Information Security

Carving fragmented JPEG images

Content-based file carving of non-contiguously fragmented JPEG images

Master Thesis

for the degree of

Diplom-Ingenieur

submitted by

Bernhard Schildendorfer, BSc.
is101510

at the
Department for Information Security at the University of Applied Science St. Poelten

Mentoring
Mentor: Dipl.-Ing. (FH) Mag. Rainer Poisel
Assistance: Titel Vorname Zuname

St. Poelten, August 4, 2012 _____ _____

(Signature of Author) (Signature of Mentor)

Declaration of honor

I affirm, that

- I have drafted this degree dissertation independently, have not used other than the stated sources and means and further, have not used any other illegitimate assistance.
- I have, neither domestically nor abroad, submitted this degree dissertation to an examiner for inquiry or in any other form as a test paper.
- This paper is identical with the paper, which has been assessed by the examiner.
- Hereby I grant St. Poelten University of Applied Sciences (Fachhochschule St. Poelten) the exclusive and spatially unrestricted right of use to this degree dissertation, for all kinds of uses, and retain the right to be referred to as author of this work.

St. Poelten, August 4, 2012

(Authors signature)

Acknowledgement

I want to thank my mentor Dipl.-Ing. (FH) Mag. Rainer Poisel for the support during writing this thesis. Without the preliminary research on the ”Multimedia File Carver”, this work would not be possible.

Many thanks to my parents who enabled my studies and supported me all the time. I also want to thank my girlfriend, who shared and perfectly understood my situation, by writing her diploma thesis too. She also motivated me whenever my algorithms didn't work as expected.

Finally, I thank my colleges for their support during my entire study. Group work, lessons and countless lab hours have been great with you!

Zusammenfassung

File carving spezialisiert sich auf die Wiederherstellung von gelöschten Dateien und stellt damit einen wichtigen Teil der IT Forensik dar. Diese Diplomarbeit zeigt unterschiedliche Carving-Techiken und beschreibt einen spezialisierten Ansatz, um gelöschte, mehrfach fragmentierte JPEG Bilddateien wiederherzustellen. Um dieses Ziel zu erreichen ist ein tiefgehendes Verständnis über die interne Funktionsweise des JPEG Dateityps ist notwendig. Sowohl der verwendete Kompressionsalgorithmus, als auch das JPEG Containerformat, wird im Rahmen dieser Diplomarbeit beschrieben. Nach der Evaluierung der forensischen Techniken und JPEG Spezifikationen, wird der implementierte JPEG File Carver erläutert. Als Grundlage für die Entwicklung wird der Open-Source Carver "Multimedia File Carver" verwendet, der um Algorithmen für JPEG file carving erweitert wird. Die beschriebenen Algorithmen ermöglichen es dem File Carver JPEG Dateiblöcke auf dem Speichermedium zu identifizieren, daraus Fragmente zu bilden und diese miteinander mit Bildverarbeitungsmethoden vergleichbar zu machen, um anschließend deren korrekte Reihenfolge ermitteln zu können. Abschließend wird der implementierte File Carver anhand eines zufällig generierten Dateisystem getestet. Um die Effektivität der umgesetzten Algorithmen zu erheben, werden die Resultate mit den Idealergebnisse verglichen und interpretiert.

Abstract

As a major task of IT forensics, file carving focuses on recovering deleted files. This thesis shows different carving techniques and describes a specialized approach, how to restore deleted JPEG image files, which are fragmented into multiple parts. To achieve this, deep knowledge of the internal mechanics of the JPEG file type is needed. Both the compression algorithm and the internal file type structure are explained. After evaluating the forensic techniques and file type specifications, the implemented JPEG file carver will be described. As a basis for development, the open-source carver ”multimedia file carver” is used, which will be extended with algorithms needed for JPEG file carving. New algorithms are required to identify JPEG data blocks on the disk, group them to file fragments and compare those fragments to each other, in order to reorder them correctly. In the end of this thesis, the implemented file carver is tested with a randomly generated disk image, that contains five different JPEG image files and many other files with different file type. The carving results will be analyzed, to evaluate the effectiveness of the implemented carving algorithms.

Contents

Introduction

1.1 Motivation

Whenever digital technology is involved in crime, digital forensics is used as major activity in law enforcement. The term digital forensics is derived from ”computer forensics” and is used in connection with all systems that can hold digital information [1, p. 1f]. As an example, if data gets unintelligible due to a technical error or an accidental deletion by the user, digital forensics could be used for data recovery. The importance of data recovery can easily be seen in the latest ”Data Loss Barometer” by KPMG. Between 2007 and 2010, 500 million people have been affected by data loss incidents, 11% of all incidents have been caused by human error or system error [2]. Using digital forensic, many inaccessible files, caused by data loss incidents, can be restored.

A common human error in electronic data processing is the erroneous deletion of a file. Although the recovery of a deleted file may seem as a difficult task for the user, this is the simplest case of file recovery by use of digital forensics on many file system. This is possible because most file systems avoid to delete files bit by bit due to performance reasons. Instead, the storage area of files get marked as available, which allows forensic programs to restore their state and therefore recover them [3, p. 60].

Recovering data becomes much more difficult if no file system information is present, which includes the fact that also the information on file size and address is absent. File Carving faces this problem by analyzing the whole data storage and locating files based on their distinct header and footer information. Afterwards, all data between the starting point and the end point is restored, regardless of the data in between. Although this makes data recovery, even without file system information, possible, the lack of data validation may cause the file to be recovered incorrectly. These recovery errors can be caused if files are not stored as a consecutive chunk of data but fragmented over different places on the data storage. Mohamad et al. [4, p. 80] described a carving method for JPEG images using image pattern matching, which is able to recover fragmented images if their fragments are stored in contiguous and linear order. All ”garbage” data in between, which does not belong to the actual file, is removed by matching image patterns.

Pal et al. [3, p. 67] presented a new approach called ”Smart Carving” which is capable of recovering randomly fragmented files. They describe a three step process which is able to efficiently

reassemble fragmented data files based on their content. Key elements of this carving process are algorithms to classify the clusters based on their content and an algorithm to reassemble the fragments in the correct order called ”Sequential Hypothesis Testing” (SHT).

1.2 Problem statement

File Carving is implemented by many different programs of which ”Foremost” [5] and ”Scalpel” [6, p. 1] are the most famous ones. In contrast to the previously described ”Smart Carving”, these tools mainly implement carving based on the header and footer information to reassemble files. As soon as the files are fragmented, the forensic results of these applications become unsatisfactory. Beside open-source file carvers, an efficient content-based Smart Carver like described by Pal and Memon is only available commercially [7]. A publicly available open-source application is desired which utilizes content based file carving to restore randomly fragmented files.

1.3 Research questions

As described in the problem definition, there are no file carvers publicly available which are able to recover fragmented JPEG images. Therefore, the goal of this diploma thesis is to describe and implement such a software. Due to its open source-nature, it can take great benefit of open-source software for file carving (e.g. ”Multimedia File Carver” [8]) and JPEG processing libraries. This will answer the following research question:

> *How can fragmented JPEG images be automatically recovered using File Carving?*

Since the recovery of fragmented JPEG images is more than just reordering file fragments, many different file carving algorithms need to be reviewed and taken into consideration before implementing the software. Speaking about reassembling file fragments, algorithms for file type detection of clusters and the identification of fragments are needed. Intensive research regarding file type classification of file fragments will be covered by the second research question.

> *How can existing algorithms for file fragment identification be enhanced to support reliable identification of JPEG file fragments?*

The identified algorithms for recovering fragmented JPEG files will be implemented using open-source software and will be publicly available. The resulting program will be based on the already existing ”Multimedia File Carver” [8] and extend its capabilities of recovering JPEG files.

1.4 Organization of the thesis

The thesis starts by evaluating related work in the field of file carving, concentrating on file carving approaches and algorithms for file type determination. To implement an JPEG file carver, deep understanding of the JPEG specification is needed, which is described in chapter three. Since the JPEG specification only describes the compression algorithm, related specifications like JFIF or Exif are also needed to be considered. Based on this research, JPEG specific file carving algorithms are implemented and specified in chapter four in detail. In addition to the algorithms, the general architecture of the file carver is shown, to give an understanding of how the file carver can be extended, in order to support further file types. In the end of this thesis, the implemented file carver is tested with a randomly generated disk image, that contains five different JPEG image files and many other files with different file type. The carving results will be analyzed, to evaluate the effectiveness of the implemented carving algorithms.

Related work

To introduce the topic of this thesis, this chapter will describe the current state of research regarding file carving, applicable algorithms and related projects facilitating those algorithms. File carving itself is a broad and difficult research area because the forensic recovery of deleted files is influenced by many different factors like the file types, storage media, file system or file fragmentation and therefore may be unreliable. This chapter describes approaches to deal with those influences and optimize the file carving algorithm in order to increase the number of correctly carved files to a maximum.

2.1 Types of file carving

In general, the term file carving describes the forensic technique of recovering arbitrary files without using file system information [3, p. 60]. Since this definition is very broad and the approaches for file carving are very specific and complex, different types and forms of file carving exist. The definitions of these types depend on the author, which carries the risk of confusing terms. Therefore, a definition of the most common file carving types will be given in this section, based on research work of Pal, Memon [3, p. 59] and Garfinkel [9, p. 2]. Table 2.1 defines common terms in the field of digital forensics, which will be used in this thesis too.

Garfinkel divides the field of file carving into two non-exclusive sections: *contiguous carving algorithms* and *fragmented carving algorithms* [9, p. 10]. The categorized algorithms differ in the ability to recover fragmented files. It needs to be mentioned that this classification takes single algorithms into account and not a complete file carving process. This is important because a file carver, which uses fragmented carving algorithms, will probably also use contiguous carving algorithms. For example, *bifragment gap carving*, described later in section 2.3.7, also uses elements of *Header/-Footer Carving*. The following file carving types are described by using the following term definition:

2.1.1 File structure based carvers

According to Pal and Memon [3, p. 63], the first generation of file carvers can be categorized as file structure based carvers, using common file specific features for the carving process. The simplest but efficient approach is to identify files by their distinct header information. The whole disk gets scanned

Term	Definition
Cluster	The size of the smallest data unit that can be written to disk.
Header	The cluster which contains the starting point of a file.
Footer	The cluster which contains the ending point of a file.
Fragment	One or a sequence of clusters of a file that are not connected to other clusters of the same file. A fragment belongs only to one file. The distance between fragments is unknown. Fragments of unallocated files are can be overwritten on therefore not in place.
Base-Fragment	The first fragment of a file which contains the header cluster.
Fragmentation Point	The last cluster of a file fragment before fragmentation occurs. If a file has *n* fragments, *n-1* fragmentation points exist.
Fragmentation Area	A set of clusters which contain the fragmentation point of a specific file.

Table 2.1: File carving term definition based on Pal et al. [10, p. 4]

for ”magic numbers”, which are stored in the first bytes of most files to specify their file type. An example of this header information is the JPEG file type, which always starts with two magic bytes `0xFFD8` and ends with `0xFFD8`, forming the footer. Not all file types contain magic numbers in their footer, like Windows Bitmap, which instead contains the size of the file in its footer cluster.

Using the header information, a file carver can scan the whole data disk to identify all starting points of a specified file type. For a chosen header, the carver assumes all clusters up to the next footer to belong to the header and restores this sequence of clusters as a file. Header/Footer carving is a contiguous carving algorithm because it doesn't take fragmentation into account. Even if a file is fragmented, all data between the header and the footer are restored, which may add wrong clusters to the file or even worse, associate footer and header of different files to each other [3, p. 63].

Another related carving algorithm, is the *Header/embedded length carving*. Many file types have their file size stored in the header, which allows structural conclusions for file carving. This allows a file carver to read as many clusters beginning with the header as the size specifies it, to restore files even without footer information [9, p. 10].

2.1.2 Semantic carving

Since many text based files don't have magic numbers or other structural features which can be used for a proper reassembly, a different carving approach is needed. Semantic carving decides reassembly decisions based on linguistic features of different files or file fragments. Of course, the reassembly is highly dependent on the used language excluding all file types which don't use any natural or artificial language (e.g. source code) suitable for a semantic analysis. For the analysis of the data, various different approaches can be facilitated in order to gain information on the used language or the order

of different fragments. Linguistic features like the frequency of different letters or the word order reveal the used language and may also allow an non-contiguous file carver to correctly reassembly different fragments. [11, p. 10]

2.1.3 Carving with validation

According to a research of Garfinkel [9, p. 6], many carving decisions can be solved using object validation. It is a decision problem in which a validator decides if a set of bytes form a file or file fragment. The validator ensures that the resulting file is syntactically correct and therefore allows someone to open the file properly. The decision may be based on specific file structure or file system information. The easiest way of validation was previously mentioned with the ”file structure based carvers”, which can be seen as a sub form of object validation. Using this carving technique, file structure, like the header data or footer data, is used to validate the correct file type of a byte sequence. Since this approach can’t validate the data between the header and footer, it is very likely that faulty bytes occur, corrupting the carved file.

More specialized validation algorithms facilitate the internal container structure of files in order to improve the decision confidence. Many file types organize their data by using a container structure. For example, JPEG consist of containers of variable length which are used to store meta information like Exif data, Huffman tables or image data. These containers are well defined by the JPEG standard and are supposed to occur in a certain order within the JPEG file. If an object validator is capable of the JPEG standard, it can decide whether the analyzed data is compliant with the standard, allowing a reliable way to determine the file type or the correct reassembly of a file.

Garfinkel [9, p. 8] also mentioned the usage of decompression as a proper validator of compressed file types. Before being able to apply the decompression, the container structure must be validated first in order to extract the compressed data and to retrieve the probably needed meta data. When JPEG is again taken as example, the compressed image data can only be decompressed if certain compression parameters, like the Huffman table or quantization table, are extracted from the meta data.

2.1.4 Smart carving

Pal and Memon [3, p. 67] presented a new structured approach for carving files, called ”Smart-Carver”. They describe it as ”design of a file carver that is not limited to recovering files containing two fragments”, [3, p. 67] like Garfinkel proofed the feasibility of a successful recovery with his ”Bifragment Gap Carver” [9, p. 10]. They analyzed the typical behaviors of disk fragmentation and used the results for reliably reassembly randomly fragmented files. Figure 2.1 shows the architectural design of the proposed file carver, structuring its algorithms into three independent components. The first phase is called preprocessing, which is responsible for preparing the storage media for the actual forensic analysis. This includes the decryption and decompression of the file system if needed, or

gathering important meta information on the storage media. If the data clusters are accessible, they are, based on their file type, classified in the collating phase. In the final step, the classified clusters are compared to each other to merge them and reconstruct the files. The wording and structure of the SmartCarver will be used in the design and development of the JPEG file carver shown in this thesis, because it allows a structured approach for implementing a file carver. Further it allows a modular expandability of various carving algorithms.

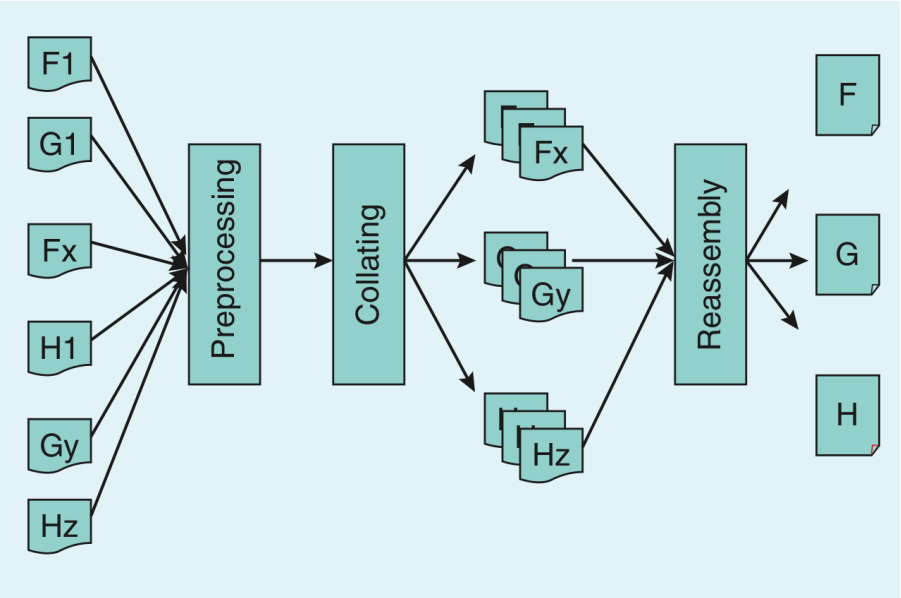


Figure 2.1: Overview of the SmartCarver, showing its three main components: preprocessing, collating and reassembly [3, p. 67]

Preprocessing The preprocessing step prepares the raw data for further forensic analysis. Therefore it is important to decrypt or decompress the data in order to supply it to the next layers in a processable form. An example for data encryption is the feature BitLocker of Windows Vista. It encrypts the data with a key, stored on a Trusted Platform Hardware Module (TPM). The preprocessing step would either need to break the encryption or, more efficiently, let the subsystem decrypt the data storage. [3, p. 67]

Another important feature of the preprocessing step is to remove all clusters which are referenced by the file system, if applicable. Since less data clusters are needed to be analyzed, the computational effort is drastically reduced in the reassembly phase. Of course, this step needs to be skipped if either no file system meta data is in place, or it is assumed that it is tampered or corrupt. [3, p. 67f]

Collating During the collation phase, unallocated data clusters are analyzed for their file type. This file type determination utilizes specialized algorithms (see chapter 2.2) because the raw data my not contain meta information which let it easily associate with a specific file type [3, p. 68f]. To explain this in more detail, lets assume a stored JPEG file, which gets deleted by marking the associated clusters as available in the file system. As described in chapter 2.1.1, the file type can easily be identified by searching the header and footer information of a file, assuming all other clusters in between to belong to the specified file type. Since the SmartCarver shall be able to reassemble fragmented files, this approach is not applicable for all data clusters. Therefore, clusters need to be analyzed independently for their file type. To get back to the JPEG example, each cluster needs to be analyzed for special features of the file type like key words, control sequences, byte frequencies or other statistical data. The reliability of file type classification is essential for an effective reassembly.

Reassembly The final step of the smart carving process has the task to identify the fragmentation point of each unrecovered file in identify the starting point of the correct next file fragment. This is done until all files are reassembled or determined to be unrecoverable. It shall be noticed that not the raw clusters are reassembled but the identified file fragments which may consist of multiple clusters. [3, p. 69ff]

The algorithm for identifying the correct order of the available file fragments is called ”reassembly algorithm”. Different approaches are explained in more detail in chapter 2.3. Basically they can be seen as a path problem, with the problem of assigning each vertex (fragment) another one in a way the edge (probability of correctness) between those vertices or of the whole graph is optimal.

2.2 Collation algorithms

Collation algorithms are used to determine the file type of a data cluster. Since a data cluster is usually a very small unit of information, the classification decision is error prone. The quality of a collation algorithm is defined by its ability to correctly identify the file types of clusters while minimizing the amounts of false-positives.

2.2.1 Signature based

Signature based collation algorithms are the most straight forward techniques for file type identification. They rely on static patterns hold by the file. The well known UNIX ”file” command makes use of the libmagic library which provides a signature based file type identification. It searches for signatures in the targeting file to identify it. Many file types start with an unique byte pattern called ”magic numbers”, like `0xFFD8` for a JPEG image, which is used for a reliable identification [12, p. 32].

A disadvantage of this technique is that the beginning of the file is needed to identify the file type. Signature based algorithms can't be used for file type identification of file fragments since the magic numbers only exist in the starting fragment. This approach can only be used to determine the beginning of files. The number of recovered starting fragments also determines the number of parallel execution paths for the later reassembly [13, p. 140].

