Eindhoven University of Technology

Department of Mathematics and Computer Science

Master's Thesis

Measuring and Improving the Quality of File Carving Methods

by

S.J.J. Kloet

Supervisor: Prof. Dr. W.J. Fokkink

Almere, October 29, 2007

Preface

Ever since I locked myself into my room as a toddler by disassembling the doorknob, I have been interested in security and how things work. This interest is what led me to visit a lecture by Robert-Jan Mora and Marcel Westerhoud of Hoffmann Forensic, which I thought would be about the recovery of deleted files. Even though the lecture was about completely different topics than I had expected, they had very much managed to gain my interest. One thing led to another and about five months later I started my master's project at Hoffmann about. . . the recovery of deleted files.

These last eight and a half months have been a complete roller coaster ride, with the goal of the project being expanded after three months, participation in an international file carving challenge and a complete thesis overhaul four weeks before the end, but it was well worth it. This has become more than just a master's project, it has become something that I will continue working on long after I have graduated.

I would like to thank a whole list of people that have helped me to get where I am today.

First of all my parents and stepparents, who have supported me in all my years of studying, even when I switched studies after three years.

Wan Fokkink, Robert-Jan Mora and Marcel Westerhoud for their guidance and support throughout this project. Many, many thanks to Joachim Metz for his invaluable guidance and advice on both my thesis and the project itself.

I'd also like to thank my friends at Spacelabs, without our combined study efforts I would *never* have passed each examination of my master on the first attempt. Special thanks to Paul van Tilburg, for all his help on studying, Linux, LaTeX, but most of all for being a great friend.

Last, but certainly not least, I'd like to thank my girlfriend Henrieke, who was my "rots in de branding", especially during the last stressful weeks. And who forced me to relax when I truly needed it but refused to admit it to myself.

Bas Kloet

Almere, October 29, 2007

Summary

Recovering deleted files plays an important role in a digital forensic investigation. One of the methods that can be used to recover these deleted files is file carving. File carving works by extracting files out of raw data, based on file format specific characteristics present in that data. There are a number of tools that can perform file carving, based on different techniques, but until now the quality of these tools and techniques was unclear.

This thesis describes a quality method that was developed to measure the quality of a tool or technique based on the results it produces. Based on the results of these measurements on current carving tools, a number of areas were identified that could be improved. A new carving framework was developed to address these points of improvements, and its results were tested using the previously developed quality method. The new carving framework achieved significantly better results on all identified improvement areas.

Contents

1 Introduction **1**

1.1 File recovery . . . 1

1.2 Current state of carving . . . 2

2 Project goal **3**

2.1 Goal . . . 3

2.2 Thesis layout . . . 5

3 Influence of datasets and carving techniques **7**

3.1 Datasets . . . 7

3.1.1 Fragmented files . . . 8

3.2 Carving techniques . . . 9

4 Carving quality criteria **15**

4.1 Quality criteria measures . . . 17

4.2 Normative scores . . . 20

5 How to measure carving result quality **23**

5.1 Datasets . . . 23

5.2 Testing procedures . . . 24

5.3 Score interpretation . . . 24

6 Quality of current tools **27**

6.1 Tested tools . . . 27

6.2 Scores on the different datasets . . . 27

6.3 Overall conclusions . . . 30

7 How to improve carving quality **31**

7.1 Specific improvement goals . . . 31

7.2 Improvement approach . . . 32

8 MultiCarve 35
8.1 Carving method handler 35
8.2 PNG file structure 37
8.3 File structure based carving 39
8.3.1 Simple file format carving 40
8.3.2 PNG file definition 42
8.4 Does MultiCarve meet its goals? 44

9 Fragmentation handling algorithm 47
9.1 Specific problems to solve 48
9.2 Scenarios 48
9.2.1 Complete file and fragmentation 49
9.2.2 Intertwined files 49
9.2.3 Partial file and fragmentation 50
9.3 Algorithm description 50
9.3.1 High-level description of the fragmentation handling algorithm 50
9.3.2 Block handling algorithm 52
9.3.3 Scenario handling makeup 56
9.3.4 Complete file and fragmentation 57
9.3.5 Intertwined files 58
9.3.6 Partial file and fragmentation 59
9.4 Algorithm weak points and improvements 60
9.4.1 Algorithm weak points 60
9.4.2 Incorrect assumptions 61
9.4.3 Revit07 solutions to the different weaknesses 62

