JPEG File Fragmentation Point Detection using Huffman Code and Quantization Array Validation

Vincent van der Meer
vincent.vandermeer@zuyd.nl
Zuyd University of Applied Sciences
Heerlen, The Netherlands

Jeroen van den Bos
j.van.den.bos@nfi.nl
Netherlands Forensic Institute
The Hague, The Netherlands

ABSTRACT

File carving is a data recovery technique used in many investigations in digital forensics, with some limitations. Especially JPEG files are difficult to recover when fragmented, because they consist almost entirely of large blobs of highly compressed entropy-coded data, with no clearly discernible structure.

This paper describes an approach that leverages two observations about many JPEG files in practice. First, the Huffman tables used to decode a large proportion of the entropy-coded data often do not use all possible code values at their longest code length, offering possibilities to detect errors when invalid codes are encountered. Second, after translating Huffman codes to symbols, the next step in decoding involves filling quantization arrays with exactly 64 values, offering another possibility to detect errors when an overflow is encountered.

This paper presents an algorithm to validate the entropy-coded data using these two observations and finds that the odds of finding fragmentation points are quite high, especially with regard to invalid Huffman codes. It will work with the example Huffman tables provided by the JPEG standard that are used by many digital cameras, but also with many optimized Huffman tables generated by specialized applications.

CCS CONCEPTS

• **Applied computing** → **Data recovery**; **Evidence collection, storage and analysis**.

KEYWORDS

Digital forensics, Data recovery, File carving, JPEG validation

ACM Reference Format:
Vincent van der Meer and Jeroen van den Bos. 2021. JPEG File Fragmentation Point Detection using Huffman Code and Quantization Array Validation. In *The 16th International Conference on Availability, Reliability and Security (ARES 2021), August 17–20, 2021, Vienna, Austria.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3465481.3470061

ARES 2021, August 17–20, 2021, Vienna, Austria

ACM ISBN 978-1-4503-9051-4/21/08…$15.00
https://doi.org/10.1145/3465481.3470061

1 INTRODUCTION

Many investigations in digital forensics rely on data recovery. Apart from suspects hiding or removing information, accidents or acts of violence often physically damage devices making their contents no longer directly accessible. A key technique in data recovery is file carving, which is a process of recovering files that is not based on metadata (e.g., such as file system information) but on recognizing the internal structure of file types. Using this technique, many types of files can be recovered quickly and reliably from a raw copy of a device's contents.

A complication that often arises when file carving is file fragmentation: file systems commonly split files into multiple pieces for performance reasons. Basic file carvers, that look for the magic values in the headers and footers of many file types cannot recover fragmented files. To recover fragmented files, more advanced carving tools are required that reconstruct fragmented files. However, this requires a method to determine where the actual fragmentation has occurred (i.e., where the end of a fragment is located).

For this purpose a file validator is employed: a recognizer for a specific file format that, when given a potential file, responds whether it indeed qualifies as a valid instance of that file type and ideally also provides detailed information about where an error occurred if the answer is no. Whether it is complicated or even possible to construct a validator for a specific file type that can accurately determine the location of such a *fragmentation point* depends on the internal structure of that file type.

Arguably the most important file type in digital forensics is JPEG: it is the most used for images in any media, on digital cameras, on websites and social media and in any other type of application. In order to effectively carve JPEG files, a file validator is needed that is capable of accurately identifying the fragmentation point when reading from the start of a JPEG fragment. JPEG has a fairly simple file structure, but because almost its entire contents is not guarded by values such as length fields, checksums or other easily identifiable markers, constructing a file validator with this capability for it is non-trivial.

In this paper we present a novel approach to validate the entropy-coded data of JPEG files based on detecting invalid Huffman codes and quantization array size overflows. Our contributions consist of both the two validation approaches as well as the described algorithm that incorporates both. In the following section we discuss the background of both file fragmentation and the file structure of JPEG. Next, we discuss some observations that lead to our description of the validation algorithm, followed by the description of an example showing how the algorithm works in practice. Finally, we analyze the success rate, discuss limitations, considerations and related work.