2.2.2 Feature based

In contrast to signature based algorithms, feature based algorithms are specialized type-x recognition approaches, which look for file type specific features. Roussev [14, p. 12] and Veenman [15, p. 397] conclude that feature based algorithms are needed for a reliable file type determination. "It seems that the two-class (or type-x) recognition models are more useful, since the false positive rate can be adjusted per file type. The multi-class (type-all) recognition models have, however, the advantage that they clearly show which file types are confused with which ones. Although, the cause for these confusions is not fully clear, focusing on them helps in defining more distinctive features." [15, p. 397]

As an example, lets have a short look at an example for feature based identification of JPEG fragments. "The header has a simple record structure where the beginning of each record is announced by the presence of a marker, a 16-bit number in the `0xFFC0` to `0xFFFE` range, which is followed by a 16-bit number describing the length of the record." [14, p. 9] Because the byte after `0xFF` are used as control sequences, the byte `0xFF` needs to be escaped if it exists in the valid image date. Therefore, a zero byte is stuffed after `0xFF` which results in `0xFF00` [12, p. 64]. Roussev found out "that the average distance between two occurrences of `0xFF00` in a jpeg file was 191 bytes" [14, p. 10]. This feature can be used to reliably classify JPEG file fragments.

Beside JPEG, feature based approaches are also feasible for HTML by looking for "<html>" or "href=" or other distinct character strings. Important is, that the features are highly dependent on the file types and should be unique [3, p. 68].

2.2.3 Normalized compression distance

"NCD is based on the idea that by using a compression algorithm on data vectors (in whatever shape or form these may come) both individually and concatenated, we will receive a measure of how distant they are. The better the combination of the two vectors compress, compared to how the individual vectors compress on their own (normalised to remove differences in length between the set of all vectors), the more similar they are." [16, p. 25] Equation 2.1 shows the normalized compression distance in a more formal way, in which C is the compression function, x and y are two vectors of which their distance is determined. The lower the output of this function is, the higher is the probability that x and y are fragments of the same file type. For this approach, both x and y don't have to be fragments of the same file, y is assumed to be a fragment of a reference file.

$$NCD(x,y) = \frac{C(x,y) - min(C(x), C(y))}{max(C(X), C(y))} \tag{2.1}$$

Axelsson also shows that this type-all algorithm can be greatly used to tell compressed and non-compressed file types apart and identify many non-compressed file types. The distinction between different compressed file types like jpeg, zip or png is not possible due to an average accuracy per class of less than 10% [16, p. 28ff].

2.2.4 Statistical approaches

All algorithms that use statistical methods for file type determination can be seen as type-all algorithms and are summarized in this chapter.

Forensic relative strength scoring system is a scoring system proposed by Shannon [17, p. 1] to determine the file type of a data set. It consists of ASCII proportionality and entropy scoring. ASCII proportionality is a very simple approach to tell textual data and binary data apart. It counts all readable ASCII characters and divides it by the sum of bytes in the file. The resulting percent value shows the ASCII portion of a file, whereas 100% would be a file containing only readable ASCII characters [17, p. 3]. The second method, entropy scoring, measures the density of information in a file. In contrast to NCD, this algorithm doesn't compare two datasets to each other but tries to guess the file type based on the entropy value. "To summarize one of Shannon's concepts, Entropy is a measure of the information density or compression state of a given unit of data. The more a given unit can be compressed, the lower the Entropy value" [17, p. 4]. Since different file types have specific entropy values, the outcome of this approach can be used to categorize the file.

File fingerprint is a more complex statistical approach presented by McDaniel and Heydari [18, p. 3f], using byte frequency distribution (BFD) and byte frequency cross-correlation (BFC). They assume that every file type has a generic fingerprint based on their statistical occurrence of bytes. Therefore, the BFD algorithm counts the occurrences of all bytes from 0 to 255, resulting a histogram. This histogram, also called "byte frequency fingerprint", can be compared to an averaged histogram trained by a test set of the specific file type. If the variance between two histograms is below a defined threshold, the file type is determined. The second technique, byte frequency cross-correlation, is used to increase the reliability of a correct file type determination. It takes the correlation between frequently occurring bytes into account. It is assumed that many file types contain certain patterns or strings, like the HTML opening tag "<" and closing tag ">" or the keyword "href=". This technique is a kind of feature based approach, which helps the BFD algorithm to achieve better results. They showed in their research that their algorithms achieve a low average accuracy rate of 30% to 45%.

Oscar Method is a specialized approach based on BFD shown by Karresand and Shahmehri [13, p. 147]. To increase the accuracy rate of the BFD algorithm, they introduced a rate of change (RoC), which takes the ordering of two consecutive bytes into account. Like in BFD, the RoC between all consecutive bytes in a file can be seen as a histogram which gives an unique fingerprint on the ordering of bytes. Afterwards, it can be compared to a reference fingerprint of a specific file type. According to Karresand and Shahmehri, this algorithm performs, depending of the file type, with an accuracy up to 86%.

Visual classification of data objects is presented by Conti et al. [19, p. 1]. The stored bytes are presented as grayscale graphical depictions, which helps to distinguish structurally different regions within a data file. This "facilitates a wide range of analytic tasks such as fragment classification, file type identification, location of regions of interest, and other tasks that require an understanding of the 'primitive' data types the objects contain." [19, p. 1] Within their paper, they discussed the visualization of a wide range of media types which helps to understand their structure to make a file type classification based on visual patterns possible.

Linear discriminant analysis is used by Calhoun and Coles [20, p. 15] to classify different files into groups of file types. "There is an initial training phase in which data from individuals belonging to known groups are used to develop a classification model. The classification model is a set of linear functions, one for each group. Data from individuals whose group membership is unknown can then be entered into the classification model. The model predicts the group to which the individual belongs according to the function that returns the highest value." [20, p. 15] The linear discriminant prediction is based on combination of statistics like the frequency of ASCII codes, entropy, standard deviation and correlation between adjacent bytes. As new approach, the prediction based on the longest common substring is used. It "is based on the idea that two files of the same type will probably have longer substrings in common than would files of different types." [20, p. 15] The results show that the accuracy of the linear discriminant method is strongly dependent on the used combinations of statistics as well as on the compared file types. Although the comparison between JPEG and PDF files show a very high success rate, JPEG was not compared to another compressed file type like ZIP which would be a more challenging task.

Support Vector Machine SVMs (Support Vector Machines) are an universal method for data classification using mathematical approaches. It operates in two phases. In the first one, the training phase, training data is processed by the SVM for which the classification is already known. Each value of the test set is represented as a vector in a vector space. The goal of the SVM is to calculate a model which separates the two sets best of each other with so the distance between them becomes the maximum. This can be seen in Figure 2.2 where a training set of two classes of test vectors get separated by a linear discriminant. If the width of the discriminant is a maximum, it is optimal (b).

Although the discriminant of (a) is broader, it is only sub-optimal because not all vectors are classified correctly.

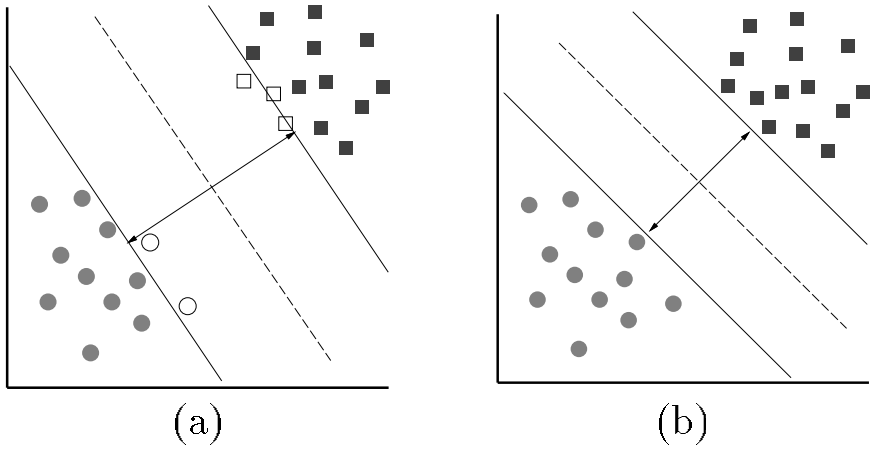


Figure 2.2: (a) A sub-optimal solution where not all vectors are classified correctly. (b) The width of the linear discriminator is optimal for this test set.[21, p. 11]

Since it may happen that vectors can't be linear separated, which is necessary to formulate a SVM, the kernel trick must be used. Using a so called kernel function, the training vectors are mapped into a higher (maybe infinite) dimensional space where they can be linear separated using a hyperplane. The hyperplane can be transformed back into the two dimensional vector space and used as discriminator for classification. [22, p. 4]

2.3 Reassembly algorithms

Based on the classified file fragments, algorithms are needed to identify their correct order. Beside the chosen algorithm, the resulting order is based on the candidate weights between the fragments, which is used to compare fragments. If a set of fragments are seen as vertices in a graph, whereas the weights represent the edges in between, the reassembly algorithm needs to find the optimal path for each file, based on the calculated weights. Based on a definition by Pal et al. [23, p. 388ff], a reassembly algorithm can be specified by the following features:

1. **Path problem**: Since all fragments belong exactly to one file, causing every vertex to be used uniquely, the reassembly algorithm usually results in a vertex disjoint path problem. Nevertheless, algorithms can also be non-disjoint, allowing vertices to be used multiple by different reassembly paths.

2. **Execution mode:** files can be reassembled serially or in parallel.

3. **Heuristic:** two heuristic modes are currently defined for reassembly algorithms, Greedy and Enhanced Greedy. They define the decision process for assigning a fragment to a file. Greedy Heuristics start with the header of a file and add it to a reconstruction path *P*. This header fragment is chosen as current fragment *s* and compared to the other available fragments. Based on the weights calculation, the best match *t* is chosen and added to the reconstruction path. *t* is now set to be the new current fragment *s*. The process is repeated until the file is recovered or no more fragments are available. Enhanced Greedy algorithms additionally verify if the next fragment *t* would have a better predecessor *b* than *s*. This greatly reduces the propagation of errors [24, p. 21f].

Based on these features, Pal et al. presented eight algorithms, of which the Unique Path algorithms are a very important subset. They are based on the k-vertex disjoint path problem. As mentioned before, this causes every vertex to be used only once in the whole reassembly process. Therefore, UP algorithms create an unique vertex path for every file in the graph.

2.3.1 Greedy Sequential Unique Path (Greedy SUP)

"Greedy sequential unique path is a sequential algorithm using the greedy heuristic. When the algorithm assigns a fragment to an image reconstruction, the fragment will be unavailable for selection in the reconstruction of any other images," [23, p. 289] creating vertex disjoint paths. The problem of this algorithm is that the reassembly is highly dependent on the order of images being processed. This is caused by the characteristic of disjoint paths that mistakes in building the reassembly paths will propagate. In other words, if an fragment is wrongly used in the reassembly path *P1*, it is unavailable in the reassembly path *P2* where it would belong. The missing fragment could cause *P2* in turn to use another wrong fragment spread, allowing the mistake to spread across multiple reassembly paths.

$$\sum_{i=1}^{F} i \tag{2.2}$$

To calculate the complexity of this algorithm, we need to Figure out how many comparisons are necessary to reassemble one file. Equation 2.2 calculate the amount of comparisons for a file, where F are the remaining file fragments. Of course, each reassembled file reduces the number of available fragments. Based on this equation, this algorithm has an average complexity of $O(n \sum n)$, which can be generalized to $O(n^2)$.

2.3.2 Greedy Non-Unique Path (Greedy NUP)

”Greedy non-unique path is also a sequential algorithm using the greedy heuristic. Since this is a NUP algorithm any fragment, other than a header fragment, that was chosen in the reconstruction of an image will be available for selection in the reassembly of another image. This prevents errors from propagating but as mentioned earlier, does not necessarily lead to disjoint paths.” [23, p. 289] In contrast to SUP, the multiple usage of fragments will prevent reassembly errors to propagate. The complexity of this algorithm with $O(n^2)$ is the same like for Greedy SUP.

2.3.3 Greedy Parallel Unique Path (Greedy PUP)

Greedy PUP builds up the same reconstruction paths like Greedy SUP, but reconstructs the files not sequentially but parallel, reducing the risk of a propagation of errors. Memon and Pal describe the algorithm as a variation of Dijkstra’s single source shortest path algorithm [23, p. 389]. Like Greedy SUP, reassembly paths P_i are created for each file i to reconstruct. The header fragments h_i are added as the first element to the according reconstruction paths. In the next step, the last fragments of all reconstruction paths (in the first run the header fragments) are compared to the remaining file fragments. The best match of all of these comparisons is chosen and added to the according reassembly path. Figure 2.3 illustrates this algorithm. (a) elects fragment six to be the overall best match and gets added to header two (b). This process gets repeated until no more fragment are available or all files are reassembled. ”The problem with Greedy PUP is that the best fragments being matched with the current set of fragments may be better matches for fragments that have not been processed as yet, thus leading to error propagation again.” [23, p. 390] The complexity of this algorithm is again $O(n^2)$.

2.3.4 Greedy Shortest Path First Unique Path (Greedy SPF UP)

Shortest Path First (SPF) makes reassembly conclusions based on the average cost of a complete reassembly path as instead of single fragment comparison. By combining the benefits of Non-unique Path (NUP) and Parallel Unique Path (PUP) to completely prevent the propagation of reassembly errors while maximizing the overall correctness of all reassembled files. The algorithm first recovers all files using the NUP algorithm, letting al chosen fragments available for further usage. At this point, it is assumed that the file with the best average cost is the best reassembly. The average cost is calculated for each file by dividing the sum of costs between all fragments of a reassembly with the amount of fragments. The file with the lowest average path costs gets recovered and all assigned fragments are removed from the list, making them unavailable for the other recovery paths. Experiments show that the SPF algorithm provides the best accuracy with 88% of files being reconstructed, whereas the PUP algorithm scores 83% [3, p. 66]. ”However, the PUP algorithm was substantially faster and scaled better than the SPF algorithm.” [3, p. 66]

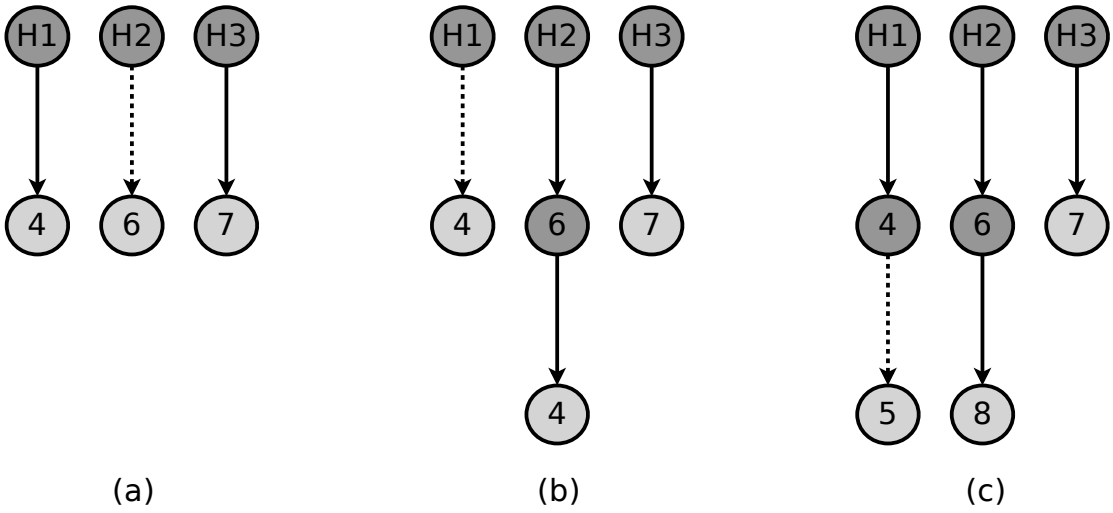


Figure 2.3: An example for Greedy Parallel Unique Path (Greedy PUP) [23, p. 390]

2.3.5 Enhanced greedy reassembly algorithms

”The greatest problem that the greedy heuristic has is that it chooses the best available matching fragment for the current fragment without attempting to take into account the possibility that fragment may be an even better match for another fragment that has not been processed as yet.” [23, p. 390] To avoid this problem, the enhanced greedy algorithms add a step before connecting to best match fragments together. The algorithm determines ”if the best match for a fragment may be an even better match for another fragment. If this is the case then the next best match is chosen, and the process is repeated until a fragment is found that is not a better match for any other fragment.” [23, p. 390] To illustrate this algorithm, s is again the current fragment and t its best match. Greedy UP would assign t to s and remove it from the list of available fragments. Enhanced greedy algorithms determine the best predecessor b for t in assumption this may not be s. t is only taken as s' best match if $s = b$ [23, p. 390]. Although it is not explicitly discussed by Memon and Pal, the PUP algorithm seems to fulfill this requirement already, although it is not explicitly described to be enhanced greedy. Because all available fragments (t's) are tested for b of all reassembly paths, it is assured that the weighting relationship is commutative.

2.3.6 Sequential Hypothesis-Testing PUP (SHT-PUP)

SHT-PUP is a modification of the PUP algorithm, testing not only whether two clusters belong to each other but also taking clusters after the comparative cluster into account. This greatly reduces

the amount of comparisons and increases the reliability of the reassembly. To explain this algorithm, assume the *i*-th current cluster $b_s i$ for the reassembly path P_i, which gets compared to the best cluster $b_t i$. Until this point, SHT-PUP and PUP are the same. SHT-PUP now also analyses clusters directly immediate after $b_t i$, starting with the data clusters b_1 to b_n, testing for two hypothesis whether the sequence of clusters belong to the fragment (H_0) or not(H_1). The algorithm tests for a longer cluster sequence as long the first hypothesis (H_0) is correct, meaning that the cluster sequence belongs to a fragment. The test stops if H_1 is met, indicating the fragmentation point. If the evaluated data do not yield into a decision, the test continues until one of the hypothesis can be confirmed.

The hypotheses are decided using a "likelihood ratio of observing sequence W under the two hypotheses, which is expressed as the ratio of the conditional distributions of observed weights under H_0 and H_1 as" shown in Equation 2.3 [3, p. 70].

$$\Delta(W) \quad = \quad \frac{Pr(W|H_0)}{Pr(W|H_1)} \tag{2.3}$$

The outcome of this likelihood ratio is then compared to certain thresholds (Equation 2.4) in order to decide which hypothesis is met. If this ratio is larger than τ^+, we assume that hypothesis H_1 is true. "If, on the other hand, test statistic is smaller than τ^-, hypothesis H_0 is true and all the fragments are merged and the test continues from the next sequential cluster. Finally, if neither case is true, testing continues until one of the thresholds is exceeded or end-of-file indicator is reached." [3, p. 70f]

$$Test \quad result \quad = \quad \begin{cases} H1, & \Delta(W) \quad > \quad \tau^+ \\ H2, & \Delta(W) \quad < \quad \tau^- \\ inconclusive, & \tau^- < \Delta(W) \quad < \quad \tau^+ \end{cases} \tag{2.4}$$

2.3.7 Bifragment Gap Carving

If a carving algorithm has the ability to carve a file of at least two fragments, it is called fragmented carving algorithm. Based on the fact that many files are not fragmented in more than two files, Garfinkel [9, p. 10] proposes bifragment gap carving, a fragmented carving algorithm, specialized on exactly two fragments per file. The algorithm assumes, that two fragments of a file are usually not separated by a huge gap. This idea can be seen in Figure 2.4, which shows two fragments *f1* and *f2*, ranging from sectors *s1* to *e1* respectively *s2* to *e2*, which are both separated by a gap of size g. Since the size of both fragments are not decided yet, the gap size needs to be determined. Therefore, all possible gap sizes between the two fragments are tried until the validation of the carved byte sequence works or g exceeds its maximum of *e2* - *s1*. Garfinkel describes the algorithm with a complexity of "$O(n^4)$ for finding all bifragmented objects of a particular type in a target, since every sector must be examined to determine if it is a header or not, and since any header might be paired with any footer." [9, p. 10]

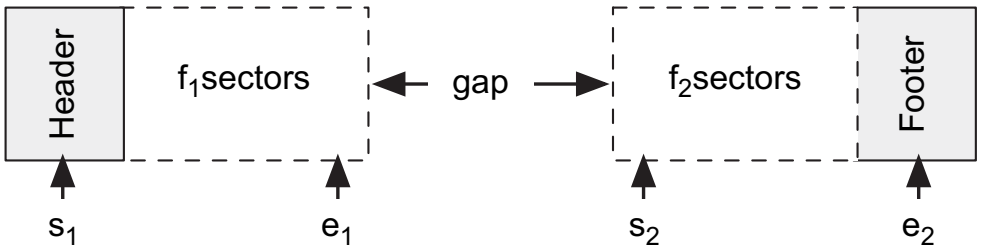


Figure 2.4: ”In Bifragment Gap Carving the sectors s1 and e2 are known; the carver must find e1 and s2.” [9, p. 10]

2.4 File fragmentation

To understand how to recover fragmented files it is essential to find the reasons out why and how those files are fragmented by the file system. File fragmentation is a necessary feature of file systems which allows the storage of files for which not enough consecutive clusters are available. After the first storage, fragmentation can still happen by performing write operations to the file, which dynamically changes its size, forcing the file system to use new clusters at a completely different location of the disk. Although files can be spread over different clusters, the file system can access them easily by maintaining a list of associated clusters for each file. Since this list can be deleted by the user or by a defect of the disk, the utilization of specialized file carver is necessary, which can restore files by reassembling their fragments.