10 Revit07 63
10.1 Revit07 and MultiCarve 64
10.1.1 Content carving techniques 64
10.1.2 Detecting corrupted files 65
10.2 Improvement goals handled by Revit07 65

11 Validator 67
11.1 Validator framework 68
11.1.1 Validator problems 69
11.2 Validation examples 70

12 Quality of the carving framework results 71
12.1 Carving framework configurations 71
12.2 Scores on the different datasets 72
12.3 Does the framework meet the improvement goals? 74
12.3.1 Higher carving_recall 74
12.3.2 Higher carving_precision 75

13 Supporting tools and dataset analysis **77**
13.1 Block content analysing tool 77
13.2 Dataset topology 78

14 Conclusion **79**
14.1 Measuring the quality of carving results 79
14.2 Improving the quality of carving results 80

15 Reflection **81**

16 Future work **83**
16.1 Fundamental areas of improvement 83
16.2 Improvements to the quality testing method 84
16.3 Improvements to the carving framework 84

References **87**

A Tool comparison tables **89**

B Tested carving tools **93**
B.1 Foremost 93
B.2 Forensic Toolkit (FTK) 94
B.3 PhotoRec 94
B.4 Recover My Files 94
B.5 Scalpel 95

C Validator tools **97**
C.1 Tool descriptions 97

Chapter 1

Introduction

Digital or computer forensics is the practice of identifying, preserving, extracting, analysing and presenting legally sound evidence from digital media such as computer hard drives[1]. In the past ten years digital forensics has changed from a technique which was almost solely used in law enforcement, to an invaluable tool for detecting and solving corporate fraud. It is a very broad field, of which this master's project handles a small but important part, namely the recovery of deleted files. The following section describes this role of file recovery in a forensic setting.

1.1 File recovery

During a digital forensic investigation many different pieces of data are preserved for investigation, of which bit-copy images of hard drives are the most common. These images contain the data allocated to files as well as the unallocated data. The unallocated data may still contain information that is relevant to an investigation, in the form of (parts of) intentionally deleted files or automatically removed temporary files. Unfortunately, this data is not always easily accessible: a string search on the raw data might recover (parts of) interesting text documents, but it won't help to get to information present in for example images or compressed files. Besides that, the exact strings to look for may not be known beforehand. To get to this information, the deleted files have to be recovered.

There are multiple ways to recover files from the unallocated space. Most techniques use information from the file system to locate and recover deleted files. The advantage of this approach is that it is relatively fast and that meta-information, like the last access date, can often be recovered as well. The downside of this approach is that these techniques become much less effective if the file system information is corrupted or overwritten. In these cases a technique is required that works independent of the file system information, by identifying the deleted files and file parts directly in the raw data and extracting them in a verifiable manner. For more information on file systems in a forensic context, "File System Forensic Analysis" by Brian Carrier[3] is highly recommended.

[1]Based on the definition from `http://www.forensicswiki.org/wiki/Computer_forensics`

There are many tools that are capable of recovering files based on file system information, but tools that get the files directly from the raw data are still rather rare. The following section describes the current state of a technique, known as (file) carving, which works according to the “directly from the raw data” principle.

1.2 Current state of carving

Carving is a general term for extracting files out of raw data, based on file format specific characteristics present in that data. Moreover, carving *only* uses the information in the raw data, *not* the filesystem information.

As Nicholas Mikus wrote on page 1 of his 2005 master’s thesis [7]:

> Disc carving is an essential aspect of Computer Forensics and is an area that has been somewhat neglected in the development of new forensic tools

In the two years since this thesis the field of carving has evolved considerably, but there still are many possible areas of improvement. Most notably there are few different carving techniques, no (objective) method of rating and comparing different carvers, little scientific information on carving and the results of carving tools can also be improved a lot.

This means that this field provides multiple possibilities for projects that combine scientific research into fundamental carving issues with practical improvements of carving tools.

