

Subsamp.	Average MCUs				Average Height (pixels)			
Scheme	$8K$	$16K$	$32K$	$64K$	$8K$	$16K$	$32K$	$64K$
Horizontal	838	1619	2958	5567	37	42	68	127
Vertical	1379	2123	3591	6279	54	68	114	199
Hor.&Ver.	562	1067	1856	3498	50	64	108	205
No-Samp.	1461	2894	5817	10275	47	46	61	107

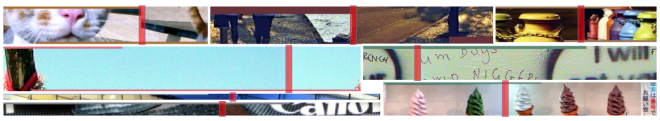


Fig. 18. Recovered images with incorrect positioning of blocks. Red lines indicate the rightmost edge in the original images.

We also measured the resultant peak signal to noise ratio (PSNR) of recovered image fragments following brightness and color adjustment. Table IX shows the average PSNR values and standard deviations for different fragment sizes and subsampling schemes. It can be seen that the PSNR values for all recovered images remained quite uniform across all cases with an overall average

value of 17.22 dB, which can be considered acceptable in a forensic recovery setting. Figure 19 shows an example of a fragment before and after brightness and color adjustment which, respectively, yielded the PSNR values of 12.68 dB and 21.14 dB. Figure 20 shows a collection of correctly recovered images from the test fragments that are encoded with different subsampling schemes at various widths.

TABLE IX

AVERAGE PSNR VALUES AND THEIR STANDARD DEVIATIONS FOR ALL SAMPLING SCHEMES AND FRAGMENT SIZES

	Horizontal	Vertical	Hor.&Ver.	No-Sampling
	avg. (std. dev.)			
8 KiB	17.20(2.03)	15.20(3.20)	16.10(1.55)	17.01(2.58)
16 KiB	17.51(2.18)	16.66(2.19)	16.81(2.75)	17.19(2.66)
32 KiB	18.08(2.48)	16.44(2.29)	17.67(2.79)	18.03(2.89)
64 KiB	18.75(2.50)	16.49(2.40)	17.95(2.80)	18.41(2.80)

(a) (b) (c)

Fig. 19. a) Image before color adjustment, PSNR=12.68 dB. b) Image after color adjustment, PSNR=21.14 dB. c) Corresponding part of the original image.

Fig. 20. Correctly recovered orphaned file fragment samples at various image widths, encoded using different chroma subsampling schemes.

B. *Test Disk Images*

We also evaluated our system on a set of disk images created by Garfinkel *et al.* to test file carving tools [28]. The dataset (named, `nps − 2009 − canon2`) includes six FAT32 formatted

SD card images created using a Canon PowerShot SD800IS digital camera. After taking photos, the card is imaged and then select photos are deleted or overwritten by new data. This process of imaging followed by file deletion is repeated for six times, yielding six different disk images. These disk images essentially store four categories of JPEG image data. The first two categories include images that are resident on the file system and those that are deleted but for which file-system metadata is still available. The third category includes files where the encoded data are intact, but no file system meta-data is available. The last group includes files that are not intact and have no file system metadata. Since recovery of files in the first two category is considered trivial, carving tools are mainly evaluated based on their success on recovering images in the latter two categories. Orphaned JPEG file fragments fall into the fourth category; however, since they have no JPEG headers associated with them, they have so far not been considered for recovery. In fact, challenge solutions showing all the fragments that can be recovered from disk images do not list any orphaned file fragments.

To evaluate our proposed technique, we identified all the clusters that were not listed in the recovery reports and run our algorithm on data extracted from those unallocated clusters. Table X presents a list of all clusters that were carved for orphaned file fragments in each generation of the disk images. This table also lists what file was residing in and around those clusters prior to deletion, which are determined by examining the previous version of the disk image. It can be seen in this table that with each deletion operation, some new orphaned file fragments are created in the disk image. In our tests, we performed recovery on each group of clusters, rather than carving each cluster individually, to obtain the partial images given in Fig. 21.

We must note however that when recovering file fragments that span less than five clusters, we observed that alignment errors have occurred at the cluster boundaries due to missing or erroneously constructed excessive blocks. By close inspection of the data in each cluster, we realized that each cluster starts with a non-random 190 byte pattern that decreases by four bytes in each consecutive cluster. (We speculate that this may be due to an attempt to prevent successful recovery by partially overwriting the data in those clusters.) We performed recovery on these orphaned file fragments on a cluster-by-cluster basis and manually combined the resulting image data as shown in Figs. 21 (a), (b), (c) and (h). The red lines within the recovered image data in those figures mark the joining points for clusters and the length of the line shows the amount of data lost due to overwriting.

