Problem solved: a reliable, deterministic method for JPEG fragmentation point detection

Vincent van der Meer[a], Jeroen van den Bos[b], Hugo Jonker[c], Laurent Dassen[a]

[a] *Zuyd University of Applied Sciences, Heerlen, The Netherlands*
[b] *Infix Technologies, The Hague, The Netherlands*
[c] *Open University of the Netherlands, Heerlen, The Netherlands*

Abstract

Recovery of deleted JPEG files is severely hindered by fragmentation. Current state-of-the-art JPEG file recovery methods rely on content-based approaches. That is, they consider whether a sequence of bytes translates into a consistent picture based on its visual representation, treating fragmentation indirectly, with varying results. In contrast, in this paper, we focus on identifying fragmentation points on bit-level, that is, identifying whether a candidate next block of bytes is a valid extension of the current JPEG. Concretely, we extend, implement and exhaustively test a novel deterministic algorithm for finding fragmentation points in JPEGs. Even in the worst case scenario, our implementation finds over 99.4% of fragmentation points within 4 kB – i.e., within the standard block size on NTFS and exFAT file systems. As such, we consider the problem of detecting JPEG fragmentation points solved.

Keywords: JPEG validation, file recovery, digital forensics

1. Introduction

Photos can contain crucial evidence in forensics. File recovery (i.e., the reconstruction of deleted digital files) is an essential forensic capability. By a significant margin, the JPEG file format is the most widely used digital storage format for photos (Hudson et al., 2018). Therefore, JPEG file recovery is an important and ongoing field of research. Recovery of JPEG files is severely hindered by file fragmentation. While not all JPEG files represent photos, those that do are sufficiently large (a few megabytes) that roughly ~8% are likely to be fragmented (van der Meer et al., 2021, Table 4) on NTFS file systems. Exacerbating this problem is the fact that JPEG files consist mostly of high-entropy data, without markers or checksums that can validate partial (nor full) integrity. This seems to imply that there is hardly any relation between the bytes of a JPEG file. Consequently, most existing literature on JPEG file recovery has thus focused on one of two approaches. The first focuses on recovering non-fragmented JPEG files based on header and footer matching, for example, the works by Karresand and Shahmehri (2008) or Fei and Abdullah (2020). The second approach focuses on recovering any JPEG file using visual compatibility, such as the works by Li et al. (2011) or Tang et al. (2016).

However, in our previous publication (van der Meer and van den Bos, 2021) we showed that there are internal consistency requirements for the high-entropy portion of a JPEG file. We proposed a theoretical algorithm that would leverage these consistency requirements to determine whether a block of bytes, as stored on-disk, is a valid continuation of an initial part of a JPEG file. While we showed feasibility, we not implement our approach and therefore did not test efficacy.

Contributions. In this work, we expand upon, implement, and measure efficacy of the algorithm introduced in our previous publication (van der Meer and van den Bos, 2021). We implemented a JPEG validator that reports the exact location where the continuation of a JPEG bitstream (specifically within the high-entropy coded data sections) becomes invalid by (1) Huffman code lookup errors and (2) quantization array overflow. For the purpose of testing these validation methods, we gathered a sizeable, relevant dataset of JPEG files, covering all variants of baseline and progressive JPEGs occurring in the real world. The performance of our validator in detecting fragmentation points is tested against these JPEGs. The results show that our validator achieves phenomenal performance: for the predominant encoding (baseline JPEGs), it achieves a success rate of 99.997% in identifying a fragmentation point.

Availability. An open-source implementation of the algorithm proposed in this paper is available for download from our GitHub repository[1]. The JPEG datasets, as described in Section 5.4, are also included in this repository. We have compiled a comprehensive list of 230,157 JPEG image filenames used in our validation process (see Section 5.1). Available as a text file in the repository, this list aids in enhancing transparency and facilitates the reproduction and validation of our research.

2. JPEG file format

The JPEG image file format is the predominant file format for photos by a significant margin (Hudson et al., 2018). Since

*Corresponding author
Email address: vincent.vandermeer@zuyd.nl (Vincent van der Meer)

[1] https://github.com/parsingdata/jpeg-fragments

Symbol	Hex value	Meaning
SOI	FFD8	Start of Image
APP0	FFE0	JFIF metadata
APP1	FFE1	Exif metadata
DQT	FFDB	Define quantization table(s)
SOF0	FFC0	Start of frame, baseline DCT
SOF2	FFC1	Start of frame, progressive DCT
DHT	FFC4	Define Huffman table(s)
DRI	FFDD	Define restart interval
SOS	FFDA	Start of scan
RSTn	FFDn	Restart ($n \in \{0, 1, ..., 7\}$)
EOI	FFD9	End of image

Table 1: Most important JPEG Markers

its introduction in 1992, it quickly became the de facto standard in many domains. The JPEG specification allows for 16 types of encoding of image data (SOF0–SOF15). In practice, two types of encoding are used: baseline JPEGs (SOF0, Fig. 1) and progressive JPEGs (SOF2, Fig. 2). Baseline JPEGs must be encodable/decodable in a single scan of the image data. Progressive JPEGs require more than one scan.

A JPEG file consists of two types of data: metadata needed for decoding, and image data (also called entropy coded data) that is actually decoded when viewing an image. The metadata specifies how the image should be decoded.

Markers. The JPEG format uses markers for structure. Markers are reserved two-byte values that may not be used for other purposes. For example, there are markers for start/end of image, type of encoding, metadata, etc. (see Table 1). By definition, each JPEG file starts with a START OF IMAGE (SOI) marker and ends with an END OF IMAGE (EOI) marker. The START OF SCAN (SOS) marker indicates the start of an entropy coded data segment, and the first such marker thus is the separator between header and content of a file. The header contains all information needed for decoding such as information on how many color channels an image has (part of START OF FRAME (SOF) marker), whether, and what type of chromatic subsampling is used (also part of the SOF marker), what values should be used for quantization (DQT marker), and compressed representations of the Huffman tables (DHT marker) that should be used.

Decoding compressed image data. At the lowest level, the JPEG format stores the values of each 8×8 block of pixels in a Minimal Coded Unit (MCU). An MCU contains either one (Y) or three (YCbCr) color channels. Each color channel is described by a two-dimensional 8×8 data structure called the quantization array (QA). The QA is stored as follows: the lengths of the quantization values are Huffman-encoded. In addition, the decoded Huffman values also encode how many (if any) zeroes must be placed in the quantization array before the actual value. The actual values are stored directly in the bitstream.