Chapter 2

Project goal

In 2006 the Digital Forensics Research Workshop (DFRWS) issued a challenge to digital forensic researchers worldwide to:
"... design and develop file carving algorithms that identify more files and reduce the number of False positives."[1] Nine teams took up this challenge, with one of these teams consisting of Joachim Metz and Robert-Jan Mora of Hoffmann Investigations.

The final results of this challenge, and its winners, caused some discussion over how a carving tool should be rated.

First of all the winning team used manual techniques to recover the deleted files [2], which, as Metz and Mora stated, does not scale for realistic data sizes. The DFRWS acknowledged this and in 2007 issued another carving challenge[2], in which competing tools had to be fully automated.

Another discussion was that there was no clear description available beforehand of how the tools would be rated and that the final rating was still not fully explained.

This ultimately led to the first part of the goal of this project, to create an objective way to measure the quality of carving tools and techniques.

The quality of a method is a measure how well that method meets its goals and since the main goal of a carver is to recover files, the quality must be measured on how well it recovers these files. The problem is that there are many different possible criteria against which this quality can be rated. Therefore the next section explains the criteria that investigators at Hoffmann have set for a good carving tool and describes the way in which these criteria are translated to an overall goal of carving quality.

2.1 Goal

In order to have an objective way to rate the quality of carving tools and techniques, the digital forensics investigators at Hoffmann want a quality rating based on the results produced by the different tools. The reasoning being that

[1]`http://dfrws.org/2006/challenge/`
[2]`http://dfrws.org/2007/challenge/`

in the end the recovered results determine which deleted information is available for analysis in a forensic investigation.

The results of a tool can have a significant impact on the information available to an investigator. If a tool produces good results, then valuable information might be uncovered. However tool results can also have a negative impact on the usability of the available information.

First of all information that is not recovered by a tool is information that will most likely be ignored, since datasets under investigation are much too large for manual inspection. Therefore it is important that a tool retrieves as much useful information as possible. Any information missed is considered critical.

Unfortunately uncovering as much information as possible can lead to other problems. If many erroneous results are produced, like unviewable images or unreadable documents, then an investigator has to manually sift through these false results. Not only does this take a lot of time, but evidence may be overlooked or (the results of) a tool may be discarded altogether.

The final thing to note is that a tool must be reliable. If a tool is unable to carve a file because it does not support its file type, then this is bad, but at least an investigator *knows* that files of that type will not be recovered. Files that are officially supported, but which are not recovered, may lead an investigator to believe that these files are not present in a dataset. Therefore in this context reliability means that a tool actually recovers the files it claims to support.

All in all the quality of a tool in a practical situation depends, for the most part, on the quality of its results.

Of course, there are also disadvantages when rating a tool based on its results.

The biggest problem is that results are a combination of both the tool and the data being examined. A tool may perform very well on certain datasets, but perform very badly on others. One of the main reasons for this is that some tools can handle fragmentation better than others, but the specific files in a dataset can also have a big impact. Therefore tool quality should always be tested on multiple datasets and resulting scores should be interpreted with care.

Another problem is that some quality aspects are simply unrelated to the carving results, like the amount of human intervention needed to process a dataset or the speed of a carver, i.e. the amount of time needed to investigate a specific amount of data.

The first of these, the amount of human interaction required, is kept outside of the scope of this project by making the following decision:
A tool is only considered for quality testing if it can carve files out of a dataset without intermediate human intervention.

The speed of a carver is a different matter. The average size of hard disks examined in an investigation at Hoffmann is between 80 and 100 Gb, with usually more than one hard disk per investigation. For a carving tool to be usable in a situation like this it must be able to handle these amounts of data in an acceptable time. Together with Joachim Metz of Hoffmann the following rule of thumb was established for the acceptable speed of a carver:
A carver should be able to process roughly 100 Gb of unallocated data per day, i.e. around 1.16 MB per second, to be considered usable in practice. Anywhere between 50 and 100 Gb per day, i.e. between 0.58 and 1.16 MB per second, is considered usable for testing. Tools that can handle less than 0.58 MB of

unallocated data per second are considered unusable. These figures are meant as a guide and the tool is assumed to be running on a modern workstation[3].

The chapters 3 to 6 describe the development of a quality testing method and its results for a number of commonly used tools. Based on these results the decision was made to try to create a tool which produces better results than the tools that were tested. This leads to the following dual goal:

"Define meaningful criteria and a method to measure the quality of carving tools. Based on the quality of current tools, develop a carving tool which achieves better results."