As an example for such a file fragmentation, Figure 2.5 shows how files are logically stored using the FAT file system. Two files are stored, `Recovery.txt` and `Hello.txt`, which are both spread over multiple clusters. Both the directory entry and the FAT cluster entry are stored at the beginning of the file system and are used to locate the stored files on a cluster basis. The directory entry stores the file names and their starting cluster. The last is used to identify the other clusters for the given file, which are stored in a single linked list. The cluster numbers stored in that list are then accessed in the disk data area which holds the actual file data. If the files are deleted, simply the FAT cluster entries are marked as deleted and their directory entry gets tagged as available, still referring to the starting cluster. Although this allows a quick identification of the starting cluster of a file, the clusters of file fragments are hard to find [3, p. 61].

Garfinkel [9, p. 4] shows that file fragmentation is relatively rare on today’s file systems because the increasing storage size of disks. However, he still underlines the importance of file fragmentation capability of modern file carvers, since highly fragmented files are mainly of interest in forensic investigations. He shows three reasons why files get fragmented today, although modern file systems try to prevent this measure:

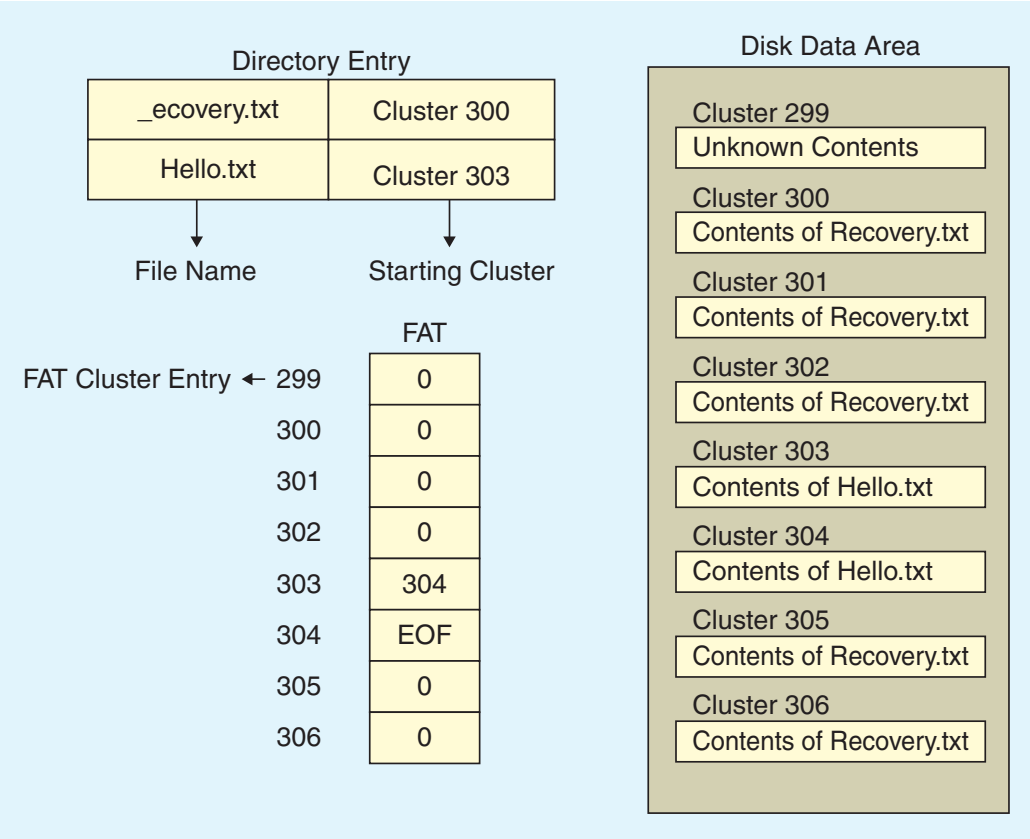


Figure 2.5: Overview of the logical storage of files using the FAT file system [3, p. 61]

1. There are not enough consecutive clusters available on the media to store the file without fragmentation. This may be the case if the media is heavily utilized, leaving many small sized available data regions on the disk.

2. New data is appended to an existing file, which has no more free clusters available at the end. The file system may now relocate the file to prevent file fragmentation. If it is a big file, the file system may also fragment the file for performance reasons, writing the new data to another location.

3. ”The file system itself may not support writing files of a certain size in a contiguous manner. For example, the Unix File System will fragment files that are long or have bytes at the end of the file that will not fit into an even number of sectors. Not surprisingly, we found that files on UFS volumes were far more likely to be fragmented than those on FAT or NTFS volumes.” [9, p. 4]

Kloet shows [25, p. 8f] different kinds of fragmentation scenarios which need to be concerned during the reassembly phase of a file carver. Beside of a non-fragmented file, three fragmentation types exist, which can be classified based on the arrangement of their fragments:

1. **Linearly fragmented files** are fragmented into at least two fragments which are arranged in the correct order on the media. This means that if the media is sequentially iterated, all fragments are read in the order they are intended to be reassembled, beginning with the header fragment and ending with the footer fragment.

2. **Non-linearly fragmented files** consist at least of two fragments where one or more fragments are not stored in the correct order. This is the case if the footer fragment is stored in clusters before the header fragment. A sequentially reassembly of the file fragments would result in an inconsistent file.

3. **Partial files** have at least two file fragments of which not all are in place, regardless of their order. Those files can't be reassembled and should be identified in order to exclude them from any file carving process.

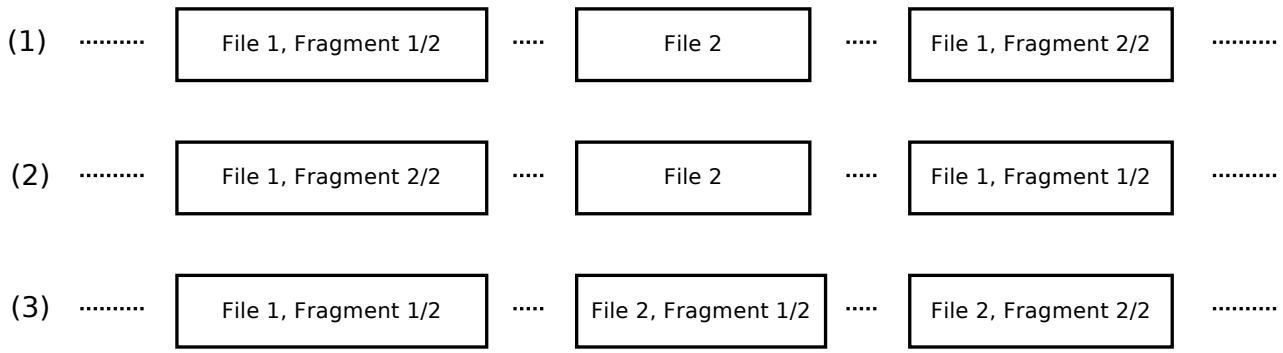


Figure 2.6: (1) linearly fragmented file (2) non-linearly fragmented file (3) partial file [25, p. 48ff]

2.5 Related projects

Foremost was originally developed by the United States Air Force Office of Special Investigations and later improved by Mikus [26, p. 59] in the course of his master's thesis. The tool allows file carving "based on their headers, footers, and internal data structures." [5]

Scalpel is an open-source file carver based on Foremost, which performance was improved by Richard III and Roussev [6, p. 1]. Although it is significantly more efficient, no considerable improvements were made in the used carving algorithms.

Adroit Photo Recovery / Forensics utilizes the SmartCarving algorithm invented by Pal and Memon [3, p. 67], allowing it to reassemble fragmented JPEG, PNG, BMP and GIF files with support for various camera raw data formats. The application also supports all versions of FAT, NTFS, HFS, HFS+ and corrupted file systems in general. The license costs for adroit photo forensics are $999 with an annual maintenance plan of $300. [7]

DataLifter allows the signature based recovery of files, supporting multi threading CPUs. It is a commercial forensic tool that can be bought for $155.00. [27]

Encase Forensic is a forensic toolkit sold by Guidance Software. Its advanced analysis feature allows the recovery of deleted files by parsing event logs or file signature analysis in unallocated disk space. [28]

Forensic ToolKit (FTK) is developed by AccessData and sold for professional use. Like Encase, only basic file carving operations are available. [29]

NFI Defraser allows, based on its documentation, the detection of ”full and partial multimedia files in data streams.” [30]

PhotoRec is an open source multi-platform application for recovery of image file. It supports many operating systems (DOS, Windows NT4/2000/XP/2003/Vista/2008/7, Linux, FreeBSD, NetBD, OpenBSD, Sun Solaris, Mac OS X) as well as file systems (FAT, NTFS, EXT2, EXT3, HFS+). The data recovery is done by using a signature based approach, working only for unfragmented files. [31]

Recover My Files is able to recover more than 200 file types, based on the official web page, including image files. No detailed file carving techniques are described on the product page. It can be purchased for €69,95. [32]

Revit is an open source proof of concept file carver, developed for the DFRWS 2006 challenge. It supports more versatile techniques than header and footer carving. It uses deep carving, which is a kind of file structure based carving approach, analyzing for very deep file specific structures [33, p. 11]. The alpha version of 2007 is no longer developed further.

Description of the JPEG standard

This chapter describes the JPEG image compression algorithm as well as the interchange formats which are needed for a proper exchange of JPEG images across multiple decoding entities like personal computers. A deep understanding of the JPEG data structure is needed to design an effective file carving algorithm for fragmented JPEG files, where both the compressed image data and the surrounding meta data needs to be analyzed and reassembled. Because of the very versatile file type specification, the standard will be explained as abstract as possible but as specific as needed to derive sufficient file carving algorithms.

Basically, the JPEG standard consists of three core elements. The encoder generates compressed image data by applying a specified set of procedures and certain table specifications to the digital source image data [12, p. 13]. The inverse operation is done by the decoder which outputs the digital reconstructed data. Both the compression algorithm, the encoder and the different modes of operations are described in the sections 3.1 and 3.2. At the end of the chapter, the interchange format is described which is required by the JPEG standard and needed for exchange between application environments [12, p. 13].

3.1 The compression algorithm

Regarding to the JPEG standard, JPEG compression can either be done lossy or lossless. Whereas the lossy compression makes use of a discrete cosine transform (DCT) to remove certain frequencies in the picture, lossless compression skips this step and does the compression only with entropy coding. In contrast to quantized DCT, entropy coding can be reversed without loss of information.

Since the lossy compression also contains the lossless compression in respect to the algorithm steps, it will be described in more detail. As you can see in Figure 3.1, the DCT-based coding consists of five steps to result the compressed image data:

1. The samples of the source image data, which are defined as elements in the two-dimensional array which comprise components, are grouped into 8x8 blocks [12, p. 6].

2. Afterwards, each block is transformed by a forward DCT (FDCT) into 64 values called DCT coefficients of which one is the DC coefficient and the other 63 are the AC coefficients. If the

DCT coefficients are also imagined as a 8x8 matrix, they are described as $S_{v,u}$ where *v* states for the x-axis and *u* for the y-axis. $S_{0,0}$ is the DC coefficient [12, p. 27].

3. In the next step, the DCT coefficient quantization is used to selectively reduce the information stored in the DCT coefficients. "The quantizer step size for each coefficient $S_{v,u}$ is the value of the corresponding element $Q_{v,u}$ from the quantization table specified by the frame parameter T_{qi}." [12, p. 28] The quantized DCT coefficients $Sq_{v,u}$ are calculated by the following formula which reduces the coefficient values by $Q_{v,u}$.

$$Sq_{v,u} \quad = \quad round \quad (\frac{Sq_{v,u}}{Q_{v,u}}) \tag{3.1}$$

4. "After quantization, the DC coefficient and the 63 AC coefficients are prepared for entropy encoding." [12, p. 15] The quantized DC coefficient of the previous block is used to predict the current quantized DC coefficient, the difference is encoded. Differential encoding is not done for the other 63 quantized AC coefficients but the 8x8 matrix is converted to a vector using a static zig-zag pattern.

5. The quantized coefficients are then passed to the entropy encoding which compresses the data further. For entropy coding, either Huffman coding or arithmetic coding can be used. The more common one, Huffman coding replaces variable bit sequences by Huffman codes, dependent of their frequency [12, p. 15].

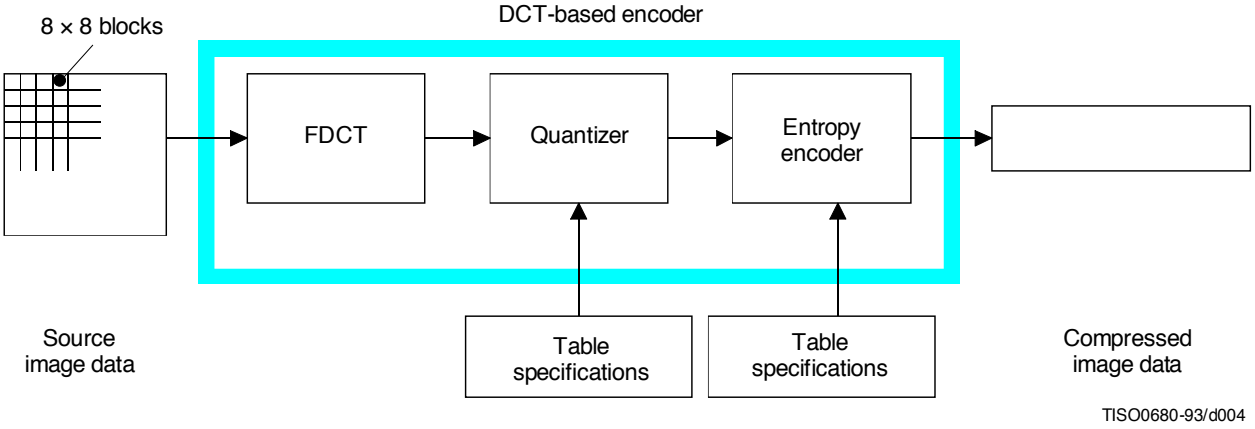


Figure 3.1: DCT-based encoder simplified diagram [12, p. 15]

Beside the act of actual compressing the image data, it is important to define how a multi component image, like RGB, is read. "In sequential mode, encoding is non-interleaved if the encoder

compresses all image data units in component A before beginning component B, and then in turn all of B before C. Encoding is interleaved if the encoder compresses a data unit from A, a data unit from B, a data unit from C, then back to A, etc." [12, p. 19] "Related to the concepts of multiple-component interleave is the minimum coded unit (MCU). If the compressed image data is non-interleaved, the MCU is defined to be one data unit. [...] If the compressed data is interleaved, the MCU contains one or more data units from each component." [12, p. 21]

3.2 Operation modes

The JPEG compression can be used in four different ways called operation modes [12, p. 17f]. In each of these modes a subset of the above described compression processes are used:

1. **Sequential DCT-based:** This mode uses exactly the processes described in section 3.1 and is also called "baseline mode". Every block of the source image data is encoded and sequentially written to the image data. Since only the DCT-coefficients of the current block needs to be stored in memory, it minimizes the size of the needed buffer.

2. **Progressive DCT-based:** The image is encoded in multiple scans. Like the sequential mode, 8x8 blocks are transformed using DCT, but instead of entropy coding the resulting coefficients, they are stored in an image-sized coefficient memory buffer. After cosine transform of the whole image and before entropy coding, one of the following actions is applied to the stored coefficients:

   - **spectral selection:** Different parts of the frequency spectrum are stored within scans. The first scan contains all DC coefficients and some AC coefficients. All remaining AC coefficients are stored in the other scans.
   - **successive approximation:** The image is not split up by its frequencies but by its resolution. Therefore, the first scan contains a specified count of the most significant bits (MSB) of the coefficients, whereas the following scans contain the differences to the MSBs. This results in a low resolution image in the first scan which will be successive upsampled with each further scan.

3. **Lossless:** The lossless compression doesn't make use of DCT or quantization but still uses entropy coding for compression since this can be decoded without loss of information.

4. **Hierarchical:** The image is encoded at multiple resolutions. First, the image gets downsampled to the lowest desired resolution and encoded. Afterwards the low resolution image is upsampled to various resolutions and the difference to the original image is stored in different scans. This allows applications fast computation of different image resolutions.

3.3 Structure of the stored data

As we have seen in the previous section, JPEG compression can be used in many different ways. To ensure compatibility between the encoding and decoding entities, meta information needs to be stored within the JPEG file. The JPEG standard defines a broad set of so called ”markers” which define the start of a certain meta information entity. A marker usually follows the type-length-value (TLV) principle, providing first the marker type then the length of the information entity, storing the actual information at the end. Table 3.1 shows a shortened list of all JPEG marker, which will be referenced later.

Code Assignment	Symbol	Description
0xFFC4	DHT	Define Huffman table(s)
0xFFCC	DAC	Define arithmetic coding conditioning(s)
0xFFD0 - 0xFFD7	RST_m	Restart with modulo 9 count ”m”
0xFFD8	SOI	Start of image
0xFFD9	EOI	End of image
0xFFDA	SOS	Start of scan
0xFFDB	DQT	Define quantization table(s)
0xFFDC	DNL	Define number of lines
0xFFDD	DRI	Define restart interval
0xFFDE	DHP	Define hierarchical progression
0xFFDF	EXP	Expand reference component(s)
0xFFE0 - 0xFFEF	APP_n	Reserved for application segments
0xFFF0 - 0xFFFD	JPG_n	Reserved for JPEG extensions
0xFFFE	COM	Comment

Table 3.1: ”Marker code assignments” [12, p. 32]

The JPEG standard defines a certain order in which these marker and their consecutive data are allowed to occur. Figure 3.2 shows a general overview of the JPEG structure as a syntax diagram. It can be seen that the only values which need to be the same in every image is the ”Start of Image” (SOI) marker and the ”End of Image” (EOI) marker. The structure of a JPEG file can be separated into a header part and the content part which is called ”frame”. The double bordered elements in Figure 3.2 aren’t specific markers but place holders for different kind of markers shown in annex A.2. For example, the element ”Tables/miscellaneous” can contain multiple different markers like a quantization table (DQT) or Huffman table (DHT). All of these markers are followed, like described previously, by a header which contains a length field and different attributes. It can be seen that an analysis of these elements needs a deep knowledge of the JPEG specification.