2 BACKGROUND

2.1 File Fragmentation in Practice

File fragmentation occurs in many situations: file systems use it on storage media, protocols in network streams and operating systems in how they allocate memory. Since file carving is performed most commonly on storage media such as hard drives and flash chips, we focus here on aspects specifically relevant to file systems, but the principles apply everywhere.

File systems allocate files on storage media. The smallest amount of data they allocate is called a block. For the NTFS file system commonly used on Windows systems, the default size of these blocks is 4096 bytes for volumes up to 16 TB[1]. So whenever a file is stored in a file system, it takes up at least one single block[2] but in many cases it will take up multiple blocks, up to thousands or millions for extremely large files. File systems use this technique in order to support very large storage devices with relatively modest metadata overhead.

A file is considered fragmented when its blocks are not stored both contiguously and in-order. Files thus need to have two blocks or more allocated to them to become susceptible to file fragmentation. A file fragment consist of one or more contiguous blocks that form a part of a file. This type of fragmentation occurs because of churn on the file system: when a system is in use, its operating system regularly creates, resizes and removes files. When storing and resizing a file on a volume with a lot of activity, the operating system often has no other choice but to cut it into fragments in order to store it in the available space.

Some research has been done on analyzing the fragmentation rate on storage media. In a data set that consists of file systems gathered between 1998 and 2006, Garfinkel [2] reports a 6% fragmentation rate. Meyer and Bolosky [6] report a 4% fragmentation rate in 2012. In a 2019 data set, Van der Meer et al. [12] report a 2.2% fragmentation rate for all files, and a 4.4% fragmentation rate for files that can be fragmented. Despite a slowly decreasing fragmentation rate, the amount of fragmented data has increased in that time period due to exponential increases in average storage media size.

2.2 The Structure of JPEG

JPEG is a standard for lossy digital image compression, and uses various transition and encoding techniques to transform pixel color values to highly compressed image data. The JPEG standard is extensive and offers many extension points and implementation options, but in practice nearly all files encountered in practice use lossy compression using Huffman encoding and are serialized into one of two popular file formats that adhere to the JPEG Interchange Format (JIF). These two formats are the JPEG File Interchange Format (JFIF) and an extension of it that is popular among digital cameras called Exchangeable Image File Format (EXIF). Two additional common variations exist: baseline and progressive encoded files. A baseline JPEG encodes the entire image in a single encoded stream, while a progressive JPEG contains several streams, each iteratively adding detail until the final image is formed.

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

[2] Except for small files that can sometimes be stored in the file system's metadata.

A typical JPEG file is the result of the following transformations:

(1) Color conversion from RGB to YCbCr. This transformation is lossless, as it only separates brightness from color data.
(2) 2D Discrete Cosine Transform (DCT), which is also lossless.
(3) Quantization of DCT-coefficients, using a quantization matrix. This transformation is lossy.
(4) The quantization matrix is mapped using a zig-zag pattern, and encoded with either delta (the first coefficient) or run-length (the remaining coefficients) encoding.
(5) Huffmann encoding to further compress the remaining data.

JPEG files are self-contained since they contain the values of the quantization tables along with information needed to construct the Huffman tables. The structure of a JPEG file follows a common pattern for file formats with magic values denoting the type of serialized data structure and length fields to simplify parsing. The magic values all start with byte value 0xFF followed by a byte that specifies its meaning:

0xD8: Start-of-Image (SOI), which describes version and other metadata. This marker should always appear at the start of the file.
0xDB: Define Quantization Table (DQT), provides the contents of one quantization table. A file may (and commonly does) contain multiple quantization tables.
0xC0: Start-of-Frame (SOF), which specifies how many frames are used in this encoding. One for a baseline encoded JPEG, up to eleven for a progressive encoded JPEG.
0xC4: Define Huffman Table (DHT), which specifies the symbols and the frequency of the code-lengths from which the resulting Huffman codes must be derived. Typically four Huffman tables are specified, which very often are the example tables provided by the specification [3].
0xDA: Start-of-Scan (SOS), a description of the compressed stream that directly follows this data structure.
0xD9: End-of-Image (EOI), should always appear at the end of the file.