TABLE X

RECOVERED ORPHANED FILE FRAGMENTS FROM FORENSIC IMAGES (`nps − 2009 − canon2` IN [28])

Disk Image	File Name	Allocated Clusters Prior to Erasure	Carved Clusters After Erasure	Carved Images Fig. 21
gen2	IMG_30	~~1484-1536~~	1518-1536	(f)
gen3	IMG_37(fr_1)	~~1853-1898~~	1860-1898	(d) fr_1+fr_2
	IMG_37(fr_2)	0216-0252	0216-0252	
	IMG_30	~~1484-1536~~	1518-1521	(c)
gen4	IMG_01	~~0004-0056~~	0004-0007	(a)
	IMG_02	~~0057-0110~~	0091-0110	(e)
	IMG_37(fr_1)	~~1853-1898~~	1860-1898	(d) fr_1+fr_2
	IMG_37(fr_2)	0216-0252	0216-0252	
	IMG_30	~~1484-1536~~	1518-1521	(c)
gen5	IMG_01	~~0004-0056~~	0004-0007	(a)
	IMG_02	~~0057-0110~~	0091-0093	(h)
	IMG_08	~~0369-0414~~	0387-0414	(i)
	IMG_30	~~1484-1536~~	1518-1521	(c)
	IMG_37(fr_1)	~~1853-1898~~	1860-1898	(g)
gen6	IMG_01	~~0004-0056~~	0004-0005	(b)
	IMG_12	~~0568-0617~~	0598-0617	(j)
	IMG_30	~~1484-1536~~	1518-1521	(c)
	IMG_37(fr_1)	~~1853-1898~~	1860-1898	(g)

VI. DISCUSSION AND CONCLUSIONS

In this work, we advance the state-of-the-art in file carving by introducing the ability to recover fragments of a JPEG compressed image file when its header, encompassing all the necessary decoding parameters, is not available. Since the proposed file carving technique uncovers a new class of evidence, it has significant implications on forensic investigations and intelligence gathering operations.

Our approach is data-driven in that it examines encoding settings of a large number of Flickr photos captured by a wide variety of digital camera makes and models. Our carving technique utilizes this information in devising an algorithm that both uncompresses the file fragment data and determines all the parameters needed for rendering the underlying image data without any human intervention. The effectiveness of the proposed technique is demonstrated on both a custom dataset of file fragments and on forensic images of a storage media.

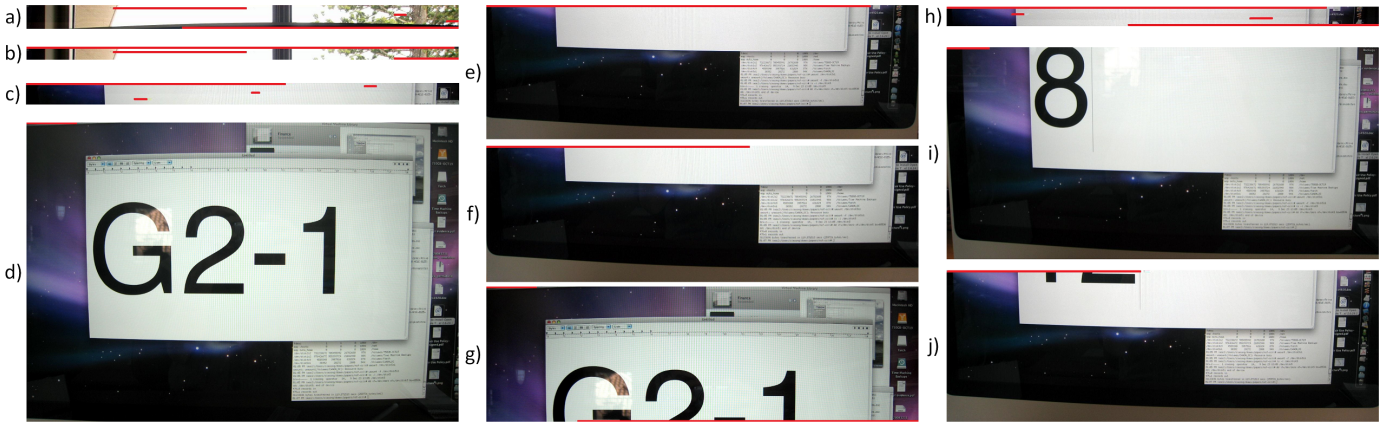


Fig. 21. Recovered file fragments from the test disk images: a) IMG_01 (gen4); b) IMG_01 (gen6); c) IMG_30 (gen3); d) IMG_37 (gen4); e) IMG_02 (gen4); f) IMG_30 (gen2); g) IMG_37 (gen6); h) IMG_02 (gen5); i) IMG_08 (gen5); j) IMG_12 (gen6)