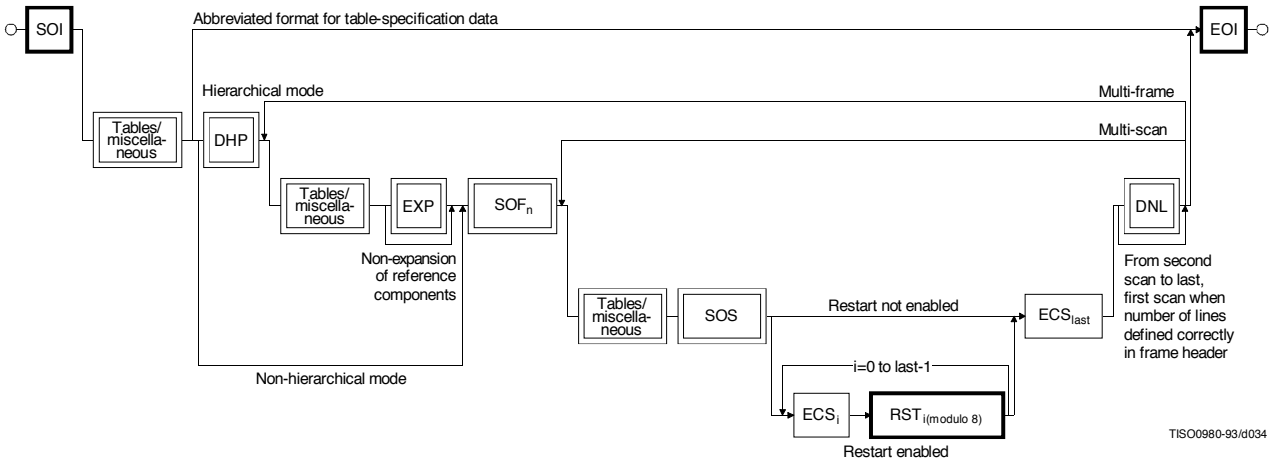


Figure 3.2: Flow of compressed data syntax [12, p. 48]

3.3.1 JPEG header

The term ”JPEG header” is not an official term defined by the standard, but is used in this thesis to denote all data structures in the beginning of the JPEG file which contain general meta information. Examples for such data structures are markers which are allowed in the syntax element ”Tables/miscellaneous” like quantization tables (DQT), Huffman coding tables (DHT) or a comment section (COM). The application markers (APP_n) have a special status in the JPEG header. They can be used by applications to store their proprietary information within the JPEG image. Also container formats like JFIF, described in section 3.4, make use of an application marker to allow compatibility across heterogeneous systems. In valid JPEG files, the application marker APP0 must directly follow the SOI marker. The JPEG header also specifies the compression indirectly. Instead of storing the information whether the mode is progressive or sequential, the chosen marker make it possible to conclude the mode. For example, if the baseline sequential mode is used, the marker SOF_0 (`0xFFC0`) is used, whereas SOF_2 (`0xFFC2`) is used for the progressive mode.

3.3.2 JPEG frame

A frame specifies exactly one image in a defined resolution. Although it is possible that multi-resolution image files, using the hierarchical mode, store more than one frame, mainly the sequential and progressive mode are used, storing only one frame. The frame is described with a ”start of frame” marker followed by meta information of the frame. Like in the JPEG header, every frame can additionally contain its own tables like quantization or Huffman table, which are used only for this

frame. After frame definition, the ”start of scan” marker (SOS) signals the beginning of the actual compressed data. A frame can contain more than one scan. Since the structure of JPEG is highly dependent on those marker bytes, it needs to prevent valid marker to occur unintentionally in the compressed image data. To circumvent random coincident, the procedure of ”byte-stuffing” is used, inserting a zero byte after each `0xFF` byte. This results in the escape sequences `0xFF00` which are reversed in the decoding process. [12, p. 91]

3.4 The interchange format

The JPEG standard claims the need of a so called ”interchange format” [12, p. 25], which extends the file in a way to exchange JPEG bitstreams between a wide variety of platforms and applications. The most common interchange formats for JPEG are Exif, which is heavily used in digital photography and the JPEG File Interchange Format (JFIF). Since JFIF is defined as the default interchange format for the MIME type JPEG by RFC2046, it is mainly used for an exchange of images over the internet [34].

3.4.1 JPEG File Interchange Format (JFIF)

JFIF is a minimal file format to fulfill the requirements of a JPEG interchange format. It adds additional attributes to a JPEG file and allows the storage of a thumbnail within the image file. Additionally, if JFIF is used, RGB can no longer be used as color space. Instead, it can be either grayscale (one component) or YCbCr (luma/blue-chroma/red-chroma - three component). Technically spoken, the JFIF data must start right after the start sequence of SOI and APP0 (`0xFFD8`, `0xFFE0`). The fields described in Table 3.2 are allowed to follow the APP0 marker.

3.4.2 Exchangeable Image File Format (Exif)

In 1998, the ”Exchangeable Image File Format” (Exif) was published by the Japan Electronic Industry Development Association [36, p. 1] to ensure data compatibility and exchangeability for image data recorded by digital cameras. Beside image data, the Exif specification can also be used for audio media but only the image file specification will be described in this section.

Figure 3.3 shows, how Exif is embedded into JPEG compressed image files. Like JFIF, the Exif data is written in an application segment which starts with a APP_n marker. In contrast to JFIF, Exif uses the APP1 marker which must directly follow the SOI marker to comply with the JPEG specification. As defined in the JPEG standard, the APP1 section is not allowed to exceed a length of 64 kB. The rest of the image file is like described previously. Various table definitions are stored after the APP-marker followed by the frame header and the scan header(s) which contain(s) the compressed image data.

Type	Length	Description
length	2 bytes	Total APP0 field byte count, including the byte count value (2 bytes), but excluding the APP0 marker itself
identifier	5 bytes	`0x4A46494600` (String ”JFIF” with terminating zero)
version	2 bytes	The most significant byte is used for major revisions, the least significant byte for minor revisions. (e.g. `0x0102`)
units	1 byte	Units for the X and Y densities (0=pixel aspect ration; 1=dots per inch; 2=dots per cm)
Xdensity	2 bytes	Horizontal pixel density
Ydensity	2 bytes	Vertical pixel density
Xthumbnail	1 bytes	Thumbnail horizontal pixel count
Ythumbnail	1 bytes	Thumbnail vertical pixel count
RGB_n	3n bytes	Packed (24-bit) RGB values for the thumbnail pixels, n = Xthumbnail * Ythumbnail

Table 3.2: JFIF attribute list [35, p. 5]

Since the Exif header has a variable size, its size is stored in the beginning of the header, followed by an identifier code which stores the string ”Exif” followed by a terminating zero. Information about the image such as image dimension, resolution or a thumbnail is stored using the ”Tagged Image File Format” (TIFF), which uses a list of predefined tags to store meta data. The TIFF tags are stored in a maximum of two IFD (Image File Directory) sections, whereas IFD0 describes the primary image and IFD1 the optional thumbnail. At the end of the Exif header, an optional JPEG thumbnail can be embedded. Except the missing APP-marker, the thumbnail follows the normal JPEG specification. Due to the huge amount of possible TIFF tags and Exif meta information, defined in the standard [36, p. 17ff], a description will be omitted.

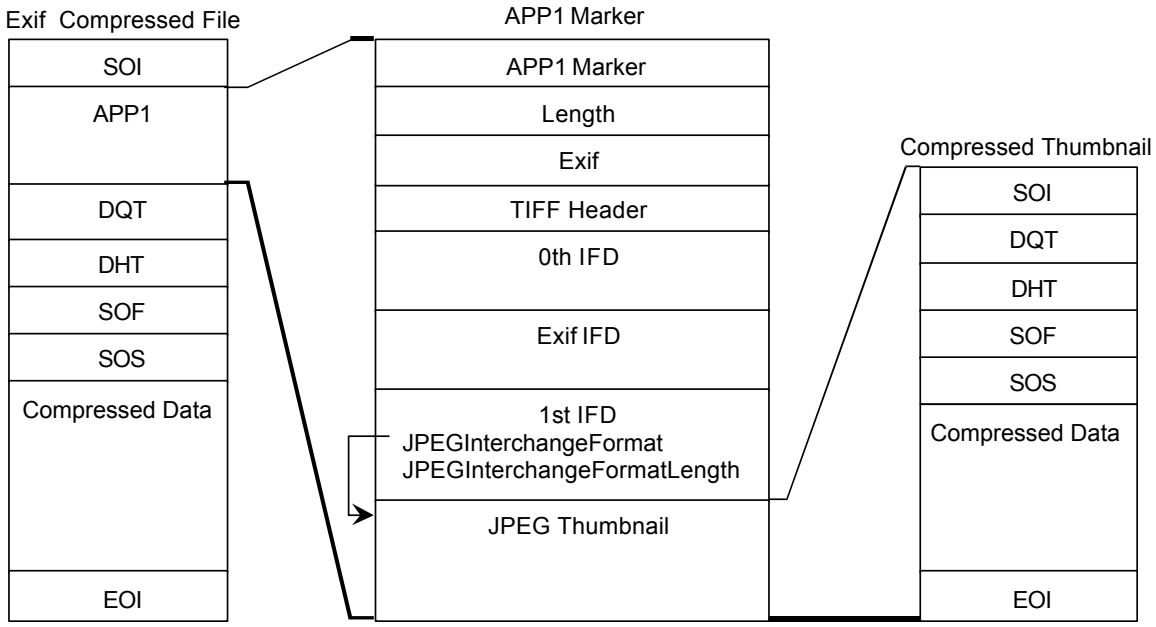


Figure 3.3: ”Structure of Exif file with compressed thumbnail” [36, p. 16]

Proposed JPEG file carver

This chapter presents the algorithms and methods used for carving fragmented JPEG files. Instead of reinventing a file carver, the proposed JPEG file carver extends the capabilities of the ”multimedia file carver” presented by Poisel et al. [37, p. 26]. First, the general architecture of the multimedia file carver will be described, outlining the proper interfaces, which can be used to extend the file carver with file type specific carving capabilities, like for JPEG. The second part of this chapter shows in great detail how the multimedia file carver is extended in order to support JPEG files and which algorithms are used to do so.

4.1 Architecture of the file carver

The architecture of the multimedia file carver is based on smart carving, proposed by Pal and Memon [3, p. 67] and shown in Figure 4.1. The tasks of these steps are not described in this chapter, since it is already described in great detail in chapter 2.1.4. The JPEG file carver will extend the steps of the architecture in a way to enhance its capabilities for JPEG file carving. Because ”preprocessing” and ”postprocessing” are independent steps in respect to the file type, they won’t be, in contrast to ”Collation” and ”Reassembly”, subjects of the JPEG file carver. This section will focus on a general overview of the technical architecture, provided by class diagrams and description of important interfaces supported by the multimedia file carver.

4.1.1 General overview

The multimedia file carver is publicly available under the GNU Lesser General Public License v3.0 [8] and is programmed in Python and C. As described in a feasibility study of Poisel and Tjoa [38, p. 58f], software from the following projects is used to extend the capabilities of the file carver:

- **PySide** offers the ability for platform independent rapid prototyping in Python. The graphical user interface for visualizing the carving results is developed by using this library.
- **ffmpeg** is used in the original multimedia file carver for decoding video data. It supports all major video formats like MPEG-4 H.264, FLV H.264 and WebM VP8. This library is not used

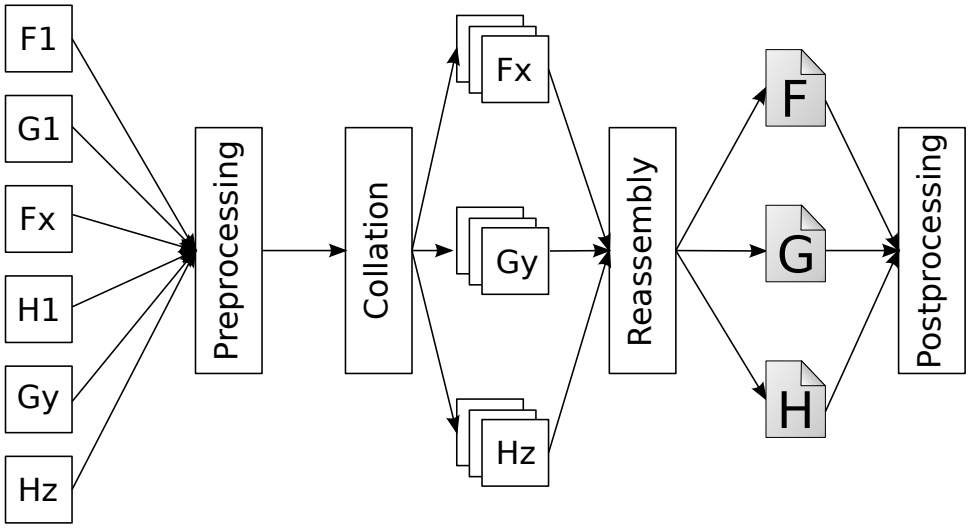


Figure 4.1: ”Architecture of the multimedia file carver” [8, p. 26]

for carving JPEG files, but still needed to support forensic analysis of video data.

- **Python Imaging Library (PIL)** is heavily used by the proposed JPEG file carver to calculate candidate weights between two carved JPEG file fragments. It offers the ability to decode JPEG files and perform comparisons on a pixel basis.
- **The Sleuth Kit (TSK)** is a well known forensic toolbox which extends the system with useful forensic commands. It is used in the preprocessing step of the file carving process to identify file system structures.

4.1.2 Architecture

The architecture of the program is strongly oriented to the smart carving architecture shown in Figure 4.1. As the file carver is mainly programmed in Python, each file carving step is encapsulated in a module of the same name to ensure modularity and clarity. Class diagrams will show the architecture of the modules, additional flow charts will visualize their functionalities and internal relations.

4.1.2.1 Preprocessing

The preprocessing phase is in charge of preparing the media for further file carving activities. The corresponding module mainly consists of two contexts, preprocessing and fsstat, which can be seen in

the class diagram of Figure 4.2. The second context, fsstat_context, uses ”The Sleuth Kit” to retrieve media information of the file system (`CFsStatContext`) and stores it into the class `CFsOptions` which is returned to the caller. The preprocessing context starts the actual preprocessing steps with the class `CResultThread` and `CPreprocessing`. It shall be remarked, that `CPreprocessing` initiates the collation phase after successful preprocessing. This is done due to performance reasons.

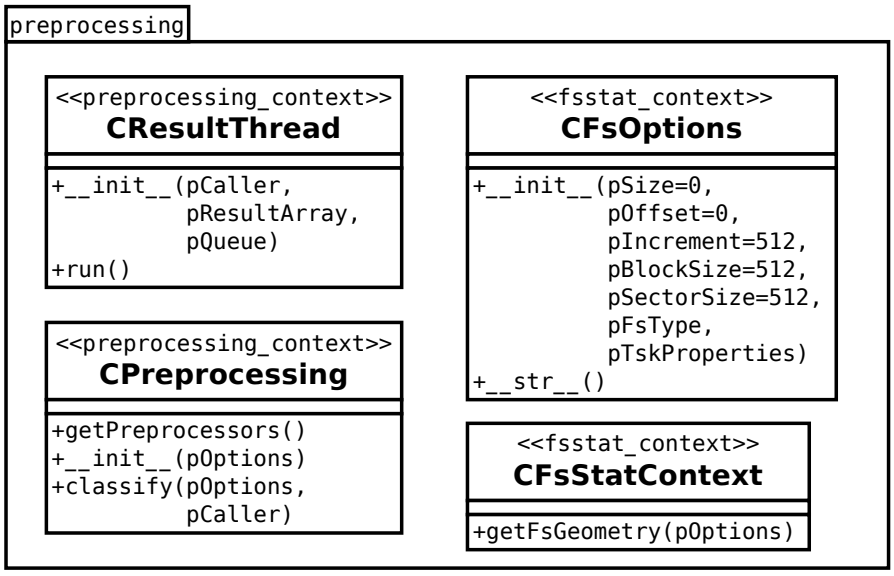


Figure 4.2: Class diagram of the preprocessing phase

4.1.2.2 Collation

The collation module classifies data clusters by their file type. It additionally groups them into fragments to increase the performance of the reassembly phase. Since the classification algorithm and the information storage for every cluster is very computational and memory intensive, it is written in C to increase the performance of these tasks. Figure 4.3 shows a class diagram of the collation module. The shared object `libblock_reader.so` contains the C-functions for the classification and is accessed by the Python class `CFragmentClassifier`, which gets, as already described before, called by the class `CPreprocessing`. The architecture behind the shared object will be described later. The classes `FileType`, `CFragmentStruct`, `ClassifyT`, `CBlockOptions` and `CFragmentCollection` are needed to exchange data with the shared object, which defines the same objects as struct data types. If `CFragmentClassifier` classifies the media using the shared object `libblock_reader.so`, it receives classified blocks which are grouped to fragments, stored in the collection class `CFragments`.

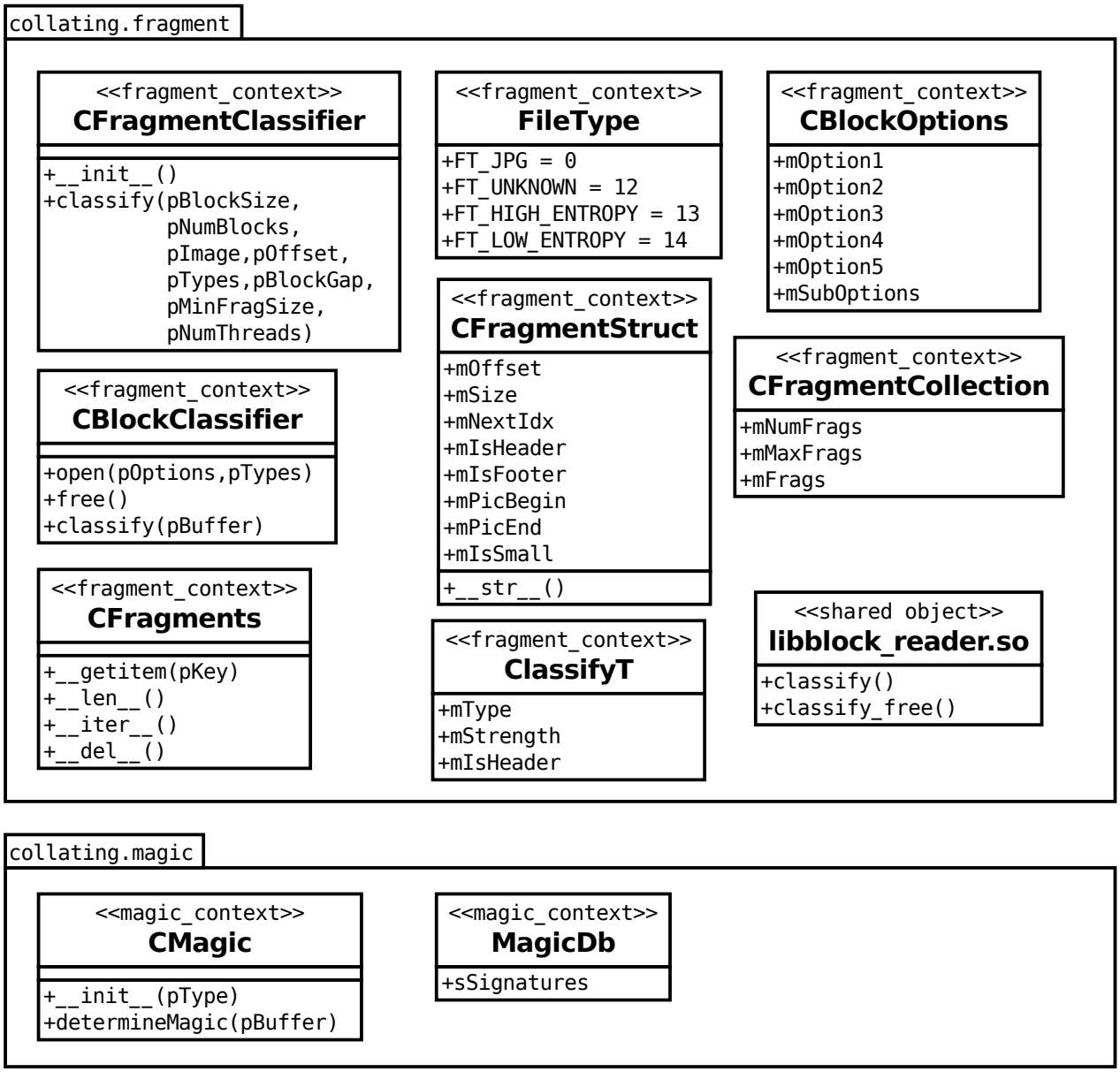


Figure 4.3: Class diagram of the collation phase

For a better understanding of the file carver's classification mechanisms, we need to take a closer look at the used shared object, which is visualized in Figure 4.4. `libblock_reader.so` combines different c files and shared objects which shall be explained here clearly:

- `block_reader.c` calls all of the next functions. It classifies first the blocks of a given media using the implemented classification algorithms in `fragment_collection.c`, storing a map of the blocks in `block_collection.c`. These blocks are passed to `fragment_collection.c` in order to group them into fragments.
- `fragment_classifier.c` needs to get initialized first to define for which data types should be classified. The more data types are searched for, the more complex the analysis becomes. After initialization, blocks are analyzed for file type specific features, using algorithms described in section 4.2. As an example for a classification algorithm, the fragment classifier uses `entropy.c` to calculate the block's entropy, to make a classification decision.
- `fragment_collection.c` is called by the `block_reader.c` and receives a set of blocks which are grouped as fragments. These fragments are then returned to the calling Python class and used for the file reassembly.