The order in which these markers occur is mostly fixed, i.e., SOI is always the first marker, SOF always precedes SOS, and EOI is always last. However, these data structures constitute only a few kilobytes of a typical JPEG file of several megabytes as produced by any modern mobile phone or camera. The rest of the data is in one or more so-called *Scans* following the SOS structure and terminating right before the EOI. Within this entropy-coded data some escaped values (starting with 0xFF) are allowed, but these are rare in practice.

As a result, any JPEG file larger than a couple of kilobytes will consist almost entirely of entropy-coded data, which is highly compressed and does not contain any checksums, length fields or other hints to easily determine whether the contents has been corrupted or fragmented. A single exception to this are restart markers, which is an optional mechanism specified by the JPEG standard that allows numbered markers to be inserted into the entropy-coded data specifically to detect errors. Unfortunately, the mechanism is optional and it is rarely encountered in practice (in our experience, but also as reported by Tang et al.[11]).

3 FRAGMENTATION POINTS IN JPEG

In order to successfully carve fragmented JPEG files, a validator is needed that can accurately pinpoint where inside a provided candidate file the fragment ends. This will allow the file carver to remove or reshuffle the blocks around only that location in order to more quickly find a valid file. The more precise the validator's information is, the smaller the search space that the file carver needs to consider.

If the fragmentation occurs near the start of a JPEG file, then this is fairly easy: the first data structures have clear markers, length fields and a well-defined structure. Unfortunately, this only concerns the first few blocks as the rest of a JPEG file is mostly entropy-coded, highly-compressed data. This paper describes a method to easily and quickly validate many parts of this entropy-coded data by looking for invalid Huffman codes and possible size overflows when constructing quantization arrays.

3.1 Invalid Huffman Codes

A large proportion of the entropy-coded data consists of Huffman codes, which when mapped to their symbol are either directly used or refer to a value in the data following the code. Because of this it is difficult to directly describe the percentage of data that consists of Huffman codes, since both Huffman codes and values have variable sizes. In practice, it seems reasonable to expect more than 50% of the entropy-coded data to consist of Huffman codes.

Instead of storing the Huffman tables directly, the DHT data structure describes the symbols and a list of code lengths to be generated by the decoder. Since this algorithm is fixed in order to guarantee interoperability, the resulting Huffman table is always the same and the DHT structure is just a form of compression.

These Huffman tables consist of a list of codes of variable length and their accompanying symbols. Whenever a code is encountered in the input, it is translated to its symbol for further interpretation. An example of such a table is shown in Table 1, which shows one of the default Huffman tables provided by the JPEG standard (and as such, used by many encoders).

Code	Length
`00`	2
`010`	3
`011`	3
`100`	3
`101`	3
`110`	3
`1110`	4
`11110`	5
`111110`	6
`1111110`	7
`11111110`	8
`111111110`	9

Table 1: One of the default Huffman tables

Huffman codes are generated in such a way that there is never any ambiguity during decoding: the shortest possible matching stream of bits that matches a code always maps to that code, since there is never a longer code with that value as prefix. An observation that holds on many Huffman tables (including all the default tables provided by the JPEG standard) is that often not all combinations of the longest code length are all used, such as the code `111111111` (of length 9) which does not appear in Table 1.

This is actually a side-effect of design choice in the algorithm to keep its complexity low. For example, in Table 1, it would have provided a marginally better compression rate to have two tokens of size 8, instead of one of size 8 and one of size 9. This would have kept the table identical, except that it would have eliminated the trailing `0` on the longest code and would have ensured that all codes of valid sizes could be translated. In many cases, it would simply not be useful to map each possible code at the highest length to a useful symbol, leaving the possibility for codes of valid length to exist that do not map to any symbol.

Due to this design, there is an opportunity to validate a stream that uses a Huffman table with this characteristic. In the above example of Table 1, whenever a Huffman code of length 9 is encountered, this value must be `111111110`, since `111111111` is of the same length but not defined in the Huffman table.

3.2 Quantization Array Size Overflows

The process of decoding Huffman codes into symbols leads to values that must be stored in a quantization array. Each first code maps to a length value that describes a value that appears in the data after the Huffman code, which must be stored in the first location in the quantization array. The following Huffman codes map to symbols that each encode two 4-bit values in a single byte. These two values (along with possibly some additional bits in the input) represent a run-length encoded description of one or more values to be inserted into the quantization array.