2.2 Thesis layout

This thesis is divided into three parts, based on the project goal.

Chapter 3 to 6 focus on the first part of the goal and describe the creation of a measurement method for tool quality.

Chapter 7 to 12 describe the development of a carving framework that was created to fulfill the second part of the project goal, i.e. to achieve better carving results.

The final part of this thesis consists of a description of two supporting tools, as well as the overall conclusion, reflection on the project and a list of possible future projects.

[3]In this case a Dell notebook with an Intel Core2 2x1.66GHz CPU and 2048 MB of RAM

Chapter 3

Influence of datasets and carving techniques

In order to determine the quality of carving results, one first has to investigate *why* different tools produce different results for the same dataset. To do this the following section describes the makeup of datasets used in file carving and identifies different situations that may negatively effect a carver's ability to extract deleted files. Section 3.2 then looks at the techniques used by current tools. For each technique a description is given of how the difficulties that may be present in a dataset are handled and what impact this has on the results.

3.1 Datasets

The datasets that are examined in a digital forensic investigation are usually bit-copies of full hard drives or individual partitions. The data present on these original drives continually changed over time: files were added, deleted, copied or moved, a partition may have been defragmented, formatted or even resized, etc.

Traces of this process can be seen in the unallocated data, which may contain full, contiguous deleted files, but also partially overwritten and fragmented deleted files.

Since carving works by recovering deleted files from the unallocated data, a carving tool needs to be able to handle the problems that arise due to the presence of fragmented and partial files.

In the rest of this thesis the datasets are considered to be completely made up of unallocated data, which means that a file in such a dataset is always a deleted file.

The following subsections succinctly describe the specific problems caused by fragmented and partially overwritten files, whereas section 3.2 describes how different carving techniques (try to) overcome these problems.

3.1.1 Fragmented files

A fragmented file is a file that has been split into multiple parts and where all parts may be placed on different locations in a dataset. On page four of [4], Garfinkel states that modern operating systems try to write files without fragmentation, but that there are three conditions under which fragmentation still occurs:

1. There are no contiguous regions of free space on the media large enough to contain the complete file, in which case it is split into two or more fragments.
2. If data is appended to an existing file, there may not be enough space available directly after that file and the data will have to be placed elsewhere.
3. Some file systems itself may simply not support writing files of a certain size in a contiguous manner.

Fragmented files can be divided into two categories:

1. files with linear fragmentation
2. files with nonlinear fragmentation

Linear fragmentation occurs when a file has been split into two or more parts, but the parts are present in the dataset in their original order. An example can be seen in figure 3.1, where there are two files, of which F1 is split into two fragments (F1(1/2) and F1(2/2)).

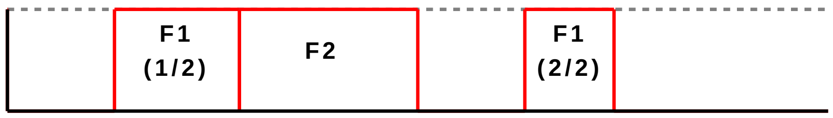


Figure 3.1: Linear fragmentation example

There is unfortunately no guarantee that fragmentation is always linear, it is also possible that the different parts exist in the dataset in a different order than in the original file. An example of this can be seen in figure 3.2, where F1 is again fragmented into two parts, but these parts are in the dataset in reverse order.

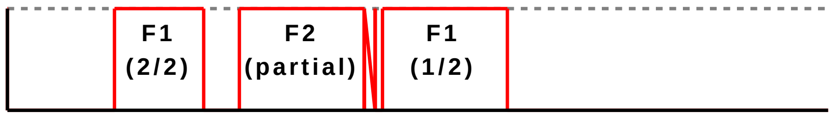


Figure 3.2: Non-linear fragmentation example

Partial files

Besides nonlinear fragmentation, figure 3.2 also shows a partially overwritten file (F2). Partially overwritten or partial files can almost never be fully recovered[1],

[1]In some cases partial files can be repaired, but this is beyond the scope of this project

but may still contain useful information. For a carver there is no real difference between a (fragmented) partial file and a fragmented file for which it has not yet located all parts. At some point a carving algorithm will have to decide that F2 is a partial file. How long a carver leaves both options open is a tradeoff between the amount of fragmented files that may be fully recovered and the time and memory needed to do these checks.