Progressive encoding. Progressive JPEGs, as previously mentioned, mandate sequential scans since the image is constructed

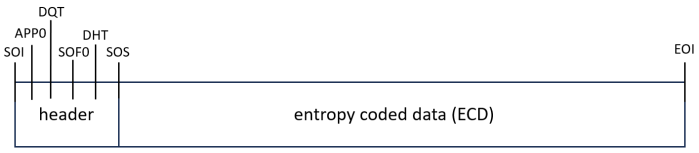

Figure 1: Example JPEG baseline file structure

of multiple layers of image data, with each subsequent layer adding additional image detail. The occurrence of markers within the high-entropy coded data section of a progressive JPEG differs compared to a baseline coded JPEG. Not only does each layer start with a SOS marker, each layer can be accompanied by new Huffman tables, indicated by the presence of a DHT marker.

Support for file recovery. In terms of file recovery, JPEG has some major shortcomings. While it does have identifiable markers, it lacks integrity checks (like CRC) or length fields that encompass the image data. Additionally, its design prioritizes data storage optimization, meaning that image data has high-entropy. By prioritizing compression over robustness, the JPEG file format's design severely complicates file recovery.

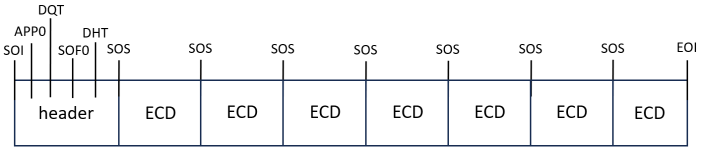

Figure 2: Example JPEG progressive file structure

3. Fragmentation point detection for JPEG

3.1. File fragmentation

When files are stored in a file system, they are stored on zero[2] or more blocks. Ideally, a file's data is stored in-order in consecutive blocks. When the file system is unable to do so, the file becomes fragmented. In that case, the file's data is stored in multiple ranges of blocks. For example, a 2-block file whose blocks are not stored in-order consecutively is already fragmented.

Fragmentation point detection. In file recovery, sometimes a file needs to be reconstructed from fragments that consists of one or more consecutive blocks of data, found on a storage device. This is generally a difficult task, since the file system meta data with the allocated blocks for that file is missing. Therefore, a file carver uses a format-specific validator to parse candidate blocks. If validation concludes successfully, the file is reconstructed from the used blocks. If the validator returns an error, the reported location helps the carver determine in which block the fragmentation occurred. The validator reports the last known good location.

[2]Resident files are stored entirely within the Master File Table.

3.2. Validation using Huffman table lookup errors

The JPEG format employs Huffman encoding to store quantization values of the MCU. The Huffman table needed for decoding is present in compressed form in the header of each JPEG file. The compressed Huffman table does not contain the Huffman codes, but instead contains the symbols and frequency of each code-length. Each Huffman table must thus be reconstructed during decoding (e.g., Table 2). Note that an optimal Huffman table would allow any bit sequence to be decoded. However, the last entry of each Huffman table in a JPEG file always ends in a superfluous '0'. As a consequence, not all possible bit sequences are valid Huffman codes. For example, in the example of Table 2, the bit sequence `11111` would result in a Huffman table lookup error. This should not occur in a valid JPEG file. Therefore, any Huffman table lookup error constitutes a validation error.

3.3. Validation using quantization array overflows

As discussed above, each quantization array describes one color channel for 8×8 pixels. Therefore, each quantization array must contain precisely 64 values.

When decoding JPEGs, Huffman table decoding and the filling of quantization arrays are interwoven. JPEG uses a mechanism to add up to 15 zeroes to a given quantization array. Decoding invalid data may therefore cause a quantization array overflow. For example, if the first 60 values of a quantization array are set, and the next decoded Huffman symbol implies 10 zeroes need to be added, this causes a quantization array overflow, which constitutes a JPEG validation error.

[For each color channel] (once for each matrix in the MCU), perform the following steps:

1. set QAcounter to 1, b to empty bitstring.
2. while (b is not a valid Huffman code)
   add next bit of bitstream to b
   If no valid Huffman code found after maximum Huffman code length: validation error, report last known good location.
3. Convert b to symbol via DC Huffman table.
4. Interpret the symbol as integer and skip this amount of bits; (re)set QAcounter to 1.
5. Until QAcounter is 64, perform the following steps:
   - 5.1. Read bits from bitstream until the shortest valid Huffman code is encountered
     If not found: validation error, report last known good location.
   - 5.2. Use AC Huffman table to convert found code to symbol.
   - 5.3. *# Validate quantization array*
     Interpret the symbol:
     - 5.3.1. If the symbol is `0x00`: the quantization array is complete, set QAcounter to 64, skip to next iteration.
     - 5.3.2. If the symbol is `0xF0`: add 16 to QAcounter.
     - 5.3.3. For all other values, split symbol into upper and lower nibble.
       - Add int(upper nibble) + 1 to QAcounter
       - skip int(lower nibble) number of bits in bitstream
   - 5.4. **If counter > 64:** validation error, report last known good location.
6. *# Validation succeeded*
   return true

Code	Symbol
`0`	`00000101`
`10`	`00000100`
`110`	`00000110`
`1110`	`00000010`
`11110`	`00000011`

Table 2: Example: reconstructed Huffman table

3.4. Algorithm for baseline JPEGs

The algorithm presented in the preceding column illustrates how to leverage the two bit-level validation techniques to detect fragmentation points in baseline JPEGs.

3.5. Validation of progressive JPEGs

Progressive JPEGs contain multiple Start-of-scan (SOS) markers. Moreover, all progressive JPEGs use spectral selection encoding. This encoding method necessitates multiple scans, each refining the image's detail level. Spectral selection affects how QA-overflow validation works. Lastly, progressive JPEGs may also make use of successive approximation encoding. Both encodings enable additional validation opportunities.

Spectral selection. Spectral selection only fills the quantization array for a specific range of values, starting from the first value. Each scan (indicated by a SOS-marker) adds a specific number of values to the quantization array. The number of values to be added is part of the SOS-marker. This alters the QA-overflow validation method, since each SOS-marker determines the upper bound for QA-indices to be filled for that scan.

JPEG structure validation using SOS markers. Each SOS marker must be present from the first bit following completion of the previous scan. In addition, each next SOS marker must cover QA-indices adjacent to the previously filled range. Violating either of these requirements constitutes a validation error.

Successive approximation. Successive approximation is a refinement of spectral selection. In successive approximation, quantization values are stored per 8 bits. In the first scan (denoted as 'DC-first' or 'AC-first' in the validation algorithm), a bit sequence of at least one bit (commonly 6 or 7 bits) are decoded. In the refinement scans, a single bit per quantization value is added. This means that the decoded Huffman value for a refinement scan must have a lower nibble with value '1', since the lower nibble of the decoded Huffman value determines the number of bits to be read. A lower nibble with a different value than '1' in a refinement scan thus constitutes a (coefficient length) validation error.