This process leads to another possibility to detect errors in the entropy-coded data. If the description leads to an overflow in filling out the quantization array, an error has been detected in the data.

An example of such an error is that if 62 of the 64 values in the array has been filled, only a few possible values are allowed: two descriptions of two individual values of a single byte, a run-length encoded description with a size that equals 2 bytes or the terminator special value that denotes that the rest of the array consists of zeros. Any other value that appears is invalid since it would lead to an overflow of the specified quantization array size.

3.3 Entropy-coded Data Validation

When validating the entropy-coded data, some considerations need to be taken. First, JPEG allows escaped values in the entropy-coded data, that always start with `0xFF`, so whenever at the start of a full byte this value is encountered, it needs to be interpreted according to the rules defined. This can include restart markers, a literal byte value of `0xFF` or another marker (only EOI is allowed).

Any incorrect values encountered at this level will immediately lead to a validation error. Examples are a restart marker without the header of the JPEG specifying them or a restart marker with an incorrect increment. Additionally, an escaped value that does not need to be escaped or an EOI marker not at the end of a logical block of decoded data both also denote a validation error.

The smallest amounts of data that can be decoded together are called Minimum Coded Units (MCU). In our example we will assume a non-chromatic subsampled baseline JPEG. The principles apply to all other common variants, but are left out here to simplify the description.

An MCU consists of 3 color channels (Y, Cb, Cr), and each color channel has 64 coefficients stored in a 8*8 matrix. The first value of this matrix is a DC value, and decoded using a Huffman table specifically defined for DC values. The next 63 values are AC values, and decoded using a separate Huffman table specific to AC values. As such, an MCU can be seen as a 3*8*8 matrix (or as 3 8*8 matrices).

Validation is done with the following algorithm for each MCU in the entropy-coded data.

For 3 iterations: (once for each matrix in the MCU), perform the following steps:

1. Read bits until the shortest valid Huffman code is encountered, use DC Huffman table to convert to symbol.
   If not found: terminate with error, report location.
2. Interpret the symbol as integer and skip this amount of bits, (re)set counter to 1.
3. Until 63 AC values have been encountered, perform the following steps:

3.1. Read bits until the shortest valid Huffman code is encountered, use AC Huffman table to convert to symbol.
If not found: terminate with error, report location.

3.2. Interpret the symbol and unpack the run-length encoded values:

3.2.1. If the symbol is `0x00` the quantization array is complete, set counter to 64, skip to next iteration.

3.2.2. If the symbol is `0xF0` the total amount of unpacked values becomes 16.

3.2.3. For all other values, the total amount of unpacked values becomes the value of the upper nibble interpreted as integer + 1. In addition, interpret the lower nibble as integer and skip this amount of bits.

3.3. Add the total amount of the unpacked values to a counter.
If counter > 64 terminate with error, report location.

The error checks are the bold-faced If-statements in the description above. If they evaluate to true, an error has been encountered in the entropy-coded data and the location of the byte where the associated data was read from is reported as the first location where validation failed. The block this location resides in can then be considered to not be part of a valid JPEG file and a fragmentation point has been identified: the end of the directly preceding block (where validation did succeed).

4 EXAMPLE AND ANALYSIS

In this section we provide an example that shows the validation of a single MCU from a JPEG file, as a practical illustration of the algorithm. Additionally, we analyze how quickly the algorithm would be expected to detect an error when encountering incorrect data. Finally, we discuss some of the considerations and limitations of applying this approach in practice.

4.1 Example: Validating a Single MCU

Table 4 contains a representation of 10 bytes that together make up a single MCU from a JPEG file generated by the GIMP application on Linux[3]. Since GIMP generates optimized Huffman tables, the example MCU does not use the standard tables as defined by the JPEG standard. Table 2 (for DC values) and Table 3 (for AC values) show the generated optimized Huffman tables as decoded from the JPEG file that contains the example MCU. In this case, a total of 4 Huffman tables were generated, which is common.