The feasibility of our approach essentially depends on the knowledge of the employed Huffman code tables. Although our findings on Flickr photo set showed that 99.5% of the JPEG coded photos taken by digital cameras are saved in the baseline mode and 98.6% of those use the standard set of tables, the fact that Huffman codes can be tuned for each image imposes a critical limitation on our orphaned file carving technique. In fact, many of the commonly available image editing tools can be used for re-encoding images with different Huffman tables. To determine how common a phenomenon this is, we examined images that have the Adobe Photoshop application marker tag in their ExIF headers, as it is the most well known and widely used tool for editing images. This revealed that out of the $139,637$ images, $117,863$ (90.01%) of them still used the standard tables, while the rest of the images used 9257 different sets of tables. Hence, it can be inferred that for the general case, having been processed by an editing tool does not necessarily curtail the use of our carving technique.

From a practical standpoint an important issue concerns the question of how to determine whether a given block of data in a storage volume contains JPEG data. (Otherwise, the carving technique has to be run on all unidentified data blocks.) File and data type classification has received considerable research attention and various techniques have been proposed towards this goal. These techniques, in common, try to achieve this task by extracting a number of statistical features from blocks of data which can then be used to build machine learning based models. Most notably, [29], [30] utilized the fact that within the encoded segment of JPEG files `0xFF`

byte is only allowed to be followed by a particular group of bytes (*i.e.*, `0x00`, `0xD0` to `0xD7`) and reported an identification accuracy of 99% with no false positives. Considering a multi-class identification setting, [31]–[34] considered a variety of features sets and different machine learning methods and reported accuracies ranging from 87% to 98% in distinguishing JPEG file data from other types of data. In the context of our application setting, the distinguishing features proposed by all these work can be utilized to construct a binary classifier to distinguish between JPEG and non-JPEG data containing blocks.

Another issue concerning the practical deployment of the technique is when to stop carving. Since, in most cases, the end of a file fragment could not be known in advance, non-fragment data will also continued to be carved during recovery. This will eventually cause decoding errors due to either encountering invalid codewords or coefficient overflows. Therefore, the fact that a long duration of successful entropy decoding followed by a sequence of failures during decoding might be interpreted as an indicator that the end of the file fragment has been reached.

The other important aspect is the computational efficiency of the proposed technique. To determine this, we measured the average computation times for carving image data from different sized fragments and determined that for 8 KiB, 16 KiB, 32 KiB and 64 KiB file fragments, average recovery time per fragment is, respectively, 4.93, 7.08, 11.60, and 11.8 seconds. (These tests were performed on a computer with Intel 2.4 GHz I7-3630 processor and 8 GB of RAM and using MATLAB as the implementation platform.) We must note that performance of our carving technique can be significantly improved by parallelizing some of the computation steps (like MCU structure and layout identification, width detection and block positioning) and by further optimizing the entropy decoding operations, similar to the way it is performed in the standard JPEG library.

APPENDIX

A. *Banned Software List*

acdsee | adobe | apple imageio | digital photo professional | fixfoto | ijg library | imatch | lead | matlab | office | paint | visio | photo gallery | nikon scan | ashampoo | jpeg wizard2 | zoombrowser | gimp | photoscape | irfanview | idimager | faststone | neatimage | xnview | stoik | photomatix | pixia | visuallightbox | 5dfly | acorn | seashore | pinta | kolourpaint | zoner | hornil | stylepix | photoperfect | xtreme | virtual | photoplus | artweaver | fotomix | pixlr | flauntr | fotoflexer | picture2life | lunapic | phoenix | pixenate | splashup | phixr | sumo | aviary | bubblesnaps | cacoo | chartgizmo | charttool | cosketch | creately | diagram.ly | doppelme | sketchup | pencil | picmarkr | photopeach | reshade | roxio | slidestory | swiffy | toondoo | voki | snipshot | pixer | picmagick | pikipimp | mypictr | 72photos | drpic | alilg.com | picnik | sprout | mixbook | sliderocket | pizap | cellsea | befunky | poladroid | pixelmator | preloadr | photoshop | tiffen | qtpfsgui | arcsoft | mediaone | picasa | vignette | scarab | f-spot | pictureproject | retro camera | camera+ | plasq | mobile fotos | picsay | filterstorm | camera genius | watermark