4.1.2.3 Reassembly

In the third step of the smart carving procedure, the identified fragments are reassembled to their original files. To do so, the fragments are compared to each other. The resulting candidate weights are used by the Parallel Unique Path (PUP) algorithm to retrieve the correct order.

Basically, the reassembly module is designed to support different reassembly algorithms. To extend the file carver with a new algorithm, the base class `CReassembly` needs to be derived and `assemble_impl` implemented. `CReassemblyPUP` is such a derived algorithm, which implements the Parallel Unique Path (PUP) algorithm. Since the algorithm needs to know how to compare the file type specific fragments to each other, it needs a file type handler at runtime, which does the file type specific work. As an example, if the user chooses to reassembly JPEG, the file carver first creates an object of `CJpegHandler`, which has the ability to compare JPEG fragments to each other and do JPEG specific operations like analyzing the JPEG meta information. This file type handler is passed to the constructor of the reassembly algorithm `CReassemblyPUP`. As you can see in Figure 4.5, the `CAbstractFileTypeHandler` requires to implement three methods which are needed for a composition with the reassembly algorithm. `prepareFiles` is called at the beginning of the reassembly algorithm, initializing the header fragments and reassembly paths. As already described, `compareFrags` is the essential method for fragment comparisons which is used by the reassembly algorithm to calculate the candidate weights.

Another feature of the reassembly module is the abstraction of file reassemblies (also called fragmentation paths) as a file of the given data type. This is done by adding arbitrary file fragments, starting with the header fragment, to the suitable file type specific child of `CFile`. Since the comparison function of the reassembly algorithm is implemented to receive a `CFile` object and a fragment, it is possible to compare a whole fragmentation path with a new fragment, allowing more versatile

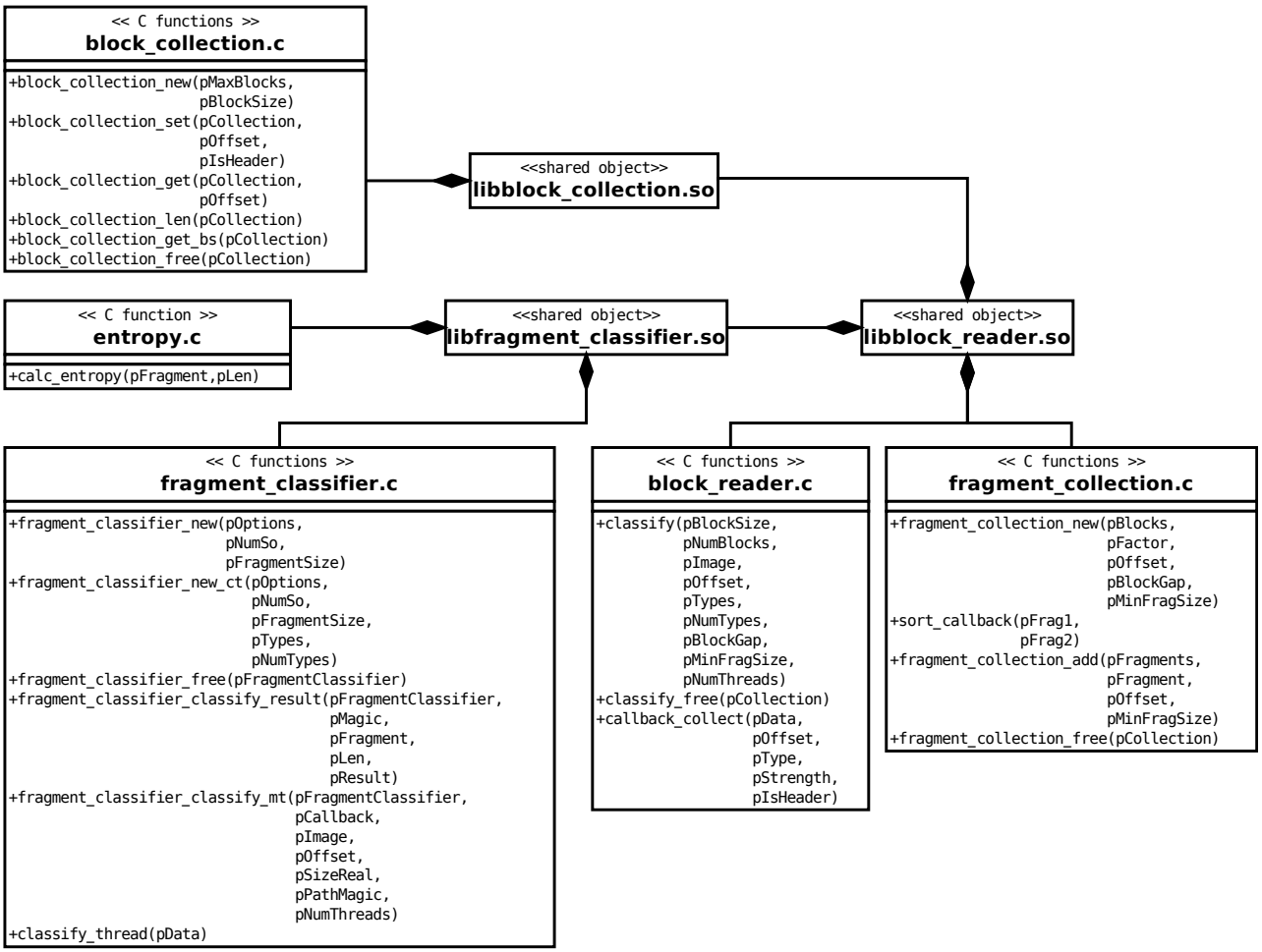


Figure 4.4: Diagram of the C shared objects used for fragment classification

comparison methods. This feature is also necessary for JPEG comparison since it is impossible to semantically compare two JPEG fragments without contextual information like the Huffman encoding or DCT-coefficients to each other.

The next file type specific class is the decoder. It is used to properly read and write encoded files like H.264 or JPEG and is also suitable to convert media files to another media format. For example, the `CJpegDecoder` allows easily to write a JPEG file as a bitmap (BMP). This can be necessary for easy access of the media data without using a JPEG decoder again.

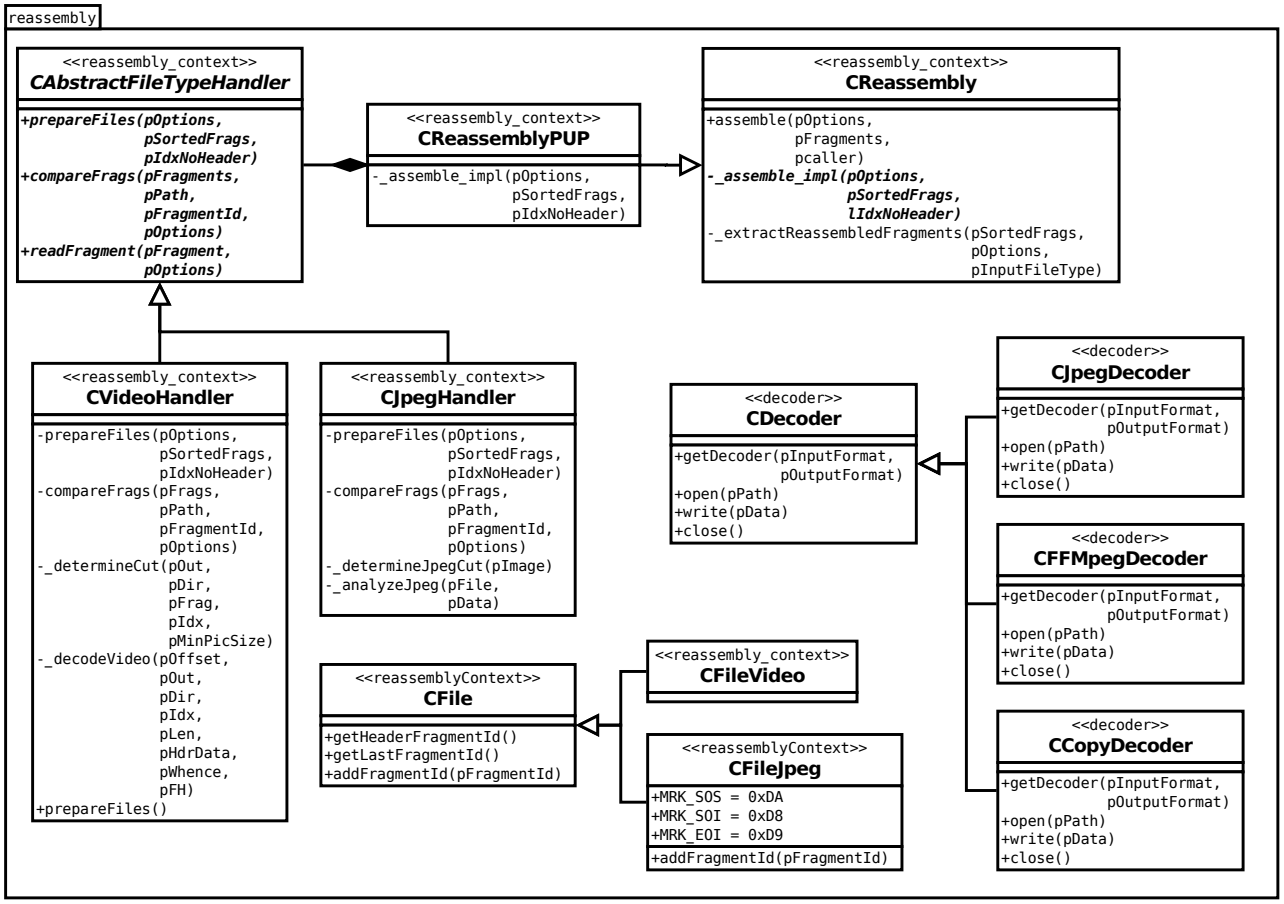


Figure 4.5: Class diagram of the reassembly phase

4.1.3 Workflow

Section 4.1.2 showed the architecture of the multimedia file carver in form of class diagrams. In this section concentrates on presenting the overall program flow of the file carver. Figure 4.6 shows a very simplified flow diagram of the whole program. It is reduced to the absolute core functions to ensure an overview. Vertically it is segmented into the smart carving phases to give an understanding how the interact with each other.

The class `CContext` is responsible for starting as well the graphical user interface (GUI) and the carving phases. Therefore it could be called the entry point and main loop of the program. After starting the GUI, the user can specify a data storage media for the forensic analysis, choose a supported file type and configure the algorithms used in the smart carving phases. If the user chooses to classify, the `runClassify()` method of the class `CPreprocessing` is called. Be-

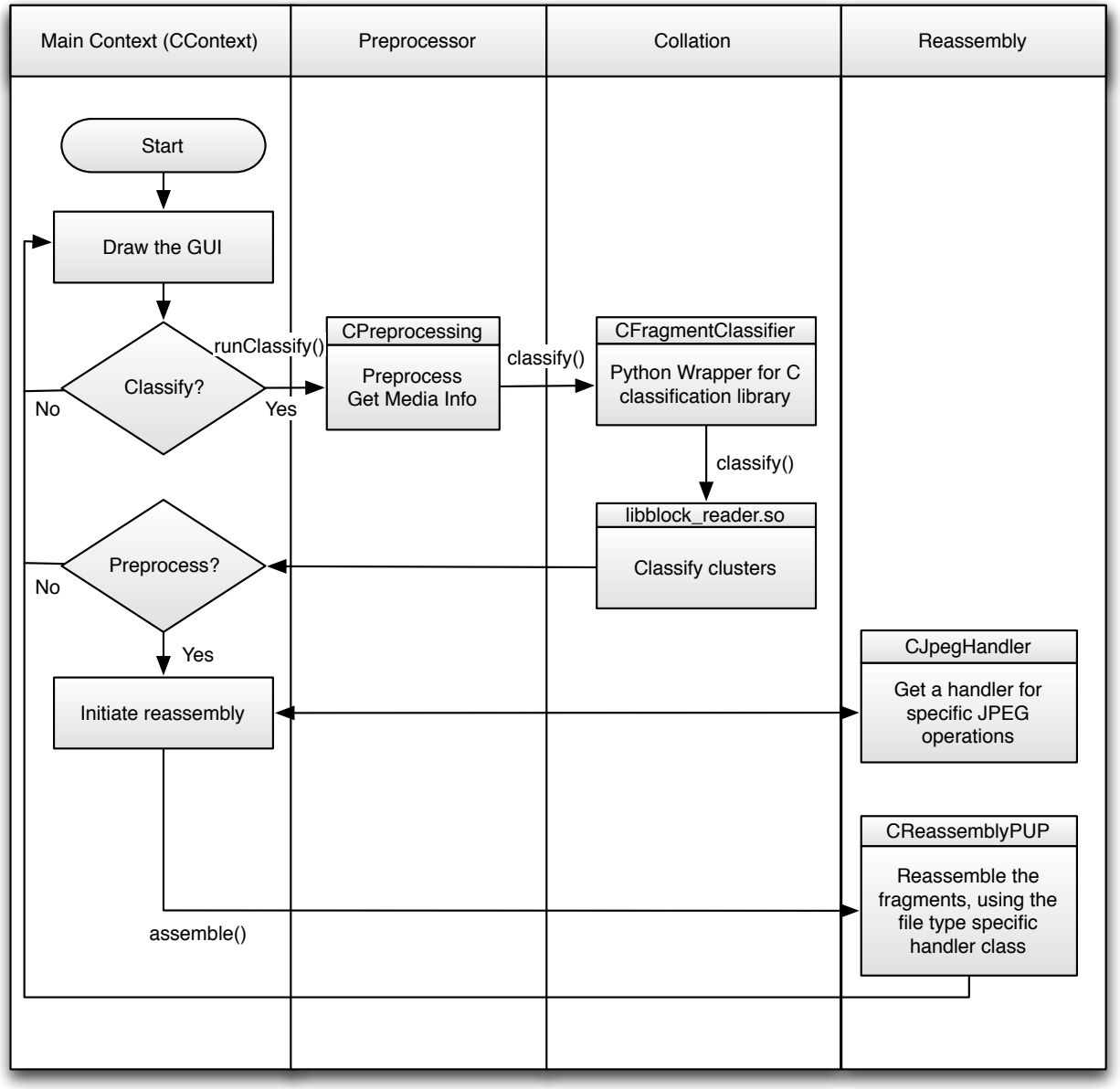


Figure 4.6: Workflow of the file carver, carving a disk image for JPEG files

fore classifying, the preprocessor gathers media information for the later processing. Afterwards, `CFragmentClassifier.classify()` is called, which calls the proper native C functions for a block classification. In this phase, the data blocks (clusters) are analyzed for the specified file type.

Classified blocks are also grouped to fragments and returned to the main context `CContext`. In the next step, if chosen by the user, the identified fragments are reassembled. To do so, an object of `CJpegHandler` is created, which has the ability to compare JPEG fragments to each other and do JPEG specific operations like analyzing the JPEG meta information. This file type handler is passed to the constructor of the reassembly algorithm `CReassemblyPUP`. The method `reassemble()` is invoked of this class, which returns after finishing the reassembling process.

4.2 Specification of the collation algorithm

Like already described in chapter 2.2, collation algorithms are used to determine the file type of a data cluster. Since a data cluster is usually a very small unit of information, the classification decision is error prone. The quality of a collation algorithm is defined by its ability to correctly identify the file types of clusters while minimizing the amounts of false-positives. Roussev [14, p. 12] and Veenman [15, p. 397] stated that only a file type specific algorithm is able to achieve highly reliable classification. Because of this conclusion, the collation algorithm described in this chapter uses a feature based algorithm which is optimized for JPEG file type identification.

Beside the feature based algorithm, the entropy of the fragments is used to quickly reject fragments. The entropy describes the amount of information hold by the data. Like MPEG or ZIP, JPEG is a compressed data format with a very high entropy value. This means that n bits of a high entropy file stores more information than the same amount of bits in a low entropy file type. If we want to identify JPEG file types, we can test the hypothesis of a high entropy like shown in listing 4.1

```
1  lEntropy = calc_entropy(pFragment, pLen);
2  if (lEntropy > 0.9)
3  {
4      pResult->mType = FT_HIGH_ENTROPY;
5      pResult->mStrength = 1;
6
7      // -- Additional check for JPEG is done here! --
8  }
9  else
10 {
11     pResult->mType = FT_LOW_ENTROPY;
12     pResult->mStrength = 1;
13 }
```

Listing 4.1: fragment_classifier.c

Chapter 3.3.2 showed already that the JPEG file format is organized by distinct byte sequences called ”marker” which can be used for classification. All of these markers consist of two bytes, of which the first one has always the decimal value 255. The second byte describes the type of marker, ranging from `0xC4` to `0xFE`. Additionally, `0xFF00` is used as escape sequence for a `0xFF` byte in the compressed image data. Since this escape sequence occurs very often within the image data, it can be used as a reliable JPEG feature to search for. Because this feature could also occur in other non

JPEG fragments, another feature is needed to improve the reliability of the algorithm. As previously described, only a little part of the possible marker range is actively used as control sequences. All other markers, `0xFF01` to `0xFFC3` and `0xFFFF` are not specified by the standard and therefore should not occur. This behavior is used to create a negative feature, rejecting all clusters from the classification algorithm which contains one of these illegal markers.

The decision tree for classifying data, using the previously specified features is shown in Figure 4.7 as a flow diagram. It can be seen that as soon as an illegal JPEG marker is identified, the data is disproved to be image data. Listing 4.2 shows how this algorithm is implemented in the programming language C.