Furthermore, the file is a baseline (as opposed to progressive) encoded JPEG that does not employ chroma subsampling. As discussed before, all principles apply across all common types of JPEG files, but this configuration was chosen because of its relative simplicity for this example.

DC#0		DC#1	
Code	Symbol	Code	Symbol
0	00000101	00	00000000
10	00000100	01	00000001
110	00000110	10	00000100
1110	00000010	110	00000010
11110	00000011	1110	00000011

Table 2: Decoding example: DC Huffman tables

AC#0		AC#1	
Code	Symbol	Code	Symbol
00	00000001	00	00000000
01	00000010	01	00000001
100	00000011	10	00010001
1010	00000000	110	00000010
1011	00000100	11100	00000011
1100	00010001	11101	00010010
1101	00100001	111100	00100001
11100	00010010	111101	00110001
11101	00110001	1111100	00010011
111100	01000001	1111101	00100010
1111010	00100010	1111110	01000001
1111011	01100001	11111110	01010001
11111000	00000101		
11111001	00010011		
11111010	00010100		
11111011	00010101		
11111100	00100011		
11111101	01010001		
11111110	10100001		
111111110	10110001		

Table 3: Decoding example: AC Huffman tables

Table 4 has five columns. The first column (Offset) contains the relative offset of the value in the MCU. The second column (Values) contains the value of the byte(s) at that offset in binary encoding. Two bytes are shown when applicable, and this illustrates clearly how Huffman-compression crosses byte boundaries. The

[3]We omit a detailed description of the process and settings since any JPEG file that employs Huffman compression should suffice for this example.

Offset(s)	Values	Huffman table lookup		Interpretation	Counter
0	11001111	DC#0:	110 ⇒ 00000110	One value: skip 6 bits, counter = 1	#1=1
0, 1	11001111 11011110			6 bits skipped	
1	11011110	AC#0:	1011 ⇒ 00000100	No zeros, one value: skip 4 bits, counter+1	#1=2
1, 2	11011110 00110111			4 skipped bits	
2	00110111	AC#0:	01 ⇒ 00000010	No zeros, one value: skip 2 bits, counter+1	#1=3
2	00110111			2 skipped bits	
2, 3	00110111 00010010	AC#0:	11100 ⇒ 00010010	One zero, one value: skip 2 bits, counter+2	#1=5
3	00010010			2 skipped bits	
3	00010010	AC#0:	00 ⇒ 00000001	No zeros, one value: skip 1 bit, counter+1	#1=6
3	00010010			1 skipped bit	
3, 4	00010010 00001010	AC#0:	00 ⇒ 00000001	No zeros, one value: skip 1 bit, counter+1	#1=7
4	00001010			1 skipped bit	
4	00001010	AC#0:	00 ⇒ 00000001	No zeros, one value: skip 1 bit, counter+1	#1=8
4	00001010			1 skipped bit	
4	00001010	AC#0:	01 ⇒ 00000010	No zeros, one value: skip 2 bits, counter+1	#1=9
4, 5	00001010 11111001			2 skipped bits	
5	11111001	AC#0:	111100 ⇒ 01000001	4 zeros, one value: skip 1 bit, counter+5	#1=14
5	11111001			1 skipped bit	
6	11101011	AC#0:	11101 ⇒ 00110001	3 zeros, one value: skip 1 bit, counter+4	#1=18
6	11101011			1 skipped bit	
6, 7	11101011 00110100	AC#0:	1100 ⇒ 00010001	1 zero, one value: skip 1 bit, counter+2	#1=20
7	00110100			1 skipped bit	
7	00110100	AC#0:	1010 ⇒ 00000000	Fill with zeros, counter=64, next array	#1=64
7, 8	00110100 11010000	DC#1:	01 ⇒ 00000001	skip one bit, counter = 1	#2=1
8	11010000			1 skipped bit	
8	11010000	AC#1:	01 ⇒ 00000001	No zeros, one value: skip 1 bit	
8	11010000				
8	11010000	AC#1:	00 ⇒ 00000000	Fill with zeros, counter=64, next array	#2=64
8, 9	11010000 00110010	DC#1:	00 ⇒ 00000000	Skip 0 bits, counter=1	#3=1
9	00110010	AC#1:	01 ⇒ 00000001	No zeros, one value: skip 1 bit	#3=2
9	00110010			1 skipped bit	
9	00110010	AC#1:	00 ⇒ 00000000	Fill with zeros, counter=64, next MCU	#3=64