3.6. Algorithm for progressive JPEGs

The description here only shows the validation mechanisms related to Huffman table lookup errors, quantization array size overflow, and violations of the single-bit refinement used in successive approximation. Interestingly, JPEGs need not be optimally encoded. The specification itself leaves room for suboptimal (or inefficient) use of Huffman encoding as well as suboptimal run-length encoding. Such inefficiencies are still valid within the JPEG standard, and thus do not cause a validation error.

The validation algorithm has four separate decoding phases. Initial quantization values are set in the 'DC-first' and AC-first' phase, respectively. Progressive JPEGs that only use spectral selection only use these two phases, the other two phases are only used for JPEGs containing successive approximation encoded values. More specifically, the 'DC-refine' and 'AC-refine' phases add bits to an existing QA-value.

The resulting validation algorithm and its subalgorithms are shown below and on the right.

Main algorithm

1. For each SOS-marker:
   (a) Determine SOS-type & validate spectral selection range: DC-first, DC-refine, AC-first, or AC-refine
   (b) For all channels in this scan, for each MCU: perform relevant validation case
2. *# validation succeeded*
   return true

Case: DC-first

1. Set b to empty bitstring.
2. while (b is not a valid Huffman code)
   add next bit of bitstream to b
   If no valid Huffman code found after maximum Huffman code length: validation error, report last known good location.
3. Convert b to symbol via DC Huffman table.
4. Interpret the symbol as integer and skip this amount of bits;

Case: AC-first

1. Set QAcounter to start-of-spectral-selection, b to empty bitstring
2. while (b is not a valid Huffman code)
   add next bit of bitstream to b
   If no valid Huffman code found after maximum Huffman code length: validation error, report last known good location.
3. *# Validate quantization array*
   Interpret the symbol:
   3.1. If the symbol is `0x00`: the quantization array is complete, skip to next iteration.
   3.2. If the symbol is `0xF0`: add 16 to QAcounter.
   3.3. For all other values, split symbol into upper and lower nibble.
   - Add int(upper nibble) + 1 to QAcounter
   - skip int(lower nibble) number of bits in bitstream
4. **If QAcounter > end-of-spectral-selection:** validation error, report last known good location.

Case: DC-refine

1. skip one bit for each MCU.

Case: AC-refine

1. for QAcntr = SpectralSelection-start to SpectralSelection-end:
   (a) while (b is not a valid Huffman code)
   add next bit of bitstream to b
   If no valid Huffman code found after maximum Huffman code length: validation error, report last known good location.
   (b) Split the symbol into a symbol into upper and lower nibble.
   (c) *# Check for coefficient length error*
   If the lower nibble is not 1:
   validation error, report last known good location.
   (d) Read one bit, add its value to the least significant position at QAcntr.
   (e) Set ZSKIP = upper nibble (i.e., #zeroes to be skipped)
   If ZSKIP > zeroes available in spectral selection range:
   validation error, report last known good location.
   (f) Skip ZSKIP zeroes in the bitstream

3.7. Runtime-performance

Our algorithm, designed for integration with a file carver, is not I/O-intensive. Compared to the tasks performed by a standard JPEG decoder, our implementation involves a subset of these operations, primarily focusing on image validation rather than display. Based on this, we anticipate that the algorithm will not significantly impact the overall runtime performance of a file carver.

4. Construction of a wide-coverage evaluation test set

To determine efficacy of these algorithms, we need a real world, diverse set of JPEGs. There are various possible sources for such a set, but not all are equally suitable. For instance, platforms like Instagram or Imgur, which are image-centric social media sites, could serve as potential sources for JPEGs However, these platforms tend to recompress and/or re-encode images in order to optimize storage and bandwidth. This leads to little divergence in relation to used encoders and encoder-settings. An alternative would be photo-sharing sites, such as Flickr. Such sites tend to preserve the original input to avoid altering the intended vision of the image. As such, the original input encoder settings are more likely to be preserved. However, such sites are typically used for high-resolution photos taken by high-end equipment. This also leads to a bias in relation to encoders and encoder-settings. The ideal source should offer a wide range of encoders and/or encoder settings for JPEGs.

One suitable source for this is Wikipedia. Wikipedia requires a very diverse set of images, from photos to maps to diagrams of electrical wiring, the solar system, to geometric proofs, to newspaper scans, etc. Succinctly put, any graphically representable concept can be found on Wikipedia in order to transfer knowledge about the concept. Moreover, Wikipedia is open to contributions from anywhere in the world. This leads to the

expectation that a large variety of image creators for a large variety of content can be found there, and thus that a large variety of encoders would be used.

4.1. Collection & sanitisation

In order to collect JPEG files from Wikipedia, a crawler[3] was used with both `wikipedia.org` and `wikimedia.org` as starting points, with a crawl depth of two. Only files with extension `.jpg` or `.jpeg` (case insensitive) were collected, with a minimal file size of 4097 bytes. [4] We collected 230,157 JPEG files in January 2022. It is important to note that the media files on Wikipedia are subject to various licenses, many of which require attribution. Therefore, while our method of data collection can be replicated, sharing the entire set of collected images directly is not feasible due to these attribution requirements. To facilitate research transparency and reproducibility, the filenames of all JPEG files in this dataset have been compiled and are available in a text file within our GitHub repository.

Files that did not start or end with a valid JPEG marker (i.e., `0xFFD8` or `0xFFD9`) were removed. For bit-identical images, only one image was kept in the dataset.

4.2. JPEG dataset characteristics

Table 3 shows characteristics of the collected dataset. As can be seen, baseline JPEGs are by far the most frequently occurring type of JPEGs.

Marker occurrence. Markers in the entropy-coded data can be leveraged for validation. Progressive JPEG files contain additional SOS markers in the entropy-coded data. Both baseline and progressive JPEG files may contain Restart Markers (RST) in entropy-coded data. However, our data collection shows that the rate of occurrence of these markers is rare. Only 0.6% of the collected files are progressive, and only 1.0% of the files contain Restart Markers.

Chromatic subsampling. Chromatic subsampling is an optional JPEG feature that reduces the resolution of the chrominance color-channels (Cb, Cr). This reduces the amount of color information by a factor 2 (either horizontal or vertical subsampling) or 4 (both horizontal and vertical subsampling). Therefore, there are twice or even four times as many Huffman table lookups (during decoding) for luminance (Y) compared to a color channel (Cb, Cr). This would bias lookup errors to occur more frequently in Huffman tables for the chrominance channel. Nearly a quarter of all JPEGs in the dataset use a form of chromatic subsampling (see Table 3). The most common form is both horizontal and vertical subsampling.