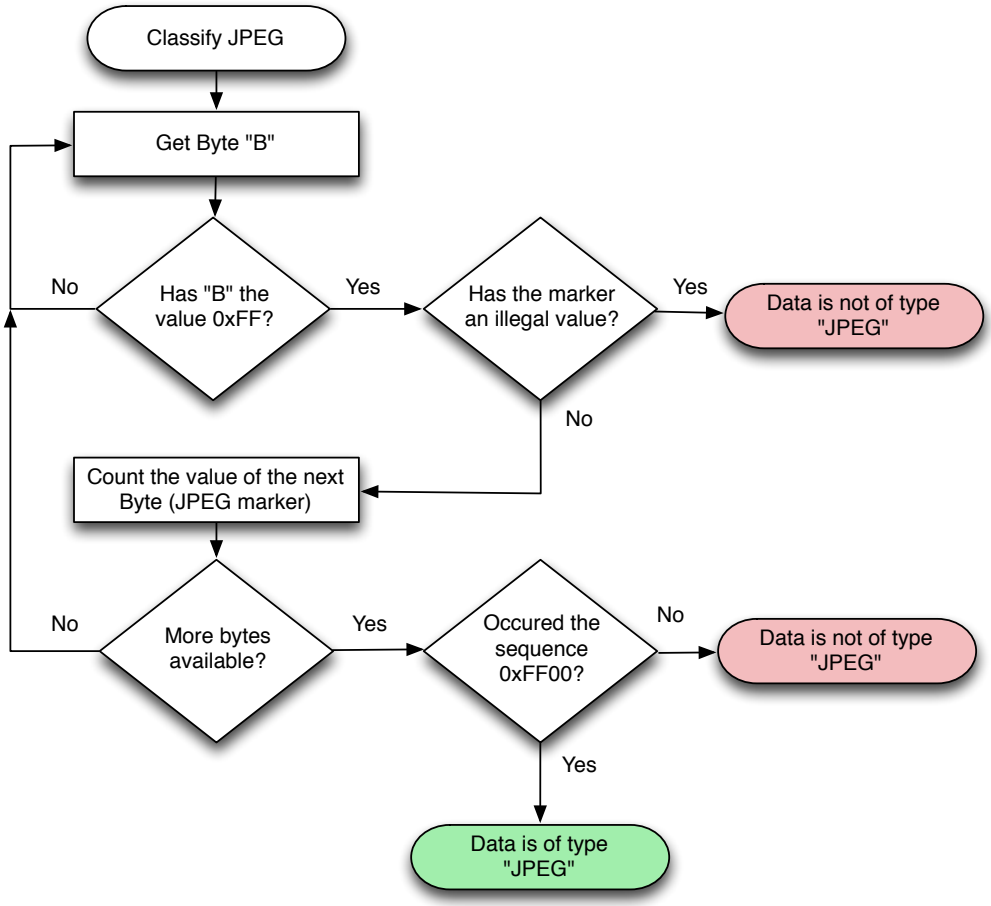


Figure 4.7: Flow diagram of JPEG classification

```
1  for (lCnt = 0; lCnt < pLen - 1; lCnt++)
2  {
3      if (pFragment[lCnt] == 0xFF)
4      {
5          /* these usually occur in JPEG file fragments */
6          if (pFragment[lCnt + 1] == 0x00)
7          {
8              lCntJpeg++;
9          }
10         /* illegal sequence in JPEG files */
11         else if (pFragment[lCnt + 1] < 0xC0 || pFragment[lCnt + 1] > 0xFE)
12         {
13             lCntJpeg = 0;
14             break;
15         }
16     }
17 }
18
19 if (lCntJpeg > 0)
20 {
21     pResult->mType = FT_JPG;
22     pResult->mStrength = 1;
23     return pResult->mStrength;
24 }
```

Listing 4.2: Source code for JPEG classification

4.3 Specification of the reassembly algorithm

The reassembly algorithm follows the collation algorithm and brings the identified fragments back into the correct order. Although the reassembly of fragmented files seems to be a simple task, it needs various error-prone tasks to achieve this goal. Firstly, the reassembly can only work with fragments, but the collation algorithm does only result a list of classified blocks. Therefore, the first step for a correct reassembly is to group blocks to fragments of the specified file type, which is called ”Grouping”. If the fragments are grouped together correctly, they can be passed to the reassembly algorithm, which has the task to find the correct order of fragments, based on comparisons between themselves. The efficiency of the reassembly algorithm has a major effect to the computational complexity and processing time of the overall carving process. Since the fragment comparisons of the reassembly phase are highly dependent on the file type, a file type specific comparison algorithm has to be implemented which is called ”weighting”. This weight represents the visual similarity of two JPEG fragments.

4.3.1 Grouping

The grouping approach for building fragments out of classified data clusters was designed by Poisel et al. [37, p. 8f]. It builds one key feature of the multimedia file carver and is used for carving fragmented JPEG files.

File type classification contains the risk of determining the file type wrongly, resulting in false negative classifications. Therefore, grouping is needed as a fault tolerance mechanism to be able to create fragments even if a cluster in between is wrongly classified. According to Garfinkel [9, p. 4], it is very unlikely that a file is fragmented into many parts with sizes of only a few sectors. This is the reason why clusters with the same file type, which are close together, are grouped into fragments. To control the grouping algorithm, Poisel describes two parameters, the ”block-gap” and the minimum fragment size.

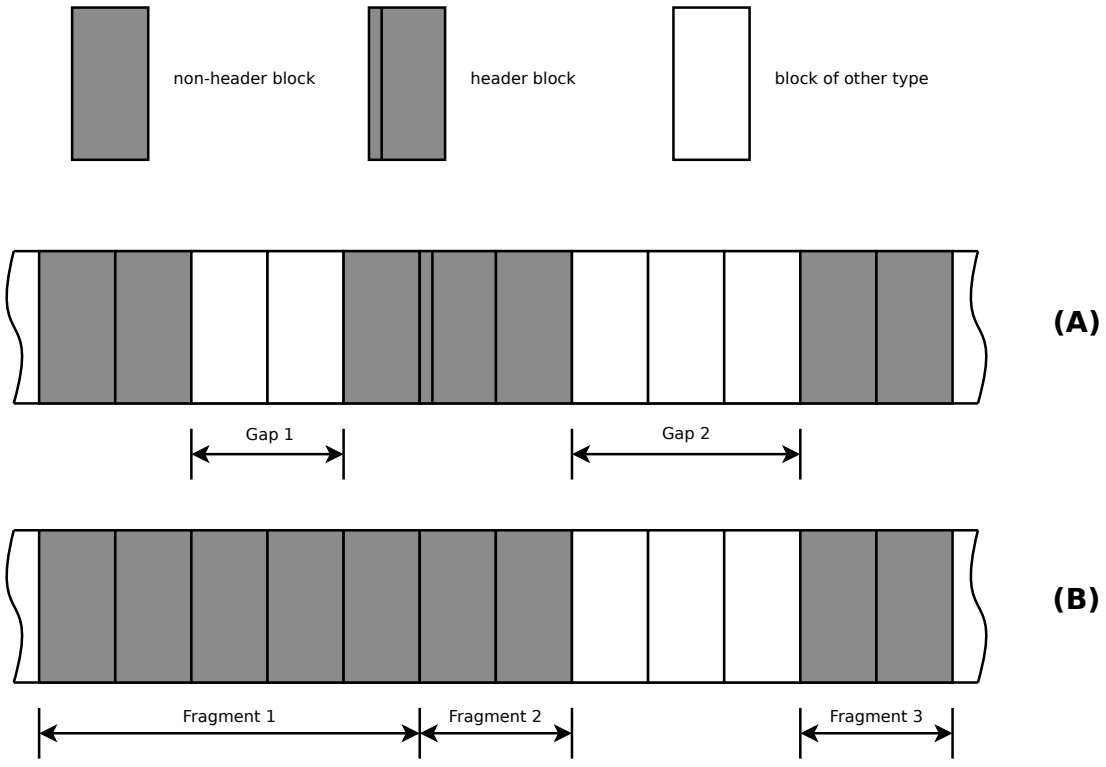


Figure 4.8: Summary of fragments [37, p. 9]

The ”block-gap” specifies, by how many blocks a fragment is allowed to be separated, until they are considered as two different fragments. If the blocks in between are less then the ”block-gap”,

they are assumed to be false negatives and classified like the surrounding ones. Figure 4.8A shows an example for this algorithm. Assuming a block-size of two, ”Gap 1” would be considered as wrong classification and added to the surrounding blocks, forming ”Fragment 1”. Because ”Gap 2” is too big, it separates ”Fragment 2” and ”Fragment 3” from each other.

The second parameter, minimum fragment size refers to the minimum number of blocks that make a fragment. A minimum fragment size of three would prevent ”Fragment 3” to be considered in the further reassembly process. Although ”Fragment 2” has the same size, it is a valid fragment because it is a header fragment.

4.3.2 Weighting

To be able to build up a graph which represents the optimal reassembly of the given fragments, weights have to be assigned between the fragments to describe the likelihood how well they fit together. Assuming there are *i* header fragments H_i and *j* non header fragments F_j, the weight is calculated by applying $W(H_i, F_j)$. Because of the nature of the JPEG file format, a weights calculation of two non header fragments can’t be done. This is reasoned because the data can’t be analyzed due to the lack of information necessary for decoding like quantization tables or entropy coding tables. If a file has more than two fragments, H_i specifies both the header fragments and the identified fragments belonging to this header fragment. A comparison should only succeed if both H_i and F_j belong to each other and all data between those two fragments is correctly in place. Again, this may only be true for the JPEG file type because not all file types are as dependent on their context as JPEG.

A comparison between JPEG fragments requires the ability to decode the fragments. As described in chapter 3.3, a JPEG image contains the compressed image data as well as the meta information needed to decode it. For example, the quantization table is needed to revert the discrete cosine transform. If this information is not in place, the fragment can’t be decoded and therefore not analyzed within the weighting process. This problem is solved by adding the non header fragment to the end of the header fragment like shown in Figure 4.10. Using this method, it is possible to compare the image data of the header fragment and the non header fragment. Since the multi component image data is mostly stored interlaced, which means that the bytes holding the color information iterate, the color channels can shift for the rest of the image if some data is missing (see Figure 4.9). Therefore, it is very likely that a comparison fails even if less image data is missing.

The weight between two JPEG fragments can be calculated using various approaches. Although not all of these methods are implemented in the file carver, they can be combined and may improve the reliability of the weighting score:

- **Histogram intersection:** using this approach, the distribution of the image values of two image segments are compared. The more similar two images are, the less differences can be seen in the histogram, which allows the easy calculation of a weight. If the fragments belong to each other, the difference should be minimal. The big problem of this algorithm is, if data is missing

Figure 4.9: Intense image corruption, caused by a missing small file fragment

in between, the second fragment is shifted which has influence to its image positioning and color. Another problem exists if the fragment is very small (e.g. one cluster). A correct pattern matching is not possible since the small amount of data can't be correctly decoded.

- **Pixel matching:** compares the bordering pixels of two image segments to each other. Based on the average differences of the pixel color values, the similarity of the images can be calculated.
- **Pattern matching:** analyzes like histogram intersection the decoded image data of the fragments. It identifies distinct structures, which spread across the comparing fragments. The algorithm analyzes, whether the comparative fragment complies to the identified geometrical constraints of the second fragment. It is assumed that the graphical analysis of image fragments cause a high computational effort.
- **Huffman decoding:** can not be used to calculate a weight but could be helpful to quickly reject fragments. The stored image data is usually compressed using Huffman coding. For a correct decompression, the correct Huffman table, stored in the header fragment, is mandatory. Assuming that JPEG files have different Huffman tables, which are not compatible to each other, they can be used to validate the data of the second fragment. The Huffman symbols in the second fragment need to comply with the stored Huffman table in the first fragment. For this approach, a Huffman decoder needs to be implemented or adapted.

4.3.2.1 Histogram intersection

In respect to the computational performance and the goal for a general approach, the file carver was first implemented to use histogram intersection for the calculation of weights between file fragments. To explain the algorithm, it is assumed to have two multiple fragmented image files, of which one shows colorful juggling balls with an image resolution of 800 x 600 pixels. Assuming to have the header fragment of the ball picture *A*, a non-header fragment *B* and the reassembled image *C*, the algorithm is as follows:

1. Choose the header fragment as fragment A
2. Decode A
3. Determine the image cut and the image fragmentation point
4. Generate image C by attaching B to A
5. Decode C
6. Get the last 8x8 block line of A (until the cut) and the first 8x8 block line of C
7. Calculate a histogram intersection of both lines
8. The result is the weight which can be used for a reassembly decision

Depending whether fragment B is the correct consecutive fragment of A, the resulting image C can either be an incorrect reassembly or a correct one like both shown in Figure 4.10 and 4.11.

Figure 4.10: Incorrect reassembly

Figure 4.11: Correct reassembly

The main purpose of the weights calculation algorithm is to assign a high value to a correct reassembly and a low value to an incorrect reassembly to tell them apart. This activity is done by

steps 3 to 8 and will be now shown in greater detail. To be able to perform a histogram intersection, it is important to know where the image data of the header fragment ends and where the data of the second fragment start starts. This image cut is shown in Figure 4.12 and described by the x- and y-axis coordinates of the first pixel outside the image data, called image fragmentation point. This should not be confused with the fragmentation point which describes the end of the binary fragment data instead of the graphical data.

After identifying the graphical end of the header fragment, the comparative fragment is attached to the end resulting in the larger image *C*. Figure 4.10 shows C with an incorrect fragment, Figure 4.11 the correct one. Instead of calculating the histogram intersection for the whole image, only the image data adjacent to the image cut is used. This is based on the assumption that image areas have a higher correlation in the histogram the closer they are. This assumption can be easily visually seen since the pixels have little color changes in respect to their quantity. To exploit this theory, only the last image line of A is compared to the first image line of B. The height of the line is specified by the JPEG standard.

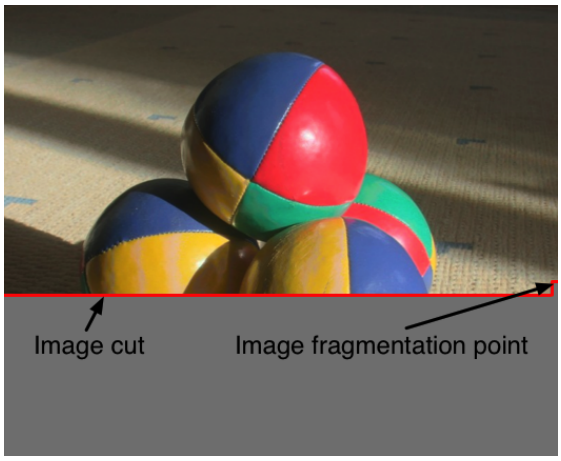


Figure 4.12: Determining image fragmentation point

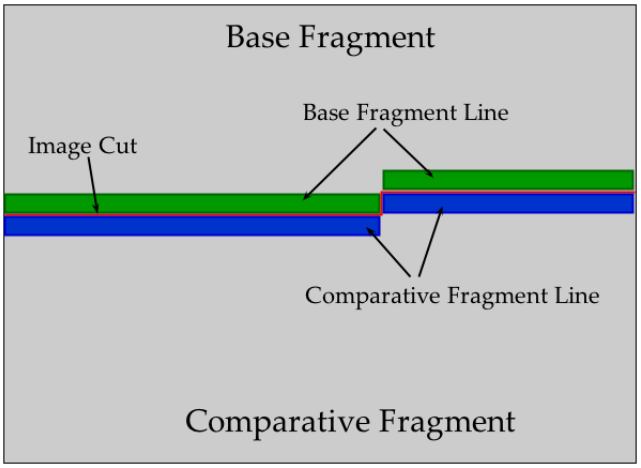


Figure 4.13: Selecting the image lines

In the last step, the histograms of both image lines need to be calculated and compared to result a weight. The histogram counts how often all possible color values of an image occur. Has an image three color channels with a color depth of 256 values each, it returns three times 256 counters how often each value occurs. This can be calculated for each fragment and compared. Listing 4.3 shows how it is implemented. The variable `lWeight` gets every time increased if the difference between both histograms for a distinct value is lower than `pSimilarity`. Therefore, the similarity value, which can be freely chosen between 2 and 256, influences how tight the histogram intersection is. The lower the value is, the more similar both fragments need to be to increase the weight.

```
1 lWeight = 0
2 for lIdx in xrange(len(lHist1)):
3     if abs(lHist1[lIdx] - lHist2[lIdx]) < pSimilarity:
4         lWeight += 1
```

Listing 4.3: reassembly_context.py

The histogram intersection can also be presented in a more generic way. Assuming W to be the weighting function for the fragments A and B, and H to be the histogram function, Equation 4.1 shows the weights calculation. The result of W(A,B) would not be the same than in the source code snippet above, hence this formula should only present the general idea of the weights calculation.

$$W(A,B) = \sum_{n=0}^{255} |H(A)[n] - H(B)[n]| \tag{4.1}$$

Experiments showed, that histogram intersection is too unreliable at distinguishing many different images. This is caused by the fact, that only the average of a complete image line was compared, eliminating all characteristic image features like edges. To overcome this disadvantage, the pixel matching algorithm is used.

4.3.2.2 Pixel Matching

Pixel matching is a simple but effective algorithm, which compares the bordering pixels of the base image and the comparative image together. The closer the image values of each pixels are, the higher is the resulting weighting score. This algorithm starts after step 6 of the histogram intersection algorithm. Instead of calculating the histogram of the complete line, only the neighboring pixels of the base fragment line and the comparative fragment line are used. Figure 4.14 shows how these pixels are compared. Depending on the color depth and color mode of the image, the file carver needs to work with different pixel values. Therefore, the image gets converted to RGB color mode and the resulting comparison weights are percentages based on color depth.

Listing 4.4 gives a closer look on how the algorithm is implemented. Because the image is very likely to be fragmented within an image line, the fragment cut generates two image lines which need to be compared. Line 1 iterates through both image lines. Afterwards (line 2-3), the pixels of the according image line are iterated in their x-axis. Comparing this to Figure 4.14, pixels 1 to 5 will be iterated in the first line and 6 to 7 in the second line. The first pixel of both the base fragment and the comparative fragment are retrieved (line 4-7) and all color channels are compared (line 8-10). The calculated score is the arithmetic mean of the color channel differences. To calculate the overall arithmetic mean of all pixel scores, the amount of pixels needs to be count (line 11). The algorithm is finished when the score is divided by the amount of pixels (line 13). The resulting comparison weight is a percent value in respect to a perfect match.

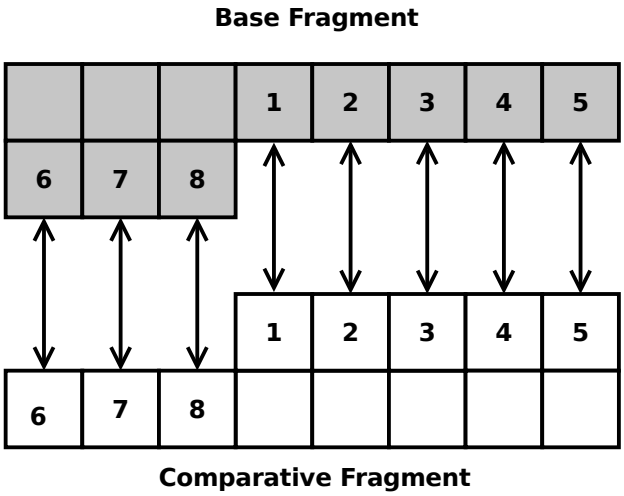


Figure 4.14: Pixel matching algorithm, based on Pal [3, p. 65] which calculates the average difference between the color values of bordering pixels

```
for lLineIdx in xrange(2):
    for lX in xrange(lBaseFragmentLine[lLineIdx][X1], \
                     lBaseFragmentLine[lLineIdx][X2]):
        lPx1 = lBaseFragmentImage.getpixel((lX , \
                                lBaseFragmentLine[lLineIdx][Y2]))
        lPx2 = lCompareFragmentImage.getpixel((lX ,\
                                lCompareFragmentLine[lLineIdx][Y1]))
        lPIMScore += (abs(lPx1[0] - lPx2[0]) + \
                      abs(lPx1[1] - lPx2[1]) + \
                      abs(lPx1[2] - lPx2[2])) / 3
        lPixels += 1
    if lPixels != 0:
        lPIMScore = lPIMScore / lPixels
        #percentage of the possible score
        return (2 ** lBits - lPIMScore) * 100 / \
            (2 ** lBits)

return 0
```

Listing 4.4: Pixel matching algorithm in `CJpegHandler`

Equation 4.2 shows the algorithm in a more formal way. p^{base} refers to the selected pixels of the base image, p^{comp} to the comparative image pixels and n the available amount of pixels. It can be seen that the average difference of those pixels is calculated, resulting in an temporal score. The pixel values are dependent on the color depth of the image and therefore range from zero to $2^{colordepth}$, which is for a 8 bit image 256.

$$score = \frac{\sum_{p=0}^{p=n} \left| p_n^{base} - p_n^{compare} \right|}{n} \qquad \begin{cases} \\ \\ \end{cases} \quad 0 \leq p \leq 2^{colordepth} \tag{4.2}$$

The resulting score is not directly used as a candidate weight because the score is dependent on the color depth of the image. Since different images could have different color depths (e.g. 8bit/16bit), the weights could not be compared to each other. Therefore, an mathematical operation needs to be applied, in order to transform the score to an uniform numbers range. As one can see in Equation 4.3, the score is transformed to a percentage value of the available color depth. Because the score can range from zero to $2^{colordepth}$, it gets subtracted from $2^{colordepth}$ resulting in a very high value for a good match and a very low value for a bad match. Therefore, the candidate weight gives the similarity as percentage value, where a perfect match would be indicate by 100%.

$$candidate \quad weight = \frac{(2^{bits} - score) \cdot 100}{2^{bits}} \tag{4.3}$$

4.3.3 Reassembly algorithm

As we are able to semantically compare image fragments with each other, an efficient reassembly algorithm needs to be found, in order to restore the correct sequence of fragments for a set of header fragments. The sequence of reassembled fragments is called fragmentation path, which results in an image file after a successful completion. Since the reassembly algorithm decides the order of fragment comparisons, it has a major effect on the computational complexity.