Table 4: Decoding example: entropy-coded data representing a single MCU in a JPEG file.

third column (Huffman table lookup) refers to a translation from a Huffman code to a symbol, if that row contains such an action. First, the applicable Huffman table is named (referencing the names in the headers of Table 2 and Table 3) and next to it the code that was matched. The translated value (the symbol) appears as a result of the lookup in the appropriate Huffman table. If no match is found the validation fails.

The fourth column shows the interpretation, in case something else needs to be done except translate a Huffman code to a symbol. In all cases this refers to interpreting the resulting symbol, along with skipping bits in the input, which would normally be read as a value to be stored in the quantization array. Since a validator is not interested in decoding a JPEG file but simply recognizing it for validation purposes, the actual values are ignored and instead a counter is incremented to note that a value would have been read.

4.2 Detecting Errors

The example in Table 4 covers only 10 bytes but has many opportunities for validation errors to occur: 18 times a Huffman code is read from the input. Since none of the four generated Huffman tables use all possible values at the longest code size, every time a Huffman code is read it can potentially have a correct size but an incorrect value (e.g., for DC#0 in Table 2 this would be 11111, since the only defined code at length 5 is 11110). In addition, three quantization arrays are filled, which can all potentially lead to a size overflow if more than 64 values are defined for it in the input.

As a result, these 10 bytes constituting a single MCU have 21 validation opportunities. In order for this validation approach to be usable to detect a fragmentation point in the entropy-coded data of a JPEG file, only one single validation needs to fail in an entire block that typically has a size of 4096 bytes (or bigger).

4.3 Practical Analysis

To gain an idea of the potential usefulness in practice, the following sections investigate the success rates of both methods of validating entropy-coded data. They are separately discussed since they have different characteristics, mostly because they operate on different levels: the Huffman codes are directly present in the data while the quantization array values are discovered after a step of decoding. For both methods, we assume a block size of 4096 bytes since that is the minimum encountered in practice on all modern systems.

4.3.1 Invalid Huffman Codes. Each Huffman table lookup has a chance of failing, which in turn fails validation on the entire JPEG

file, which indicates that the current block is not part of a valid JPEG file. The success rate of this approach depends on both the maximum length of each Huffman table that is used, as well as the amount of lookups that are performed within a block. In the example in this section, the chance for a failed lookup for each Huffman table is $(1/2)^5$ (DC#0), $(1/2)^4$ (DC#1), $(1/2)^9$ (AC#0) and $(1/2)^8$ (AC#1). These values represent the chance of encountering an invalid bit at the end of a string of bits of the longest size in the respective Huffman table.

On average, in this example 1.8 lookups are performed per byte. Extrapolated to 4096 bytes, that would result in more than 7,000 lookups per block. Given that only one lookup needs to be erroneous to establish that a fragmentation point has passed, the odds of this method being successful are extremely high even though each individual chance is low. However, considering JPEG files with longer maximum Huffman code sizes, the odds change, but since every JPEG potentially has a different set of tables, it is difficult to estimate a realistic worst case.

4.3.2 Quantization Array Size Overflows. The success rate of a validation failure caused by a quantization array size overflow depends not on reading a single value in the input data, but relies on a longer sequence of AC-huffman table lookups that together cause the total size to overflow. A typical MCU consists of three quantization arrays. A small investigation of different JPEG images made with different cameras shows that there are between 490 and 640 arrays per block.

As with detecting invalid Huffman codes, only one quantization array size overflow is needed for validation to fail. In reference to the algorithm explained in section 3.1, the loop at step 3 must be repeated without 3.2.1 occurring or the loop finishing at exactly 64. The chance of this happening will be dependent on both the used quantization tables while encoding (i.e., the lower the compression due to quantization, the higher the chance of success), as well as the Huffman code used to decode the special 0x00 symbol. The longer that specific Huffman code is, the higher the chance of detecting a quantization array size overflow.