[3] WFDownloader

[4] Smaller files fit into one default-sized NTFS block, which implies they cannot be fragmented (in default settings). The default block size on exFAT file systems varies with volume size, but is at least 4096 bytes, so the same reasoning applies.

Description	# files	% of total
JPEG files in dataset	230,157	100.0%
Baseline JPEG (SOF0)	228,793	99.4%
Progressive JPEG (SOF2)	1,364	0.6%
– using spectral selection	397	0.2%
– using successive approximation	967	0.4%
JPEGs that include or use		
Grayscale	26,359	11.5%
Restart markers	2,309	1.0%
Chromatic subsampling	53,128	23.1%
– horizontal subsampling	13,335	5.8%
– vertical subsampling	1,855	0.8%
– horizontal and vertical subsampling	37,937	16.5%

Table 3: JPEG dataset characteristics

4.3. Huffman code lengths

An important mechanism of validating JPEGs relies on Huffman table lookups (Sec. 3.2). If one or more of a JPEG's Huffman tables have short maximum code-lengths, a random bitstream continuation is more likely to trigger a lookup validation error. Therefore, it is relevant to know what Huffman table lengths occur in our dataset. The distribution of maximum Huffman table code lengths is shown in two figures, in Figure 3 for Luminance-DC and AC, and in Figure 4 for Chrominance-DC and AC.

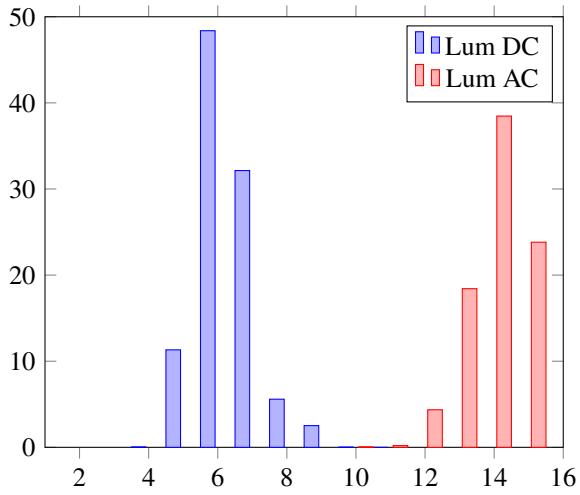

Figure 3: Maximum HT code lengths for luminance-DC and luminance-AC

5. Evaluation design

5.1. Validating the validator

We deliberately aimed to collect a dataset of JPEGs that would show a broad diversity in how the file format is used. Even if these violate the official JPEG specification, these files are in use in the real world and therefore we hold that, ideally, all these files should be correctly processed. We fed the validator process each file in the dataset individually. Our validator

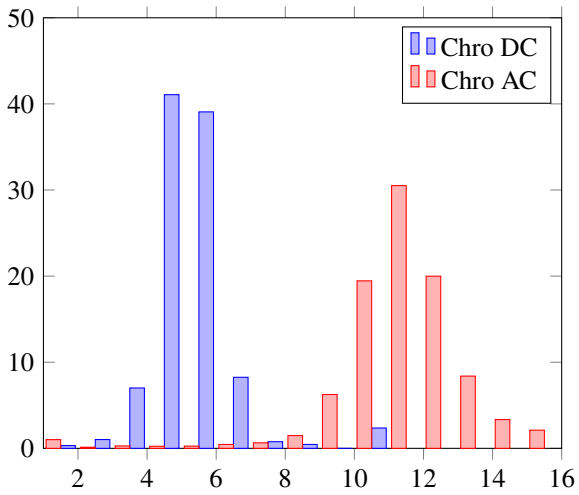

Figure 4: Maximum HT code lengths for chrominance DC and chrominance AC

(at the time of writing) is able to correctly process 230,149 of 230,157 files. The 8 remaining files exhibit rarely occurring deviations of the JPEG specification. If desired, support for these deviations can still be added in the future.

5.2. Choosing fragmentation points

To test the validator's efficacy, we will feed it a stream of bits from a JPEG file from the dataset, and then suddenly switch to a random bitstream.

There are three main criteria for determining at which point we want to break the input stream, which simulates a fragmentation point. First, we will only fragment JPEGs after the header, since our validation mechanism is only designed to validate high-entropy coded data sections. The second criteria is to align with the most frequently occurring block size. The default block size on both exFAT and (modern) NTFS file systems is at least 4 kB.[5] Lastly, for robustness, we opt to measure validation performance at two distinct fragmentation points. These criteria led us to test fragmentation after 16 kB and after 32 kB.

5.3. Post-fragmentation point data

An important experiment design consideration is what data is presented after a fragmentation point. JPEG's entropy-coded data exhibits high entropy. Various types of file formats tend to use compression, including audio, video, and office file formats. Compressed data is, by definition, high-entropy data and would resemble random data. Therefore, we generate and inject random data after the fragmentation point.

5.4. Experiment goals

The main goal of our experiments is to determine efficacy of the validation algorithms described in Sec. 3.4 and 3.6. That is, how well do they perform in identifying fragmentation points? We are interested in (1) how well they perform against the entire set, but also (2) the contribution of each validation mechanism to the validation result. Lastly, we want to evaluate (3) how well validation mechanisms perform in extreme scenarios.

1. Benchmarking validation success. We evaluate our implementation of the baseline validation algorithm, as introduced in our previous study, using the SOF0-set of baseline JPEGs. Additionally, we evaluate our extensions for progressive validation using the SOF2-set of progressive JPEGs. Since our dataset contains an order of magnitude more baseline than progressive JPEGs, the SOF0 set is 10× as large as the SOF2-set. We select subsets of 1,000 and 100 JPEGs, respectively, from the baseline and progressive sets, of at least 100 kB. Baseline JPEGs may optionally contain any of the following: restart markers, gray scale, or chromatic subsampled JPEGs. Progressive JPEGs may, in addition, also optionally use successive approximation.

2. Benchmarking the contribution of each validation mechanism. To this end, we keep track of which validation mechanism raised the error for all test-sets. We include the new coefficient length mechanism (Sec. 3.5), the various variants of Huffman lookup table and QA-overflow mechanisms, and existing JPEG file marker mechanisms. It's important to note that validation halts upon detecting the first failure, as any subsequent bits would be incorrect, rendering further validation pointless. These tests thus show which mechanisms trigger the soonest, not how well a validation mechanism performs. The latter would require a different kind of experiment, one where other validation mechanisms are excluded.