For the JPEG file carver, the reassembly algorithm of the multimedia file carver, developed by Poisel, Tjoa and Tavolato is used. It is based on the ”Greedy Parallel Unique Path” (Greedy PUP) algorithm invented by Pal and Memon [23, p. 389]. Because of its graph based nature, the fragments are represented by vertices, which are connected by their comparison weights. The goal of the algorithm is to find as many unique paths in this graph as header fragments exists, while reducing the weights to a minimum.

Figure 4.15 shows a simple example of the algorithm by reassembling two header fragments (H1 and H2) and three non header fragments (F1, F2 and F3). In the first steps (A and B), the fragments are classified and sorted, to bring the header fragments to the beginning of the list. These actions are not done in the reassembly phase, but are shown here to give a better understanding of the process. In the next step (C), each header fragment (Hx) is compared to a non header fragment (Fx). The file type specific comparison function calculates the candidate weights to determine the best match. Since the algorithm is greedy, the best match (F2) is added to the reassembly path of H1, removing it from the list of available fragments. At this point, a new feature is introduced to the algorithm. After assigning a fragment to a fragmentation path, the new fragment is analyzed for file type specific footer

information. If it is a footer fragment, the fragmentation path is assumed to be completed, removing it completely from the reassembly algorithm. The analysis for footer information is highly file type specific and can already be done in the classification phase. The analysis for footer information reduces the overall amount of necessary comparisons because the reassembly path can be removed very early from the reassembly algorithm. This can be seen in step (D) of Figure 4.16, where H1 and F2 are reassembled to ”File 1” and no longer considered in the comparisons. Steps (E) and (F) repeat the previously described actions to reorder the remaining fragments correctly.

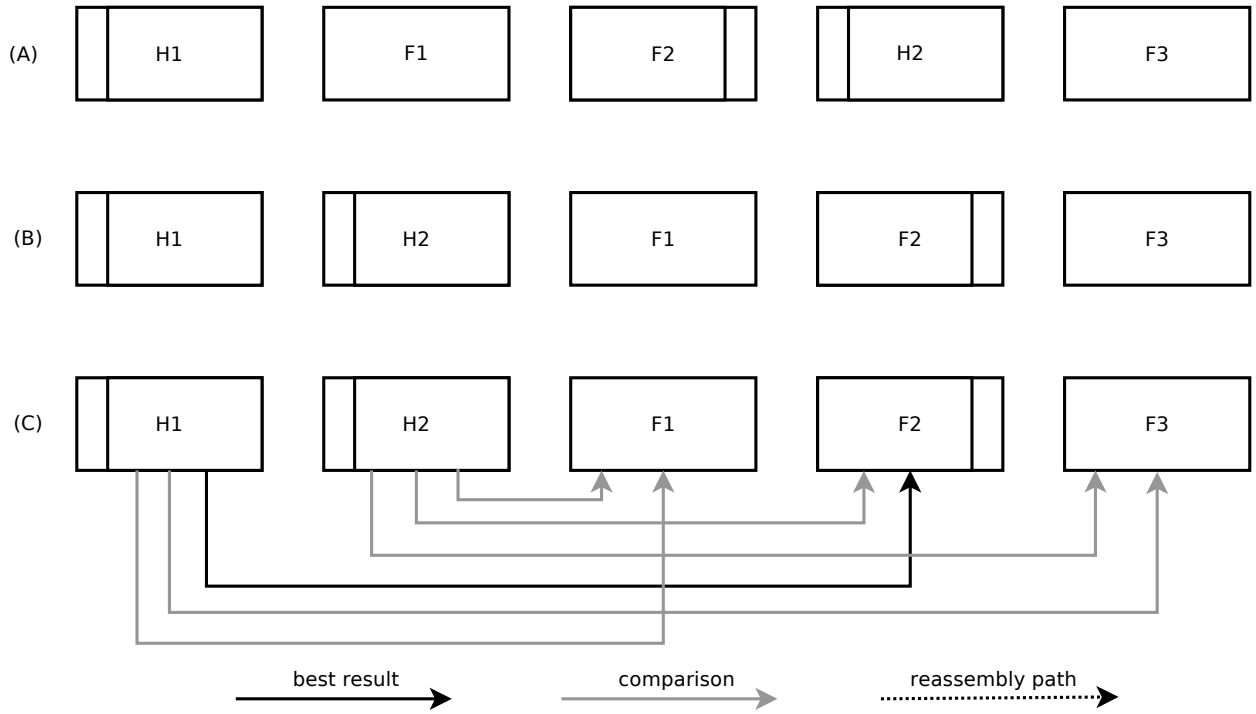


Figure 4.15: Sorting and first comparisons using greedy PUP. Figure is based on Poisel et al. [37, p. 11]

The maximum number of comparisons needed for the reassembly phase can be calculated as shown in Equation 4.4. H refers to the number of header fragments and F to the number of non header fragments. Because header fragments are removed from the reassembly algorithm at completion, this equation describes the worst case scenario, finding not a single footer fragment, leaving all reassembly paths unfinished.

$$Comparisons = \sum_{n=1}^{F} H * n \tag{4.4}$$

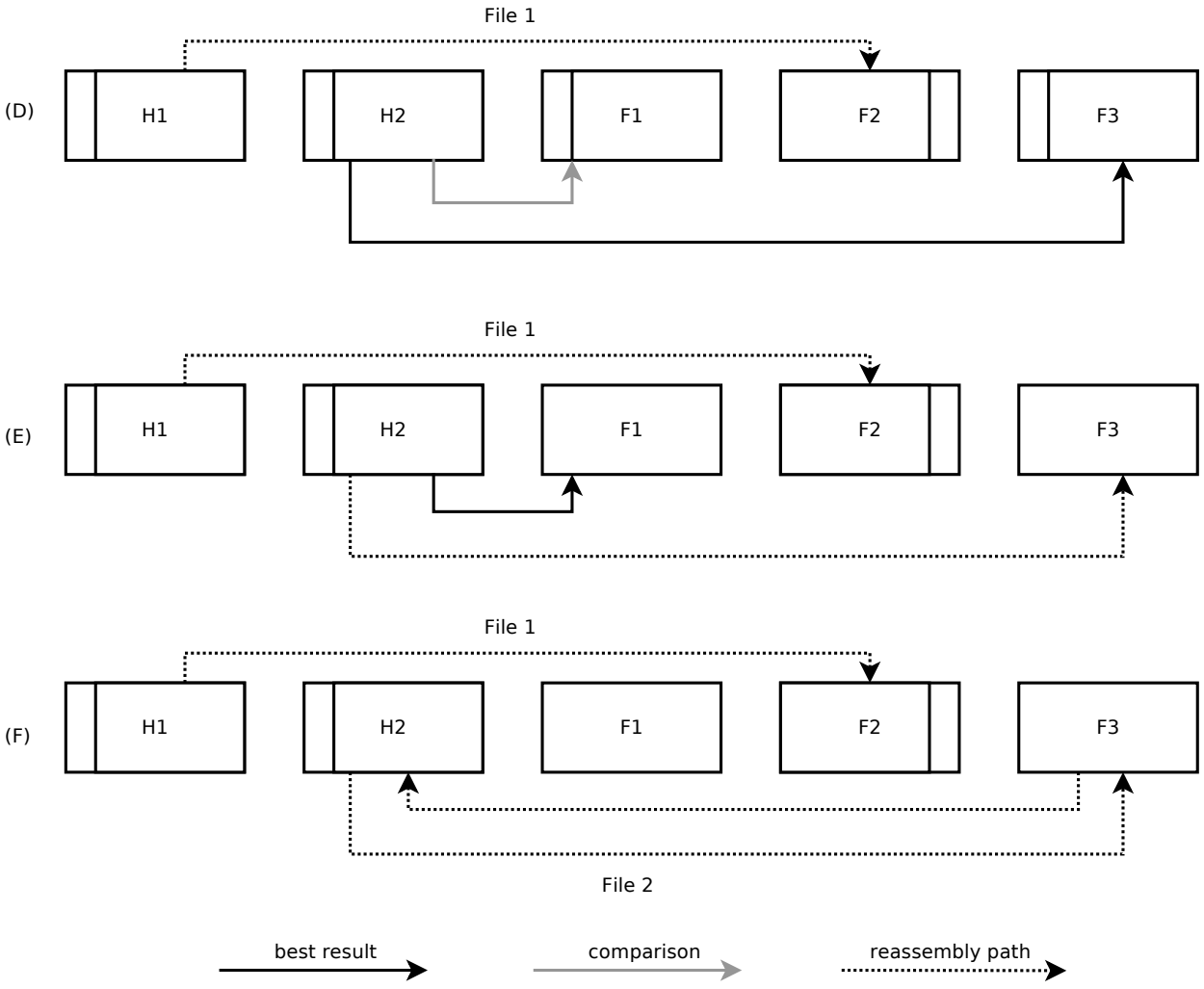


Figure 4.16: Reassembling file fragments using greedy PUP. Figure is based on Poisel et al. [37, p. 11]

As a possible approach to improve the performance of the algorithm, Poisel et al. [37, p. 10] recommended to cache previous comparison results for future use. This would prevent the calculation of the same comparisons between fragments multiple times.

Based on Pal and Memon's [3, p. 67ff] taxonomy, the smartcarving procedure consists of three steps which differ in computational complexity: preprocessing, collation and reassembly. The preprocessing and collation phase has a linear complexity $O(n)$ since the data is sequentially analyzed to

detect file fragments. The sorting algorithm for the reassembly phase also results in linear complexity, since the fragments are only separated by using a binary decision whether the fragment is a header fragment or a non-header fragment. As previously already mentioned, the reassembly algorithm has the biggest effect to the complexity because it needs to compare the fragments with each other, resulting in a quadratic complexity $O(n^2)$. The overall complexity is dominated by the most complex step, resulting in a quadratic complexity like shown in Equation 4.5.

$$Complexity = O(n) + O(n) + O(n^2) \cong O(n^2) \tag{4.5}$$

As usual, the computational complexity approximates the maximum computations assuming a worst case scenario of the algorithm. Of course, the algorithm can finish much faster than $O(n^2)$, if lot of header fragments exist and little non-header fragments are in place. In this case, the image files are possible to be reassembled very quick because of the lack of possible combinations. Since the algorithm removes reassembled fragments, the overall needed comparisons may reduce drastically. However, for the approximation of the overall complexity, the worst case is assumed, resulting in quadratic complexity.

Verification of the effectiveness of the implemented file carver

As the technical details and used carving algorithms were shown, this chapter will analyze the carving results of a defined test data set. This set is randomly created by a self written generator, which writes JPEG and non-JPEG files to a disk image in order to ensure high fragmentation within the test data set. This chapter will first describe the used test data set in detail and analyze the carving results in the second half. The analysis will give an insight in the effectiveness of the file carver and the used algorithms.

5.1 Test data set

The test data set consists of both JPEG and non-JPEG files, to test the file carver's ability to identify data clusters of JPEG files. Additionally, the disk image and its actual file fragmentation will be shown, to be able to compare it to the carving results.

5.1.1 JPEG files

The chosen test set contains five different JPEG files with different features, which are detailed described in this chapter in order to maintain the reproducibility of this testing case. The JPEG files were randomly downloaded from public available sources to get media material which is used "in the wild". As you can see in Table 5.1, this results in different JPEG features which are all needed to test the effectiveness of the JPEG file carver.

Table 5.1 gives a general overview of the JPEG files. The shown pictures are used as reference images, to which the carved files can be compared to later. This is essential for an objective quality measurement of the carved results. During this test, only images with a JFIF application segment were used. File carving of files with EXIF meta-data is very likely to work but application testing was primary focused on JFIF files. The carver supports arbitrary image resolutions, as well as the color types "RGB" and "Grayscale" since these are very common. Another very important attribute of JPEG images is the sampling factor, which describes the size of the pixel block used for the discrete

Picture	File name	Interchange Format	Dimension	Color type	Size (Byte)	Sampling Factor
	balls.jpg	JFIF 1.01	800x600	RGB	146.534	2x2
	nature.jpg	JFIF 1.01	1024x1024	RGB	204.587	2x2
	python.jpg	JFIF 1.01	750x500	Grayscale	74.497	1x1
	space.jpg	JFIF 1.02	1600x1200	RGB	467.491	1x1
	war.jpg	JFIF 1.01	800x600	RGB	127.102	2x2

Table 5.1: List of JPEG images used in the test set

cosine transform. JPEG uses at least pixel blocks of size of 8x8 which corresponds to the sampling factor 1x1. However, other sampling factors like 2x2 (16x16px), 1x2(8x16px) or 2x1(16x8px) can occur. The application supports the very common 1x1 and 2x2 sampling factors. All can be decoded using the sequential mode, meaning that the data of all color channels are iteratively stored, allowing an sequential decoding of the whole image.

5.1.2 Automatic generation of disk images

The creation of the disk image, containing the test data set, was completely automated to ensure completely random data storage. Listing 5.1 shows the bash script used for creating the disk image. The general intent of the script is to generate a NTFS or FAT disk image and write randomly non-JPEG files to it until it is full. Afterwards, the JPEG images specified in Table 5.1 are copied to the disk. To ensure fragmentation, files on the disk are deleted until a JPEG image can be copied.

Line 1 to 24 initialize the script and check for command line parameters `vfat` and `ntfs`. Afterwards (25-39), the disk image is newly generated, or if a template is already existent, copied, and gets formatted with the file system specified as command line parameter. The correctly formatted disk image gets mounted to allow file operations (43-60). As a first step, all non-jpeg files are copied to the disk until the media is full (44). Line 47 randomly iterates through all JPEG images of the test set and tries to copy them to the disk. If no space is available, a randomly chosen non-JPEG file is deleted. If there is afterwards enough disk space, the JPEG file gets copied. This ensures file fragmentation of the image data, because most non-JPEG files are smaller than the JPEG files, remaining small chunks of free disk space after deletion, forcing the file system to fragment the images. This process is repeated until all images are copied to disk. After unmounting the disk, it can be used by the file carver as a testing image. Of course, in a real environment, the file system information like the Master File Table (MFT) could be deleted or defect. Although the recovery of a formatted disk would be perfectly possible, this disk image is not formatted, to have an exact look at the file storage. Otherwise it wouldn't be possible to verify the correctness of the carved data, preventing an interpretation of the test results.

```
1  #!/bin/bash
2
3  USAGE="Usage: diskGenerator.sh [vfat|ntfs]"
4
5  if [ $# -ne 1 ]; then
6     echo $USAGE
7     exit 1
8  fi
9
10 if [ "x$1" != "xvfat" ] && [ "x$1" != "xntfs" ] ; then
11    echo $USAGE
12    exit 1
13 fi
14
15 FS=$1
16 MOUNT="mount"
```

```
17 DISK="disk.img"
18 SIZE="5M"
19
20 #Remove old Image
21 if [ -e $DISK ]; then
22    rm $DISK
23 fi
24
25 #================= DIST GENERATION =====================
26 echo "␣-␣Generating␣$SIZE␣$FS␣Disk"
27 if [ ! -e template.img ]; then
28    dd if=/dev/zero of=template.img bs=1 count=$SIZE
29 fi
30 cp template.img $DISK
31
32 if [ $FS == "vfat" ]; then
33    mkfs -t $FS $DISK
34 else
35    mkfs -t $FS -c 2048 -s 512 -F $DISK
36 fi
37
38 echo "␣-␣Mounting␣$DISK"
39 sudo mount -t $FS -o loop $DISK $MOUNT
40
41
42 #================== COPY ============================
43 echo "␣-␣Copying␣data␣to␣$MOUNT"
44 sudo cp data_$FS/* $MOUNT &> /dev/null
45
46 #try to copy all jpegs
47 for jpeg in $(find jpeg/ -maxdepth 1 -mindepth 1 -type f | sort --random-sort)
48 do
49   #delete files until enough space for jpeg
50   echo "[+]␣try␣to␣copy␣$jpeg␣to␣$MOUNT"
51   for delete in $(find $MOUNT -maxdepth 1 -mindepth 1 -type f | grep -v .jpg | sort --random-sort)
52   do
53     echo "[-]␣need␣to␣delete␣$delete"
54     rm $delete
55     cp $jpeg $MOUNT &> /dev/null
56     if [ $? -eq 0 ]; then
57        break
58     fi
59   done;
60 done;
61
62 echo "␣-␣Unmounting␣$DISK"
63 sudo umount $MOUNT
64
65 exit 0
```

Listing 5.1: Automatic creation of the disk image

5.1.3 Description of the used disk image

This chapter describes the generated disk image, which will be used in this chapter to test the implemented file carver. For a verification of the randomly generated data on the disk, the command `fls` of "The Sleuth Kit" is used. It reads the file system information of the disk without mounting it. Table 5.2 shows the content of the disk "image_ref_jpeg_ntfs_DA.img". It can be seen that all JPEG files are stored at the disk. Additionally to the image files, other files of different file types are stored to the disk in order to enforce file fragmentation and to test the collation algorithm of the carver. A total of 21 files of different file types are stored on the image, involving both plain text file types (SVG and Text) and compressed file types (H.264 and JPEG). The tool `fls` also lists the directory entries of the files which can be used to access the stored files.

File name	File type	Size (Byte)	Directory Entry
Block_Diagram_Delta-Sigma_1.svg	SVG Vector Graphic	86.203	69-128-2
1308163855.svg	SVG Vector Graphic	67.224	67-128-2
1308163855_1.svg	SVG Vector Graphic	67.224	64-128-2
1308163855_3.svg	SVG Vector Graphic	67.224	66-128-2
Block_Diagram_Delta-Sigma_2.svg	SVG Vector Graphic	86.203	70-128-2
brou_fema.txt	text data	113.182	81-128-2
brou_fema_1.txt	text data	113.182	72-128-2
brou_femav_2.txt	text data	113.182	73-128-2
brou_fema_3.txt	text data	113.182	74-128-2
brou_fema_4.txt	text data	113.182	75-128-2
brou_fema_5.txt	text data	113.182	76-128-2
brou_fema_6.txt	text data	113.182	77-128-2
FVDO_Freeway_qcif.h264	H.264 video	143.826	82-128-2
FVDO_Girl_qcif.h264	H.264 video	134.112	85-128-2
FVDO_Girl_qcif_2.h264	H.264 video	134.112	87-128-2
FVDO_Girl_qcif_3.h264	H.264 video	134.112	88-128-2
python.jpg	JPEG Image	74.497	80-128-2
nature.jpg	JPEG Image	204.587	91-128-2
balls.jpg	JPEG Image	146.534	78-128-2
space.jpg	JPEG Image	467.491	84-128-2
war.jpg	JPEG Image	127.102	89-128-2

Table 5.2: Detailed list of all files stored on the disk image

To analyze the results of the file carver, it is important to know the exact fragmentation of the targeted JPEG files. As file system information is still available, the fragment locations of the images

can be identified with the `istat` command. The tool reads the MFT of the disk image and shows all clusters that are assigned to an directory entry. Table 5.3 gives an overview of the JPEG file fragments which are supposed to be identified by the file carver. Since the most images have at least three fragments, they can be called heavily fragmented in respect to their small file size. The image space.jpg has even six fragments. To make things worse, the fragments are partially very close together. Therefore, this test set can be rather seen as a worst case scenario than a best case scenario, to truly test the quality of the file carver.

File name	Fragment 1	Fragment 2	Fragment 3	Fragment 4	Fragment 5	Fragment 6
python.jpg	2417-2453	-	-	-	-	-
nature.jpg	2361-2416	2454-2472	297-319	5-6	-	-
balls.jpg	1262-1278	2544-2558	320-326	249-281	-	-
space.jpg	1196-1261	1084-1139	761-803	561-593	660-674	54-69
war.jpg	149-173	224-248	282-294	-	-	-

Table 5.3: Cluster numbers of all JPEG fragments

5.2 Analysis of the test results

To verify the effectiveness of the developed JPEG file carver, a qualitative analysis of the carving results is necessary. For the test, the multimedia file carver was used in the SVN revision 931. The used data set, as described before, is also available in this revision with the file name ”image_ref_jpeg_ntfs_DA.img”. The default preprocessing and collation parameters were used to evaluate the carver in its default state.