Given the high interdependence between many factors it seems unhelpful to speculate about practical odds. These factors include the sizes of the relevant Huffman tables, the size of the code in these tables that maps to the special `0x00` symbol and during a decode process how far along the construction of a quantization array is. Instead, validating quantization array sizes can better be considered an almost free extra chance to discover errors that incurs no performance penalty to the validator and may sometimes yield a validation error.

4.4 Limitations and Considerations

Even though the expected success rate of invalid Huffman code detection looks promising, this approach has some limitations that must be considered when applying it in practice.

Empty blocks (zero-blocks) will not necessarily be recognized as incorrect, due to one or more zeros likely mapping to reasonable symbols in each Huffman table. However, recognizing and removing (partially) empty blocks is typically a concern handled by the file carver that assembles candidate files (e.g., by excluding blocks with very low entropy as not being candidates for compressed data).

When the validator encounters other JPEG fragments that are encoded with the same Huffman tables, chances decrease of this validation technique being successful. However, the entropy-coded data is not automatically byte-aligned, so fragmentation can occur at any point in the entropy-coded data. Thus, fragmentation point detection is still possible in this case, but the odds decrease slightly in case the fragments end up being aligned.

Even if validation does succeed in an entire block that is not part of the original JPEG file, subsequent blocks can still fail to validate. The first block (or more) after the fragmentation point will then be missed by the file carver, but in this way a partial file may potentially still be recovered.

The described approach will work when the default Huffman tables are used, which are very popular especially with digital cameras. The Huffman tables as they are presented in the JPEG specification were not meant as a standard, but were presented as a good reference point. However, these tables became a de facto standard[4], because calculating custom Huffman tables for each picture comes with processing costs, with only limited gains with in compression efficiency. Since each default table in the JPEG spec has the same property as described in Section 3.1, both the default Huffman tables and many custom-generated Huffman tables will work with this approach.

5 RELATED WORK

Many different approaches to detecting fragmentation points in JPEG files have been attempted. Existing research can roughly be divided in three categories, focusing on different aspects of JPEG: content, file structure and decoding characteristics.

Content based approaches are based on fully decoding candidate JPEG files. Li et al. [5] describe an approach to look for visual abnormalities in the resulting output. The proposed validator looks for abrupt changes in the DC coefficient, and evaluates the distribution of the AC coefficients. Tang et al. [11] propose a statistical metric (Coherence of Euclidean Distance) that is applied to the decoded RGB-values of pixels. These resulting values are then used to evaluate if JPEG fragments belong together. Both content based approaches are susceptible to false positives when sudden changes in brightness or color in the actual picture are present.

Another content-based approach is described by Birmingham et al. [1], that uses an embedded thumbnail in the JPEG file to aid in reconstructing the actual JPEG file. The thumbnail information is scaled and used as a predictor for the full-size image, and is able to identify invalid JPEG data.

Validation categories can also be combined: Karresand and Shahmehri [4] use both the restart markers and their expected occurrence within a file in addition to using metrics in the expected maximum change in luminosity in order to try to reassemble JPEG files. Similar combinations of validations are done by Pal et al. [9], where syntactical tests (keywords and file head matching) are combined with statistical tests in the form of entropy analysis. With a matching-metric (tuned with a training-set of images), chances are calculated for the likeliness of two fragments belonging together.

Validation can also be pursued without visual expectations, and focus solely on decoding and JPEG specification characteristics.

[4] https://www.impulseadventure.com/photo/optimized-jpeg.html

However, Pal and Memon [8] have shown that a successful decode does not always imply that the file was reconstructed correctly. Nescar and Memon [10] propose a method to match JPEG fragments to JPEG headers, based on pattern recognition in the encoded data. Mohamad and Deris [7] focus on detecting fragmentation within the Define Huffman Table (DHT) segment of a JPEG file, which is practically of little value since files will almost never fragment in the area of the file where the DHT segments are defined.

Research not directly related to fragmentation point detection but with an emphasis on fragment reconstruction with a mathematical approach is done by Ying and Thing [13]. They approach the process of matching JPEG fragments as a combinatorial problem, proposing a weighted graph approach, where each file fragment is assigned to and represented by a node. The weighted edges in this graph represent the likelihood of a match between two fragments (nodes). A path-finding algorithm then identifies the correct order of the given JPEG fragments.