3. Benchmarking extreme cases. We consider 3 extreme cases for the validation mechanisms: HT-max, consisting of 100 JPEGs with the longest Huffman table codes from the data set; HT-min, consisting of 100 JPEGs with the shortest Huffman table codes; RST, consisting of 100 JPEGs containing restart markers.

The length of the Huffman table codes impacts the likelihood of a lookup error. Longer code length means more bit sequences are valid Huffman symbols. This implies that the longer the code length, the less likely a bit sequence from elsewhere triggers a lookup error. The longest code lengths thus constitute a worst-case scenario for this validation mechanism (HT-max set). Conversely, the shortest code lengths constitute a best-case scenario (HT-min set). Lastly, the RST-set is created to test RST-marker validation. Although restart markers are only present in 1.0% of all JPEG files in the dataset (Table 3), RST-markers are one of the few marker-based validation mechanism usable within the entropy-coded data sections (aside from SOS markers in progressive JPEGs).

5.5. Execution

Both the SOF0- and SOF2-sets underwent tests at two fragmentation offsets: 16 kB and 32 kB. This approach ensures any validation behavior unique to the 16 kB fragmentation point

[5] https://support.microsoft.com/en-us/topic/default-cluster-size-for-ntfs-fat-and-exfat-9772e6f1-e31a-00d7-e18f-73169155af95

will be revealed. For the other data sets, we conducted the tests using a single fragmentation offset, specifically at 16 kB from the start of the data, as opposed to the dual 16 kB and 32 kB offsets used for the SOF0- and SOF2-sets. Each file is tested 100 times, with each of these 100 tests performed with different random data after the fragmentation point. The number of tests performed per fragmentation offset is therefore 100,000 for the SOF0 set and 10,000 each for the SOF2, HT-max, HT-min, and RST-sets.

6. Results

The evaluation results are shown in Tables 4– 11.

6.1. Overall performance

Table 4 shows the performance of the algorithms in terms of correctly identifying the fragmentation point within a number of bytes. In Table 5, we present distribution of the location (in bytes) at which the validation algorithms detected the fragmentation points in the evaluation. Negative offsets indicate that the last known good location precedes the fragmentation point.

	SOF0 frag. point at		SOF2 frag. point at	
bytes	16 kB	32 kB	16 kB	32 kB
< 512	88.854%	88.878%	96.97%	93.15%
< 1,024	97.566%	97.607%	98.72%	97.28%
< 2,048	99.849%	99.853%	99.59%	99.21%
< 4,096	99.997%	99.999%	99.78%	99.92%
< 8,192	100.000%	100.000%	99.92%	100.00%
< 16,384	100.000%	100.000%	100.00%	100.00%
≥ 16,384	100.000%	100.000%	100.00%	100.00%

Table 4: Fragmentation point detection within given number of bytes

	SOF0 frag. point at		SOF2 frag. point at	
	16 kB	32 kB	16 kB	32 kB
Minimum	-3	-2	-1	-25
25th percentile	52	51	8	12
50th percentile (median)	131	129	25	36
75th percentile	285	284	77	124
95th percentile	729	729	340	684
99th percentile	1,272	1,255	1,393	1,963
99.9th percentile	2,193	2,193	5,712	3,773
Maximum	5,675	4,941	11,076	5,810

Table 5: Number of bytes from frag. point till validation error (distribution)

6.2. Contributions of individual validation mechanisms

Tables 6 and 7 show the contribution of each validation mechanism to fragmentation point detection. The results are split into two tables, since the validation mechanisms needed to be adapted, albeit slightly, for progressive JPEGs. In particular, Huffman encoding validation is performed in four distinct phases for progressive JPEGs (Sec. 3.6). Note that the *Coefficient length* validation mechanism did not trigger once in our experiments.

SOF0 validation type	frag. point at	
	16 kB	32 kB
Huffman-DC	25.944%	25.801%
Huffman-AC	0.893%	0.870%
QA-overflow	72.741%	72.979%
Restart marker	0.422%	0.350%
Start of scan marker	0.000%	0.000%
End of file marker	0.000%	0.000%

Table 6: Baseline (SOF0): Validation per validation type

SOF2 validation type	frag. point at	
	16 kB	32 kB
Huffman-DC First	47.87%	20.00%
Huffman-AC First	1.47%	1.07%
Huffman-DC Refine	0.00%	0.00%
Huffman-AC Refine	0.00%	0.43%
Coefficient length	0.00%	0.00%
QA-overflow	49.55%	76.13%
Restart marker	0.00%	0.00%
Start of scan marker	1.11%	2.37%
End of file marker	0.00%	0.00%

Table 7: Progressive (SOF2): Validation per validation-type

In addition, Table 8 splits the results per validation type and per channel (being luminance (Y), blueness (Cb), and redness (Cr) from the YCbCr color space).

Validation type	SOF0 frag. point at	
	16 kB	32 kB
Huffman-DC	25.944%	25.801%
– luminance (Y)	14.143%	13.956%
– blueness (Cb)	6.070%	6.108%
– redness (Cr)	5.731%	5.737%
Huffman-AC	0.893%	0.870%
– luminance	0.245%	0.282%
– blueness	0.478%	0.451%
– redness	0.170%	0.137%
QA-overflow	72.741%	72.979%
– luminance	68.910%	69.098%
– blueness	1.948%	1.941%
– redness	1.883%	1.940%

Table 8: Baseline (SOF0): Validation types per color channel

6.3. Validation performance for extreme cases

Tables 9, 10, and 11, are the counterparts to Tables 4, 5, and 6, respectively. These tables present the performance of the validation algorithm for extreme cases (HT-max,HT-min, and RST-sets).

7. Analysis of the results

7.1. Overall performance

Tables 4 and 5 show within how many bytes fragmentation was detected. Table 4 present this data in number of bytes used in block sizes on file systems; Table 5 shows the distribution of the number of bytes from the fragmentation point till a validation error occurred.

First, we consider the position of the fragmentation point: 16 kB vs. 32 kB. In a rare few SOF2 cases, fragmentation is not detected within 8,192 bytes from the 32 kB fragmentation point (it always is for the 16 kB point). However, the distributions associated with the two fragmentation points are rather similar, and differences in validation performance are tiny. We therefore conclude that no special validation behaviour should be expected at the 16 kB point.