factory | rawtherapee | microsoft windows photo | imagenomic | autopano | elements organizer | photosynth | pudding camera | corel photo | lightzone | microsoft ice | silkypix | canoscan | ptgui | mobile panorama | kashmir3d | procamera | photo ditto | sigma photo | acd systems | viewnx | mp navigator | ofotonow | ez controller | camedia master | k-x ver | nikon editor | lucidiom apm | capture nx | hdrtist | dxo optics | dcraw | studioline | nikon transfer | darktable | hp scanjet | scan dual | coachware | quicktime | photowatermark | imageready | bibble | capture one | copiks | digikam | snapseed | instagram | picmonkey | hdr photo | imagemagick | photogene | iphoto | phototoaster | borderfx | flickstackr | mediatek | kitcam | iwatermark | tsr watermark image | aftershot | marksta | vscocam | shotwell | slow shutter cam | picajet | torturedmind | playmemories | illuminator | easyhdr | photowizard | picture window | fx photo studio | meitu | photo pos | photogenie | photivo | photophilia | turbo photo | time machine | robogeo | anabuilder | xara | picture maker | photocleaner | picturegear | photoramp | photomizer | spoolrimage | pakonima | nimagefilejpg | photosnap | iconfactory flare | magix foto | color efex | hdr efex | ralpha | alien skin exposure | portrait professional | a better camera | onone perfect photo | jasc | squaready | xara | photo ninja | dxo filmpack | aperture | leonardo | silver efex | doubletake | photo album | artizen | fotoware | intensify | picturepackage | irodio | graphicsmagick | kipi-plugins | inkscape | flickr | nokia panorama | oloneo | image cropper | sagelight | autostitch | panoramastudio | fhotoroom | topaz | flamoz fixer | photofxlab | jerrys developer | silverfast | swankolab | percolator | altphoto | fisheye | imgsource | picsart | handy photo | thumba | panstoria | mpro app | memories | waterlogue | snapheal | picture kiosk | resco | epson scan | camera360 | epson photolier | gnome-screenshot | flickd | greenshot | top camera | colorstrokes | photo fx | mobilemonet | easy-photoprint | mattebox | colorblast | camera noir | bracket mode | brushstroke | instaflash | liquidpixels | luminance | macromedia | masterflexd | autocollage | cogitap | nightcap | phoebe | slowshutter | superimpose | thumbsplus | facebook camera | eztouch | wingnut lo-fi | aoao watermark | by.blooddy.crypto | fotoschau | retrica | camera boost | click | crop and straighten | enfusegui | etchings | exif viewer | gcam | huaqin | cameramatic | instant110 | jpeg gpx merger | photofiltre | photoimpression | photostudio | rawstudio | rookie | stepoks | tomo

REFERENCES

[1] G. G. Richard III and V. Roussev, "Scalpel: A frugal, high performance file carver." *in Proc. of Digital Forensic Research Workshop*, August 2005.

[2] Access Data. (2015, March 22) Forensic toolkit. [Online]. Available: www.accessdata.com/solutions/digital-forensics/ftk

[3] Guidance Software. (2015, March 22) Encase. [Online]. Available: www.guidancesoftware.com

[4] B. Carrier. (2015, March 10) The sleuth kit and autopsy: forensics tools for linux and other unixes. [Online]. Available: www.sleuthkit.org

[5] A. Pal, K. Shanmugasundaram, and N. Memon, "Automated reassembly of fragmented images," *in Proc. of IEEE International Conference on Multimedia and Expo*, vol. 1, pp. 625–628, July 2003.

[6] S. L. Garfinkel, "Carving contiguous and fragmented files with fast object validation," *in Proc. of Digital Investigation*, vol. 4, pp. 2–12, 2007.

[7] N. Memon and A. Pal, "Automated reassembly of file fragmented images using greedy algorithms," *IEEE Transactions on Image Processing*, vol. 15, no. 2, pp. 385–393, 2006.

[8] A. Pal, H. T. Sencar, and N. Memon, "Detecting file fragmentation point using sequential hypothesis testing," *in Proc. of Digital Investigation*, vol. 5, pp. 2–13, 2008.

[9] H. T. Sencar and N. Memon, "Identification and recovery of JPEG files with missing fragments," *in Proc. of Digital Investigation*, vol. 6, pp. 88–98, 2009.