6 CONCLUSION

Even though file carving is a powerful data recovery technique used in many investigations in digital forensics, it would be even more useful if it could reliably recover fragmented files in especially the most relevant file format: JPEG. Unfortunately, JPEG files consist almost entirely of large blobs of highly compressed entropy-coded data, making it very difficult to construct a reliable validator to aid file carvers in recovering fragmented files.

In this paper we describe an approach that leverages two observations about many JPEG files in practice. First, the Huffman tables used to decode a large proportion of the entropy-coded data often do not use all possible code values at their longest code length, offering possibilities to detect errors when invalid codes are encountered. Second, after translating Huffman codes to symbols, the next step in decoding involves filling quantization arrays with exactly 64 values, offering another possibility to detect errors when an overflow is encountered.

This paper describes an algorithm to validate the entropy-coded data using these two observations and finds that the odds of finding fragmentation points in practice are quite high, especially with regard to invalid Huffman codes. It will work with the example Huffman tables provide by the JPEG standard that are used by many digital cameras, but also with many optimized Huffman tables generated by specialized applications.

The next step is to implement this approach to work with a file carver designed to recover fragmented files and report on practical findings.

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

REFERENCES

[1] Brandon Birmingham, Reuben A. Farrugia, and Mark Vella. 2017. Using thumbnail affinity for fragmentation point detection of JPEG files. In *17th International Conference on Smart Technologies (IEEE EUROCON)*. IEEE, 3–8.
[2] Simson L Garfinkel. 2007. Carving contiguous and fragmented files with fast object validation. *Digital Investigation* 4 (2007), 2–12.
[3] ITU/CCIT/JPEG. 1992. *Recommendation T.81: Digital Compression and Coding of Continuous-Tone Still Images - Requirements and Guidelines.* Technical Report. International Telecommunication Union.
[4] Martin Karresand and Nahid Shahmehri. 2008. Reassembly of fragmented JPEG images containing restart markers. In *2008 European Conference on Computer Network Defense*. IEEE, 25–32.
[5] Qiming Li, Bilgehan Sahin, Ee-Chien Chang, and Vrizlynn L. L. Thing. 2011. Content based JPEG fragmentation point detection. In *IEEE International Conference on Multimedia and Expo (ICME)*. IEEE, 1–6.
[6] Dutch T. Meyer and William J. Bolosky. 2012. A study of practical deduplication. *ACM Transactions on Storage* 7, 4 (2012), 14:1–14:20.
[7] Kamaruddin Malik Mohamad and Mustafa Mat Deris. 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, 173–180.
[8] Anandabrata Pal and Nasir Memon. 2009. The evolution of file carving. *IEEE Signal Processing Magazine* 26, 2 (2009), 59–71.
[9] Anandabrata Pal, Husrev T Sencar, and Nasir Memon. 2008. Detecting file fragmentation point using sequential hypothesis testing. *Digital Investigation* 5 (2008), S2–S13.
[10] Husrev T Sencar and Nasir Memon. 2009. Identification and recovery of JPEG files with missing fragments. *Digital Investigation* 6 (2009), S88–S98.
[11] Yanbin Tang, Junbin Fang, KP Chow, SM Yiu, Jun Xu, Bo Feng, Qiong Li, and Qi Han. 2016. Recovery of heavily fragmented JPEG files. *Digital Investigation* 18 (2016), S108–S117.
[12] Vincent van der Meer, Hugo Jonker, Guy Dols, Harm van Beek, Jeroen van den Bos, and Marko van Eekelen. 2019. File Fragmentation in the Wild: a Privacy-Friendly Approach. In *IEEE International Workshop on Information Forensics and Security (WIFS)*. IEEE, 1–6.
[13] Hwei-Ming Ying and Vrizlynn L. L. Thing. 2010. A Novel Inequality-Based Fragmented File Carving Technique. In *Third International ICST Conference, e-Forensics (LNICST, Vol. 56)*. Springer, 28–39.