Secondly, we examine performance on the level of file system blocks. In Table 4, we include the full range of allowed block sizes for completeness sake. The default block size on NTFS volumes is 4 kB; on exFAT volumes, 4 kB is the minimum size.[6] The main takeaway from the table then is that, for both SOF0 and SOF2 JPEGs, our validator is all but certain to determine fragmentation occurred within the first file system block following the fragmentation point. Moreover, the validation mechanism is fully deterministic. These two factors combined enable file carvers to use more efficient search strategies, since there is next to no uncertainty in the results of the validation mechanism.

Lastly we consider the distribution of the amount of bytes after the fragmentation point before a validation error is triggered. In Table 5, we see that the last known good location can occur before the fragmentation point. This can happen when JPEG-markers or Huffman table values start close to the fragmentation point. Such cases may end up not continuing / terminating correctly after the fragmentation point, triggering the validation error. The table also shows that the distributions are rather skewed. The 95th and 99th percentiles are within 2 kB from the fragmentation point, and even the 99.9th percentile for SOF0 JPEGs (the most frequently occurring type of JPEGs in our dataset) is slightly less than half the maximum value found.

[6] For exFAT volumes larger than 256 MB, the block size is at least 32 kB.

bytes	HT-max	HT-min	RST
< 512	79.74%	95.84%	84.20%
< 1,024	95.21%	99.72%	96.87%
< 2,048	98.81%	100.00%	99.89%
< 4,096	99.41%	100.00%	100.00%
< 8,192	99.74%	100.00%	100.00%
< 16,384	99.90%	100.00%	100.00%
< 32,768	100.00%	100.00%	100.00%
≥ 32,768	100.00%	100.00%	100.00%

Table 9: Fragmentation point detection within given number of bytes for HT-max, HT-min, and RST-sets

	HT-max	HT-min	RST
Minimum	-1	-1	0
25th percentile	94	41	78
50th percentile (median)	208	102	175
75th percentile	418	206	359
95th percentile	996	481	871
99th percentile	2,468	758	1,312
99.9th percentile	16,253	1,344	2,049
Maximum	32,766	1,621	2,705

Table 10: Number of bytes from frag. point till validation error (distribution)

Validation type	HT-max	HT-min	RST
Huffman-DC	5.61%	28.24%	10.91%
Huffman-AC	0.34%	0.93%	0.36%
QA-overflow	94.03%	70.83%	58.92%
Restart marker	0.00%	0.00%	29.81%
Start of scan marker	0.00%	0.00%	0.00%
End of file marker	0.02%	0.00%	0.00%

Table 11: Validation types for HT-max-, HT-min-, and RST-set

7.2. Contribution of individual validation mechanisms

In our previous study where we proposed the validation algorithm for SOF0 JPEGs, we hypothesized on the use of quantization array overflows that they "may sometimes yield a validation error". Indeed they do: In Tables 6 and 7, we see that the QA-overflow mechanism triggers first in ~73% of SOF0 cases and SOF2 cases with a 32 kB fragmentation point.

Interestingly, the SOF2 set at the 16 kB fragmentation point is an outlier. In this case, the Huffman-DC First validation mechanism shows unexpected strong performance. This may be linked to properties of progressive JPEGs. In particular, we recall that progressive JPEGs contain multiple scans of varying length, each of which allows the use of new Huffman tables. We suspect that the fact that Huffman tables may be replaced in progressive JPEGs is part of the reason, but finding a full explanation will require further experimentation.

The difference between the performance of the Huffman-DC and Huffman-AC mechanisms are explained by the fact that DC-tables have a maximum length of 11, whereas the AC-tables have a maximum length of 16. Amplifying this point, Figures 3 and 4 show that the DC-max lengths often are much shorter than their theoretical maximum length compared to their AC counterparts.

When evaluating the results per color-channel (Table 8), we see that validation errors are most frequently detected in the luminance channel. A factor that plays a role is chromatic subsampling. First, chromatic subsampling is relatively common (23.1% of SOF0+SOF2 JPEGs). Second, chromatic subsampling reduces the amount of chromatic data (compared to luminance data) by either a factor of 2 or a factor of 4. The number of Huffman table lookups for chrominance are thus reduced by the same factor.

When considering the QA-overflow mechanism, we see an even more dominant result for the luminance channel.

This is not surprising: chrominance is often much more compressed than luminance in the quantization tables (see e.g., ITU/CCIT/JPEG (1992, Table K.1, K.2)). Chrominance quantization arrays therefore typically contain significantly less non-zero values than luminance arrays. They are therefore more likely to contain a "fill out with zeroes" command, which fills out the quantization array completely and correctly, thereby avoiding triggering a QA-overflow.

7.3. Validation performance for extreme cases

The HT-max set represents a worst-case scenario for Huffman-based validation mechanisms. Validation for this set is indeed significantly hindered, as shown in Tables 9, and 10. Table 11 shows clearly that the otherwise so reliable Huffman-DC validation mechanism is kneecapped by this set. In several test cases, the End of File marker mechanism was even triggered before any other validation mechanism. This implies that the entire (fragmented) bitstream was considered valid, except only for the absence of an End of File marker at the correct bit.

Conversely the HT-min set constitutes a best-case scenario for this mechanism. The distribution of number of bytes before a validation error was triggered is only slightly better than for the SOF0 set (Table 5 vs. Table 10). The performance of the Huffman-DC mechanism is only slightly better in this best-case scenario than for the SOF0 set (Table 6 vs. Table 11). Apparently, the common case is thus not far off from the best-case scenario.

Table 11 shows that ~30% of all validation errors in the RST-set are due to the RST validation mechanism. Even in a dataset where all files contain restart markers, the QA-overflow mechanism remains the dominant validation mechanism.

8. Related work

Since the early 2000s, the recovery of deleted computer files has been a topic of extensive research, as highlighted by Pal and Memon (2009). Recovery strategies primarily fall into two main categories: file structure and content-based approaches, though there are other methods and combinations of these methods as well.

JPEG file structure. Research on recovering JPEG files often rely on the identification of JPEG markers. The work by Mohamad and Deris (2009) focuses on the Define Huffman Table (DHT) marker in a JPEG files. By analyzing the length fields of the DHT marker, they determined how to validate DHT data, highlighting fragmentation points if the subsequent data fails this validation. The effectiveness of this method hinges on the frequency of fragmentation within the JPEG's DHT section.

In a study on thumbnail carving using image pattern matching, Abdullah et al. (2013) use the fact that each JPEG starts and ends with a Start of image (SOI) and End of image marker (EOI). Thumbnails, smaller versions of the original picture, found within the JPEG header, also use these markers. Therefore, a JPEG might house multiple SOI/EOI pairs. When tested on the DFRWS 2006 and 2007 datasets, their file carver flagged 4 files incorrectly as thumbnails but surpassed a benchmark algorithm, recovering 31 compared to the latter's 28 files.