[10] E. Kee, M. K. Johnson, and H. Farid, "Digital image authentication from JPEG headers," *IEEE Transactions on Information Forensics and Security*, vol. 6, no. 3, pp. 1066–1075, 2011.

[11] B. Mahdian, S. Saic, and R. Nedbal, "Blind verification of digital image originality: A statistical approach," *IEEE Transactions on Information Forensics and Security*, vol. 8, no. 9, pp. 1531–1540, 2013.

[12] E. Hamilton, "JPEG file interchange format. Version 1.02," *C-Cube Microsystems*, 1992.

[13] Camera & Imaging Products Association, "Exchangeable image file format for digital still cameras: Exif version 2.3," , Tech. Rep. CIPA DC-008-2012 & JEITA CP-3451C Standard, 2010.

[14] D. Salomon, *Data compression: the complete reference*. Springer, 2004.

[15] International Telecommunication Union, "Information technology-digital compression and coding of continuous-tone still images: Requirements and guidelines," Tech. Rep. CCITT Rec. T.81, 1992.

[16] M. Karresand and N. Shahmehri, "Reassembly of fragmented JPEG images containing restart markers," *in Proc. of European conference on computer network defense*, 2008.

[17] T. Gloe, "Forensic analysis of ordered data structures on the example of JPEG files." *in Proc. of IEEE International Workshop on Information Forensics and Security*, pp. 139–144, 2012.

[18] C. Haas. (2015, April 1) JPEGsnoop-JPEG file decoding utility. [Online]. Available: www.impulseadventure.com/photo/jpeg-snoop-source.html

[19] T. Ferguson and J. Rabinowitz, "Self-synchronizing huffman codes," *IEEE Transactions on Information Theory*, vol. 30, no. 4, pp. 687–693, 1984.

[20] C. F. Freiling, D. S. Jungreis, F. Théberge, and K. Zeger, "Almost all complete binary prefix codes have a self-synchronizing string," *IEEE Transactions on Information Theory*, vol. 49, no. 9, pp. 2219–2225, 2003.

[21] V. K. W. Wei and R. A. Scholtz, "On the characterization of statistically synchronizable codes," *IEEE Transactions on Information Theory*, vol. 26, pp. 733–735, 1980.

[22] R. M. Capocelli, L. Gargano, and U. Vaccaro, "On the characterization of statistically synchronizable variable-length codes," *IEEE Transactions on Information Theory*, vol. 34, no. 4, pp. 817–825, 1988.

[23] M. T. Biskup and W. Plandowski, "Shortest synchronizing strings for huffman codes," *Theoretical Computer Science*, vol. 410, no. 38, pp. 3925–3941, 2009.

[24] J.-E. Pin, "On two combinatorial problems arising from automata theory," *North-Holland Mathematics Studies*, vol. 75, pp. 535–548, January 1983.

[25] V. Roussev and C. Quates, "File fragment encoding classificationan empirical approach," *Digital Investigation*, vol. 10, pp. S69–S77, 2013.

[26] S. T. Klein and Y. Wiseman, "Parallel huffman decoding with applications to JPEG files," *The Computer Journal*, vol. 46, no. 5, pp. 487–497, 2003.

[27] S. Lyu, *Natural Image Statistics in Digital Image Forensics*. Springer, 2013.

[28] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *in Proc. of Digital Investigation*, vol. 6, pp. 2–11, 2009.

[29] M. Karresand and N. Shahmehri, "Oscarfile type identification of binary data in disk clusters and ram pages," *in Proc. of Security and Privacy in Dynamic Environments*, pp. 413–424, 2006.

[30] V. Roussev and S. Garfinkel, "File fragment classification-the case for specialized approaches," *in Proc. of Fourth International IEEE Workshop on Systematic Approaches to Digital Forensic Engineering.*, pp. 3–14, 2009.

[31] W. C. Calhoun and D. Coles, "Predicting the types of file fragments," *in Proc. of Digital Investigation*, vol. 5, pp. 14–20, 2008.

[32] C. J. Veenman, "Statistical disk cluster classification for file carving," *in Proc. of IEEE International Symposium on Information Assurance and Security*, pp. 393–398, 2007.

[33] L. Sportiello and S. Zanero, "File block classification by support vector machine," *in Proc. of IEEE International Conference on Availability, Reliability and Security*, pp. 307–312, 2011.

[34] Q. Li, A. Y. Ong, P. N. Suganthan, and V. L. Thing, "A novel support vector machine approach to high entropy data fragment classification." *in Proc. of South African Information Security Multi-Conference*, pp. 236–247, 2010.