The carving process has two major steps, which mainly influence the carving results: collation and reassembly. For a detailed analysis, the collation phase will be analyzed first. In the current carving state, the file fragments can’t be associated to a JPEG file by the file carver. They are seen by the carver as separated chunks of JPEG data without any contextual information other than the header state. Since we have detailed knowledge of the disk image (see Table 5.3), the correctness of the carved fragments can be verified. By comparing the classification results with the correct cluster numbers in Table 5.4, an overall reliability of 99,34% can be calculated (see appendix A.1), meaning that more than 99% of all available clusters were correctly classified. Additionally, it is possible to match the carved fragments to the correct fragments in order to associate files to them. In Table 5.4 this analysis can be seen, showing the carved file fragment and the files they belong to. It can be seen that fragment 4 and fragment 6 don’t belong only to one file. For example, fragment 4 ranges from cluster 2417 to 2472, but one fragment of python.jpg is stored at 2417 to 2453, followed by a fragment of nature.jpg at 2454 to 2472. Since the carver can’t distinguish between the image data

of two different fragments, it assumes them to be one. It is assumed that this problem can hardly be solved, because the collation algorithm would need to be able to decode the image data in order to distinguish the images. This can only be done by providing the correct compression parameters which are stored in the header that is only available at reassembly time. The same problem occurs at fragment 6, preventing nature.jpg, war.jpg and balls.jpg to recover completely correct in the next phase.

Fragment	Fragment type	Start Cluster	End Cluster	Associated File
Fragment0	Header	149	173	war.jpg
Fragment1	Header	1196	1261	space.jpg
Fragment2	Header	1262	1278	balls.jpg
Fragment3	Header	2361	2416	nature.jpg
Fragment4	Header	2417	2472	python.jpg, nature,jpg
Fragment5	Non-Header	54	80	space.jpg
Fragment6	Non-Header	224	326	war.jpg, balls.jpg, nature.jpg
Fragment7	Non-Header	561	593	space.jpg
Fragment8	Non-Header	660	674	space.jpg
Fragment9	Non-Header	761	803	space.jpg
Fragment10	Non-Header	1084	1139	space.jpg
Fragment11	Non-Header	2544	2558	balls.jpg

Table 5.4: List of all carved fragments

Because most fragments were carved correctly, it is still possible for the file carver to reassembly most parts of the JPEG files. However, it can be assumed that nature.jpg, war.jpg and balls.jpg are partly corrupted due to a hardly preventable worst-case situation. It is assumed that this is an effect of very high fragmentation on a relatively small disk image. As a result of the reassembly phase, the carver creates the following reassembly paths:

- Fragment 0 (war.jpg): 0, 6
- Fragment 1 (space.jpg): 1, 10, 9, 7, 8, 5
- Fragment 2 (balls.jpg): 2, 11
- Fragment 3 (nature.jpg): 3
- Fragment 4 (python.jpg): 4

As a remarkable result, it can be seen that all six fragments of space.jpg are reassembled correctly. Fragment 0, 2 and 3 are only partially reassembled, because of the errors in the collation phase.

Fragment 4 doesn't need any reassembly because the whole file was stored in this header fragment. Table 5.5 illustrates the carving results by comparing the original image file to the header fragment, which would be even recovered by ordinary file carvers, and to the recovered file. By comparing the image data of the header fragment to the recovered image, the benefit of the multimedia file carver can clearly be seen.

Let's take a closer look at each recovered image. The first image, balls.jpg, got recovered by a technique called "object validation". Because of an decoding error at the header fragment, it is impossible to compare it to other fragments in order to make a reassembly decision. However, it was seen that images with an decoding error can mostly only be correctly decoded by reassembling it with its correct following fragment. Therefore, the faulty header fragment was compared to all other fragments, assigning a high candidate weight at successful decoding. Because the header fragment could also be decoded successfully with an incorrect successor fragment, the assigned candidate weight is as high to be an "instant win". It is still possible for other reassembly paths to win with an better match.

The next image, nature.jpg had no possibility to get an fragment because they were all incorrectly assigned to other fragments used in python.jpg or war.jpg. Python.jpg didn't need any reassembly because the file was not fragmented. However, the recovered image is much larger than the original file, because it also contained large chunks of image data of nature.jpg. Due to the End-of-Image JPEG marker, this image data doesn't get decoded and is therefore not seen. The fourth image, space.jpg, is completely correct reassembled. Even though the image has large areas of black, the comparison algorithm was able to re-order the fragments correctly, pointing out the effectiveness of the reassembly algorithm. War.jpg could also be reassembled to let someone semantically interpret the image. The last quarter of the image was also not able to recover because of the merged file fragments.

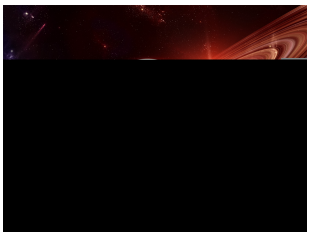
File name	Original Image	Restored Header Fragment	Recovered Image
balls.jpg		n.A.	
nature.jpg			
python.jpg			
space.jpg			
war.jpg			

Table 5.5: Comparison between the original JPEG image and the recovered

Conclusion and outlook

This thesis showed, which algorithms can be used to recover randomly fragmented JPEG files without file system information. Different carving approaches were shown, of which the smart carving architecture is most useful for implementing a file type specific file carver in a modular way. It consists of three highly specialized phases: preprocessing, collation and reassembly. Preprocessing prepares the target data to allow analysis in the next two steps. Afterwards, in the collation phase, the data clusters are analyzed for their containing data type. The reassembly phase groups the identified clusters to fragments and reorders the fragments to retrieve the original files.

For carving fragmented JPEG files, the open-source multimedia file carver [8] was modified. Therefore, the business logic was needed to be extended by many file type specific algorithms. In order to create those algorithms, a deep understanding of encoding and internal storage mechanisms of JPEG files was needed and explained in chapter 3.1. Beside many architectural and GUI based source code modifications, major changes of the file carver were done in its classification and reassembly phase.

The file carver needs to be able to identify raw data clusters as JPEG compressed image data. Different approaches were studied to find the most reliable approach for data classification. Although algorithms like Support Vector Machines (SVM) or Normalized Compression Distance (NCD) offer a generic approach (type-all) for file type identification, Roussev [14, p. 12] and Veenman [15, p. 397] stated that only a file type specific algorithm is able to achieve highly reliable classification results. Due to the unique features of the JPEG-format marker sequences, it is appropriate to develop a file type specific collation algorithm, which bases its classification decision on those features. In addition to positive file type identification, also negative exclusion is implemented by rejecting clusters if character sequences occur, which are not allowed according to the JPEG standard. The shown algorithms result in a highly reliable JPEG classifier which meets the requirements for the JPEG file carver.

The reassembly phase of the multimedia file carver needed major changes to support JPEG reassembly. Parallel Unique Path (PUP) was used as reassembly algorithm, which offers good balance between performance and reassembly quality. In order to be able to reassemble JPEG fragments, they need to be comparable. After discarding the histogram intersection algorithm, pixel matching was used to perform reliable comparisons of different JPEG file fragments. Although the comparison of JPEG image parts looks simple at first glance, a lot of effort was invested to prepare the fragments in a way to make them decodable. This is a difficult task, because file fragments can't be decoded

without, at least partial valid, meta information, which is stored in the header fragments. As a result, it is not possible to compare arbitrary non-header fragment to each other. The only solution of this problem is to compare complete reassembly paths to a fragment, which was done by adapting the algorithms in place.

At the end of the thesis, the developed JPEG file carver was tested and its test results were analyzed to verify the carving reliability. It was shown that the collation algorithm classified 99,34% of the available clusters correct. The false positive rate is at 0,10% and the false negative rate at 2,93% (see appendix A.1). The biggest problem in the collation phase is, that it is not possible to distinguish two JPEG file fragments which have no non JPEG clusters in between. Although this worst case scenario occurred multiple times, it was possible to completely recover a JPEG image with six file fragments in the shown test scenario.

To improve the carving results further, different possibilities were pointed out:

- According to Poisel et al. [37, p. 10], it is recommended to cache previous comparison results of file fragments for future use. This would prevent the calculation of the same comparisons between fragments multiple times and improve the performance.
- Since JPEG stores compressed image data, it needs to be decoded prior to use. Huffman compression could be used to reject file fragments in an early comparison phase. The JPEG header stores a specific Huffman table which can be different for each JPEG file. If two file fragments are compared to each other, the Huffman table of the first fragment can be applied to the second, to test for valid Huffman symbols. If the Huffman symbols are not compatible with each other, the second fragment is very likely no valid successor.
- The classifier should not only check for signatures of header fragments, but also for footer fragments. This would increase the chance of correctly carving file fragments and reduce the risk of merged file fragments.

Appendix

A.1 Calculations

The values used in the following calculations are based on the test data set shown in Table 5.3 and the carving results of Table 5.4.

Correct classified clusters	$Clusters_{correct} = 2541$
Available clusters	$Clusters_{available} = 2558$
JPEG clusters	$Clusters_{jpeg} = 512$
Non-JPEG clusters	$Clusters_{\neg jpeg} = 2046$
False Positives	$Clusters_{FP} = 2$
False Negatives	$Clusters_{FN} = 15$
Classification Reliability	$CR = \frac{Clusters_{correct}}{Clusters_{available}} = \frac{2541}{2558} = 99,34\%$
False Positive Rate	$FPR = \frac{Clusters_{FP}}{Clusters_{\neg jpeg}} = \frac{2}{2046} = 0,10\%$
False Negative Rate	$FNR = \frac{Clusters_{FN}}{Clusters_{jpeg}} = \frac{15}{512} = 2,93\%$

A.2 JPEG specification

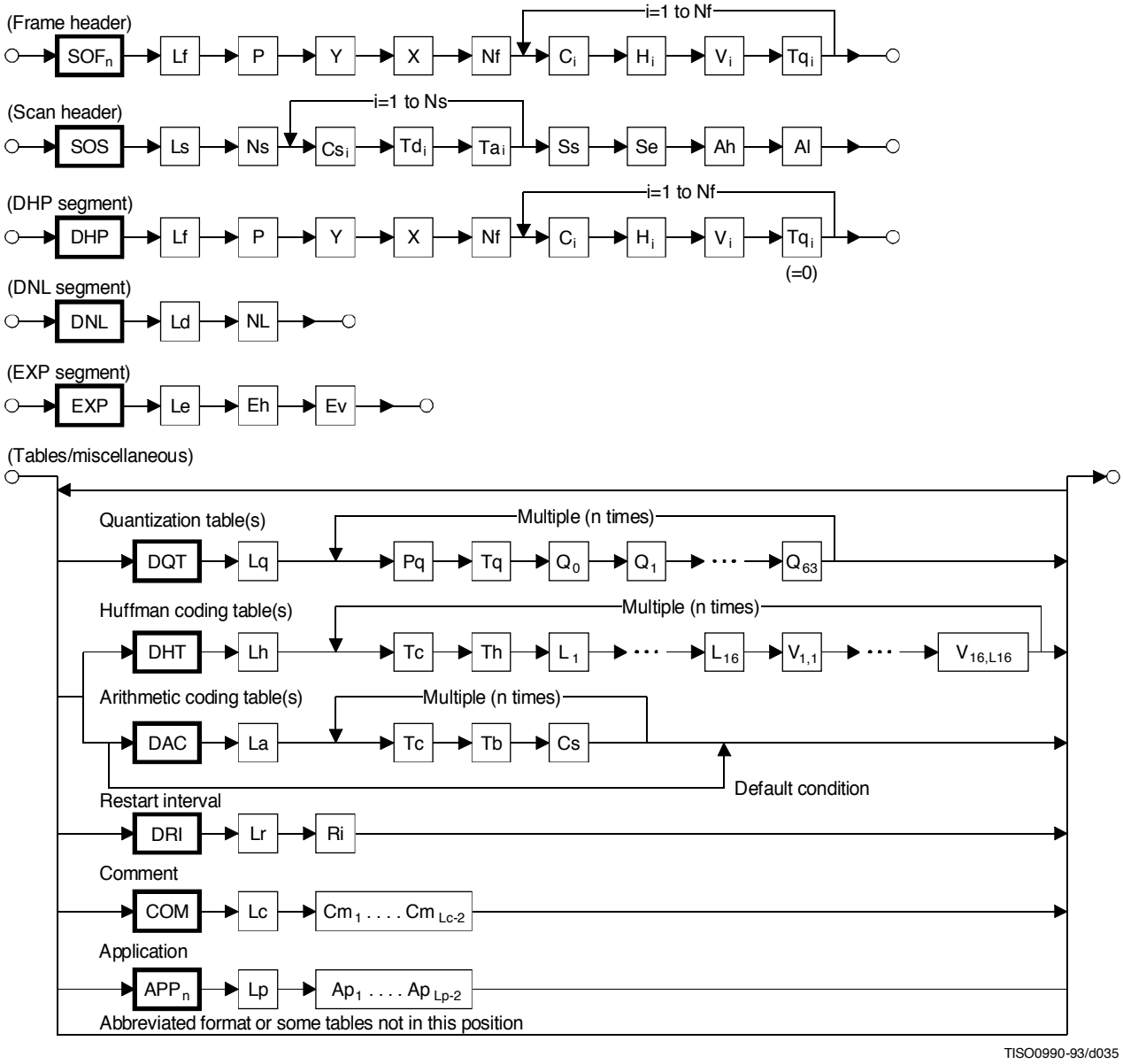


Figure A.1: ”Flow of marker segment” [12, p. 49]

Bibliography

[1] M. Reith, C. Carr, and G. Gunsch, "An examination of digital forensic models," *International Journal of Digital Evidence*, vol. 1, issue 3, 2002.

[2] KPMG, "Data loss barometer," Internet, Nov. 2010, http://www.datalossbarometer.com/docs/KPMG_Data_Loss_Barometer_-_Issue_3_-_November_2010.pdf [Accessed: Feb. 14, 2012].

[3] A. Pal and N. Memon, "The evolution of file carving," *IEEE Signal Processing Magazine*, vol. 26 issue: 2, pp. 59–71, 2009.

[4] K. M. Mohamad, A. Patel, and M. M. Deris, "Carving jpeg images and thumbnails using image pattern matching," in *Proc. IEEE Symp. Computers & Informatics (ISCI)*, 2011, pp. 78–83.

[5] "Sourceforge.net: Foremost 1.5.7," Oct. 2010, http://foremost.sourceforge.net [Accessed: Feb. 15, 2012].

[6] G. G. R. III and V. Roussev, "Scalpel: A frugal, high performance file carver." in *DFRWS*, 2005.

[7] D. Assembly, "Adroit photo forensics 2011," Feb. 2012, http://digital-assembly.com [Accessed: Feb. 14, 2012].

[8] R. Poisel and S. Tjoa, "Multimedia file carver," Nov. 2011, http://www.digitalforensics.at/wordpress/?page_id=162 [Accessed: Feb. 14, 2012].

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

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

[11] O. Avni and T. Knierim, "Carving und semantische analyse in der digitalen forensik," *Seminar: Digital Forensics*, 2010, http://www.halvani.de/math/pdf/(Oren_Halvani)-Carving_und_semantische_Analyse_in_der_digitalen_Forensik.pdf [Accessed: Jun. 29, 2012].

[12] ITU/CCIT, *Digital Compression and Coding of Continuous-Tone still Images T.81*, CCIT Std., Sept. 1992, http://www.w3.org/Graphics/JPEG/itu-t81.pdf [Accessed: Feb. 14, 2012].

[13] M. Karresand and N. Shahmehri, "File type identification of data fragments by their binary structure," in *Proc. IEEE Information Assurance Workshop*, 2006, pp. 140–147.

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

[15] C. J. Veenman, "Statistical disk cluster classification for file carving," in *Proc. Third Int. Symp. Information Assurance and Security IAS 2007*, 2007, pp. 393–398.

[16] S. Axelsson, "Using normalized compression distance for classifying file fragments," in *Proc. ARES '10 Int Availability, Reliability, and Security Conf*, 2010, pp. 641–646.

[17] M. M. Shannon, "Forensic relative strength scoring: Ascii and entropy scoring," *International Journal of Digital Evidence*, vol. 2(4), 2004.

[18] M. McDaniel and M. H. Heydari, "Content based file type detection algorithms," in *Proc. 36th Annual Hawaii Int System Sciences Conf*, 2003.

[19] G. Conti, S. Bratus, A. Shubina, A. Lichtenberg, R. Ragsdale, R. Perez-Alemany, B. Sangster, and M. Supan, "A visual study of primitive binary fragment types," *Black Hat USA*, 2010, http://www.rumint.org/gregconti/publications/taxonomy-bh.pdf [Accessed: Mar. 06, 2012].

[20] W. C. Calhoun and D. Coles, "Predicting the types of file fragments," in *Digital Investigation, 5, Supplement(0): S14 S20. Proceedings of the Eighth Annual DFRWS Conference*, 2008.

[21] E. Osuna, R. Freund, and F. Girosi, "Training support vector machines: an application to face detection," *IEEE Conference on Computer Vision and Pattern Recognition*, p. 130, 1997.

[22] C.-W. Hsu, C.-C. Chang, and C.-J. Lin, "A practical guide to support vector classification," *National Taiwan University, Department of Computer Science*, 2003, http://www.csie.ntu.edu.tw/~cjlin/papers/guide/guide.pdf [Accessed: Jun. 13, 2012].

[23] N. Memon and A. Pal, "Automated reassembly of file fragmented images using greedy algorithms," *Trans. Img. Proc.*, vol. 15, no. 2, pp. 385–393, Feb. 2006.

[24] A. Pal, "Automated reassembly of file fragmented images using greedy algorithms," Master's thesis, Polytechnic University, 2005, http://digital-assembly.com/technology/research/pubs/pal-msthesis.pdf [Accessed: Feb. 14, 2012].

[25] S. Kloet, "Measuring and improving the quality of file carving methods," Master's thesis, Eindhoven University of Technology Department of Mathematics and Computer Science, 2007, http://www.forensicswiki.org/w/images/b/b9/Kloet_2007.pdf [Accessed: Jun. 18, 2012].

[26] N. Mikus, "An analysis of disc carving techniques," Master's thesis, Naval Postgraduate School, 2005, http://cisr.nps.edu/downloads/theses/05thesis_mikus.pdf [Accessed: Jun. 26, 2012].

[27] "Data lifter forensic software," http://www.datalifter.com/products.htm [Accessed: Jun. 26, 2012].

[28] "Encase forensic," http://www.guidancesoftware.com/encase-forensic.htm [Accessed: Jun. 26, 2012].

[29] "Forensic toolkit (ftk)," http://accessdata.com/products/computer-forensics/ftk [Accessed: Jun. 26, 2012].

[30] "Nfi defraser," http://defraser.sourceforge.net/ [Accessed: Jun. 26, 2012].

[31] "Photorec," http://www.cgsecurity.org/wiki/PhotoRec [Accessed: Jun. 26, 2012].

[32] "Recover my files," http://www.recovermyfiles.com/ [Accessed: Jun. 26, 2012].

[33] B. Carrier, V. Wietse, and C. Eoghan, "Forensic challenge 2006," http://www.dfrws.org/2006/challenge [Accessed: Feb. 14, 2012].

[34] N. Freed and N. Borenstein, "Multipurpose internet mail extensions (mime) part two: Media types," Internet, Nov. 1996, http://www.ietf.org/rfc/rfc2046.txt [Accessed: Apr. 13, 2012].

[35] E. Hamilton, *JPEG File Interchange Format Version 1.02*, C-Cube Microsystem Std., Sept. 1992, http://www.jpeg.org/public/jfif.pdf [Accessed: Feb. 14, 2012].

[36] JEIDA, *Digital Still Camera Image File Format Standard (EXIF) Version 2.1*, Japan Electronic Industry Development Association (JEIDA) Std., June 1998, http://www.exif.org/Exif2-1.PDF [Accessed: Feb. 14, 2012].

[37] R. Poisel and S. Tjoa, "Roadmap to approaches for carving of fragmented multimedia files," in *Proc. Sixth Int Availability, Reliability and Security (ARES) Conf*, 2011, pp. 752–757.

[38] ——, "Feasibility study multimedia file carving," St. Poelten University of Applied Science, Tech. Rep., 2011.

List of Figures

List of Tables

Listings