In their work, Fei and Abdullah (2020) introduced a file carver tailored for the recovery of in-order fragmented JPEGs, using the file's structure. When tested on the DFRWS 2006 dataset, this carver successfully retrieved 8 of the 12 JPEG images.

Content-based approach. The recovery of JPEG files can also be approached through analyzing the visual representation of its data—a content-based strategy. Most studies interpreting potential JPEG image data often incorporate knowledge of the JPEG file structure to optimize their analyses. Memon and Pal (2006) focus on file fragment classification and present techniques for image reconstruction. Using a greedy search algorithm, the sum of differences metric outperformed a pixel matching strategy for file recovery purposes. Li et al. (2011) delve into artifacts emerging from fragmented or corrupted data. Notably, they highlight the DC-values, which are delta encoded, as indicators of sudden color shifts. Furthermore, they demonstrate that the distribution of AC-values can signal errors in JPEG data. Uzun and Sencar (2015) propose a method to infer Huffman tables, subsampling ratios, and quantization values for dealing with a missing JPEG header. They analyzed statistics from photos uploaded to Flickr to determine common values for JPEG decoding metadata. Given that 99.5% of the Flickr dataset was encoded with default Huffman tables, their method offers a reliable way to discern the remaining decoder settings for JPEG fragments. Birmingham et al. (2017) leverage the embedded thumbnail within the JPEG header to predict the primary image's characteristics. By applying a probabilistic model centered on thumbnail affinity, they showcase this method's capacity to pinpoint invalid JPEG data.

Recovery related to metadata. Metadata may be derived from image data, and be used for image identification. In their research, Thai et al. (2017) proposed a method to estimate quantization steps for an image originally compressed as JPEG but later saved in a lossless format. Through their analysis, they demonstrated that a fingerprint technique could suggest potential true quantization steps, achieving over 99% identification accuracy on grayscale images from the Dresden database (Gloe and Böhme, 2010).

Unlike traditional digital cameras, smartphones frequently undergo software updates and setting modifications. In a comprehensive study on Apple smartphones, Mullan et al. (2019) explored how evolving software might influence source identification. Using machine learning, they devised classifications from EXIF data and quantization matrices. Their findings reveal that while EXIF headers and JPEG quantization table values can effectively differentiate specific apps or OS versions, identifying images from smartphones proves more challenging than from standard digital cameras.

Statistics oriented. Another perspective on JPEG recovery involves statistical analysis. Kadir et al. (2015) employed statistical byte frequency analysis to distinguish groups of JPEG fragments, noting that each image exhibits distinct characteristics.

Their study on 4 JPEG files indicated that byte frequency analysis unveiled multiple unique patterns. Taking a similar route, Tang et al. (2016) introduced a novel similarity metric, the Coherence of Euclidean Distance (CED), to determine if two data blocks belong to the same JPEG. Their results showed the CED algorithm outshining the Adriot Photo Forensics (APF) in file recovery. For 3-piece JPEGS, CED recovered 96 out of 109 files, whereas APF managed 66. For 4-piece JPEGS, CED retrieved 61 out of 75, with APF securing only 32. Lastly, Azhan et al. (2022) developed the Error Level Analysis technique to pinpoint the distinct signature of 8x8-pixel JPEG blocks. Their tests on 21 JPEG images demonstrated the uniqueness of each block.

Camera sensor information. Each camera's sensor introduces unique noise to an image. This sensor noise can determine if an image originated from a specific camera (Lukás et al., 2006). Building on this concept, Durmus et al. (2017) demonstrated how JPEG fragments can be both attributed to a particular camera and pinpointed to their location within an image, assuming the originating camera is known. To verify fragment correctness, the researchers employed Sum-of-Differences and Histogram Differences. In tests, their method achieved a true positive rate of 94.2%, correctly identifying 21,713 out of 23,040 fragments. In a subsequent study, Durmus et al. (2019) noted the limitations of their earlier work, especially its potential weaknesses under real-world conditions due to overlooked brightness and color artifacts. To address this, they introduced a compatibility metric for fragment matching and subsequent image stitching. They tested their approach on 2,000 images from a single camera, all converted to JPEG with identical quality settings. Results showed a 52.4% correct fragment identification rate for JPEGs at a quality factor of 90, and a 42.0% identification rate for those at 80.

JPEG file carvers (other). de Bock and de Smet (2016) presented a novel file carving approach, implemented in the tool JPGcarve, which employs an external decoder library (libjpeg-turbo) as a validation mechanism for the JPEG data. While validating JPEG data, the decoder either processes it successfully or fails, indicating a fragmentation point. The file carver itself includes support for single- and multifragment file recovery, and search space reduction techniques. In tests across six datasets, JPEGcarve successfully recovered all multi-fragmented JPEGs, totaling 46 images.

Further advancing file carving methods, Ali and Mohamad (2021) introduced RX_myKarve, combining the Extreme Learning Machine and JPEG structure validation. This dual approach classifies file fragments to distinguish between JPEG and non-JPEG fragments. Subsequent structure-based carving aids in JPEG reconstruction. The authors highlight its efficacy, noting the recovery of all 19 images from the DFRWS 2006 dataset and 18 from the DFRWS 2007 dataset.

Generic file recovery. Not all JPEG recovery techniques are exclusive to the JPEG file format. Generic file recovery approaches can sometimes be applicable to JPEG. For example, Ying and Thing (2010) posed file fragment reconstruction as a graph-theoretic challenge. In a test involving 10 files, their method surpassed a brute-force technique, successfully restoring all files to their original state.

In a study on hash-based file carving, Garfinkel and McCarrin (2015) introduce a modified whole file hashing approach. While the hash value of known files can identify intact files, this method struggles with fragmented, altered, or incomplete files. With their hash-based carving technique, centered on individual data blocks of a target file and leveraging a target hash database, the authors demonstrate the feasibility of this approach.

Leveraging Convolutional Neural Networks (CNN), Ghaleb et al. (2023) unveil a light-weight file fragment classification model. They report enhanced time efficiency and comparable accuracy to earlier CNNs, achieving 79% accuracy on the FFT-75 dataset. However, the team echoes prior findings, pointing out the challenge in classifying high-entropy file fragments due to their lack of distinguishable statistical patterns.

9. Conclusion

We implemented the algorithm for fragmentation point detection in baseline JPEGs, initially introduced in our previous publication (van der Meer and van den Bos, 2021). Additionally, we have adapted this algorithm to accommodate progressive JPEGs. To rigorously evaluate our validator, we assembled a comprehensive test set comprising over 230,000 JPEG files by scraping WikiMedia. This test set encompassed a diverse range of variations in JPEG files, including differences in cameras, encoders, and encoder settings. We tested our validator thoroughly, in different scenarios where each JPEG file was tested 100 times, with different random bitstreams following the fragmentation point. For each test scenario, we focused on specific JPEG properties and specific fragmentation points. Of all the validation mechanisms, the QA-overflow most often was the mechanism that triggered first.

Considering the combined performance of all validation mechanisms, in the worst case scenario, our implementation has over 99.4% probability of correctly invalidating an incorrect bitstream within 4096 bytes of the fragmentation point. For the most common case of baseline JPEGs, the validator achieves over 99.99%. These results are, frankly, astounding – especially given the diversity and quantity of the test set. Therefore, we consider the problem of finding JPEG fragmentation points solved in practice.

Future Work. The presented algorithms allow recovery of fragmented high-entropy data for the JPEG file format. The impressive success rate invites investigating the applicability of bit-level validation as a fragmentation point detection mechanism for other file formats laden with significant amounts of high-entropy data.

Secondly, we are currently in the process of incorporating the JPEG validator into a novel file carving framework.

Acknowledgements.. Van der Meer was supported by the Netherlands Organisation for Scientific Research (NWO) through Doctoral Grant for Teachers number 023.012.047.

References

Abdullah, N.A., Ibrahim, R., Mohamad, K.M., 2013. Carving thumbnail/s and embedded jpeg files using image pattern matching. Journal of Software Engineering and Applications 6, 62.

Ali, R.R., Mohamad, K.M., 2021. Rx_mykarve carving framework for reassembling complex fragmentations of jpeg images. Journal of King Saud University-Computer and Information Sciences 33, 21–32.

Azhan, N.A.N., Ikuesan, R.A., Razak, S.A., Kebande, V.R., 2022. Error level analysis technique for identifying jpeg block unique signature for digital forensic analysis. Electronics 11, 1468.

Birmingham, B., Farrugia, R.A., Vella, M., 2017. Using thumbnail affinity for fragmentation point detection of JPEG files, in: 17th International Conference on Smart Technologies (IEEE EUROCON), IEEE. pp. 3–8.

de Bock, J., de Smet, P., 2016. Jpgcarve: An advanced tool for automated recovery of fragmented JPEG files. IEEE Transactions on Information Forensics and Security 11, 19–34.

Durmus, E., Korus, P., Memon, N.D., 2019. Every shred helps: Assembling evidence from orphaned JPEG fragments. IEEE Trans. Inf. Forensics Secur. 14, 2372–2386.

Durmus, E., Mohanty, M., Taspinar, S., Uzun, E., Memon, N.D., 2017. Image carving with missing headers and missing fragments, in: 2017 IEEE Workshop on Information Forensics and Security, WIFS 2017, Rennes, France, December 4-7, 2017, IEEE. pp. 1–6.

Fei, T.K., Abdullah, N.A., 2020. Data carving linearly fragmented jpeg using file structured based technique. Applied Information Technology And Computer Science 1, 173–180.

Garfinkel, S.L., McCarrin, M., 2015. Hash-based carving: Searching media for complete files and file fragments with sector hashing and hashdb. Digital Investigations 14 Supplement 1, S95–S105.

Ghaleb, M., Saaim, K.M., Felemban, M., Alsaleh, S., Almulhem, A., 2023. File fragment classification using light-weight convolutional neural networks. CoRR abs/2305.00656.

Gloe, T., Böhme, R., 2010. The dresden image database for benchmarking digital image forensics. Journal of Digital Forensic Practice 3, 150–159.

Hudson, G., Léger, A., Niss, B., Sebestyén, I., Vaaben, J., 2018. JPEG-1 standard 25 years: past, present, and future reasons for a success. Journal of Electronic Imaging 27, 040901.

ITU/CCIT/JPEG, 1992. Recommendation T.81: Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines. Technical Report. International Telecommunication Union.

Kadir, N.F.A., Abd Razak, S., Chizari, H., 2015. Identification of fragmented jpeg files in the absence of file systems, in: 2015 IEEE Conference on Open Systems (ICOS), IEEE. pp. 1–6.

Karresand, M., Shahmehri, N., 2008. Reassembly of fragmented jpeg images containing restart markers, in: 2008 European Conference on Computer Network Defense, IEEE. pp. 25–32.

Li, Q., Sahin, B., Chang, E., Thing, V.L.L., 2011. Content based JPEG fragmentation point detection, in: IEEE International Conference on Multimedia and Expo (ICME), IEEE. pp. 1–6.

Lukás, J., Fridrich, J.J., Goljan, M., 2006. Digital camera identification from sensor pattern noise. IEEE Trans. Inf. Forensics Secur. 1, 205–214.

van der Meer, V., van den Bos, J., 2021. JPEG file fragmentation point detection using huffman code and quantization array validation, in: ARES 2021: The 16th International Conference on Availability, Reliability and Security, Vienna, Austria, August 17-20, 2021, ACM. pp. 46:1–46:7.

van der Meer, V., Jonker, H., van den Bos, J., 2021. A contemporary investigation of ntfs file fragmentation. Forensic Science International: Digital Investigation 38, 301125.

Memon, N.D., Pal, A., 2006. Automated reassembly of file fragmented images using greedy algorithms. IEEE Transactions on Image Processing 15, 385–393.

Mohamad, K.M., Deris, M.M., 2009. Fragmentation point detection of jpeg images at dht using validator, in: First International Conference on Future Generation Information Technology (FGIT), Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 173–180.

Mullan, P., Riess, C., Freiling, F., 2019. Forensic source identification using jpeg image headers: The case of smartphones. Digital Investigation 28, S68–S76.

Pal, A., Memon, N., 2009. The evolution of file carving. IEEE Signal Processing Magazine 26, 59–71.

Tang, Y., Fang, J., Chow, K., Yiu, S., Xu, J., Feng, B., Li, Q., Han, Q., 2016. Recovery of heavily fragmented jpeg files. Digital Investigation 18, S108–S117.

Thai, T.H., Cogranne, R., Retraint, F., Doan, T., 2017. JPEG quantization step estimation and its applications to digital image forensics. IEEE Transactions on Information Forensics and Security 12, 123–133.

Uzun, E., Sencar, H.T., 2015. Carving orphaned JPEG file fragments. IEEE Transactions on Information Forensics and Security 10, 1549–1563.

Ying, H., Thing, V.L.L., 2010. A novel inequality-based fragmented file carving technique, in: Third International ICST Conference, e-Forensics, Springer. pp. 28–39.