3.2 Carving techniques

This section describes the different carving techniques that are used by the open-source tools tested in chapter 6 and/or were used in the 2006 DFRWS carving challenge. The strengths and weaknesses of the different techniques are discussed in relation to fragmented and partial files. The assumption is that the closed-source tools use techniques that are comparable to those used by open-source tools.

These techniques are:

- Header-footer or header-“maximum file size” carving
- File structure based carving
- Content based carving

The following paragraphs give a concise description of each technique and explain how the technique handles complete, partial and fragmented files. Each paragraph also describes the problems that may occur if the technique handles these files incorrectly.

Figure 3.3 shows a very simple image of the structure of a PNG file, which is used to help in these explanations. PNG files are used as examples throughout this thesis to explain different carving aspects, since PNG files have a very clear and structured file format.

Header-footer or header-“maximum file size” carving

Header-footer carving is the most basic carving technique. It works by searching the dataset for the patterns that mark the beginning of a file (header), like `x89PNGx0Dx0Ax1Ax0A`[2] for PNG files and then looks for the first occurrence of a corresponding end marker or footer (`IENDxAEx42x60x82`). The data between the header and the footer is carved as a file.

Header-footer carving has a number of major problems.

First of all, if the header and/or footer markers are short, then these also occur throughout the data at points where there is no file start or end. PNG headers and footers are relatively long, but JPEG headers and footers for example are only 2 bytes (`xFFxD8` and `xFFxD9`). Since the carver does not check any additional characteristics, these are treated as valid markers. This means that this method produces many results which are not present as files in the original dataset. These are referred to as *False positives* and, as chapter 6 shows, can be extremely numerous with this technique.

[2]Hexadecimal values are represented by x##

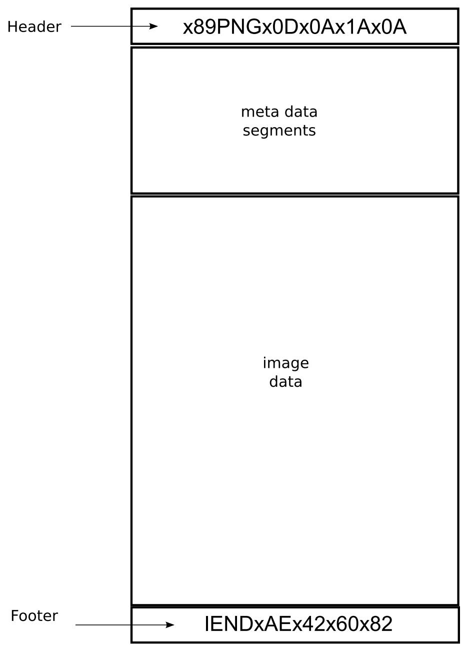


Figure 3.3: Basic PNG file structure

Another problem is that this technique cannot handle fragmentation and partial files, it simply has no way of detecting this. It may even match the start of one file to the end of another.

The last problem is that some files cannot be carved at all, since they do not have fixed headers. This is the case for most plain-text file formats, like text documents and html files.

Even though most file types have a unique header, not all file types have a fixed footer. In the case of header-"maximum file size" carving a maximum file size is defined for these file types. If a header is found, then a piece of data is carved of "maximum file size" length. Note that this maximum file size is usually an "educated guess" and not based on the file format definition.

Header-"maximum file size" carving suffers from the same problems as header-footer carving, but has two additional problems.

First and foremost, this technique will almost always return results that are much larger than the original file. It is left to the investigator to manually locate the correct end of the file in the carved result and to discard the additional data. This is extremely time consuming.

The second problem is that, since the maximum file size is usually an educated guess, it may sometimes be too *small*. Therefore even full, contiguous files may be carved incorrectly, if they are larger than the defined maximum file size.

File structure based carving

File structure based carving addresses the problems that header-footer carving has by using much more information from a file than just the header and footer. To understand file structure based carving, it is best to first get a basic idea of the type of structures that can be present in a file. A small PNG file is used as an example, since the PNG file format is relatively simple and well structured.

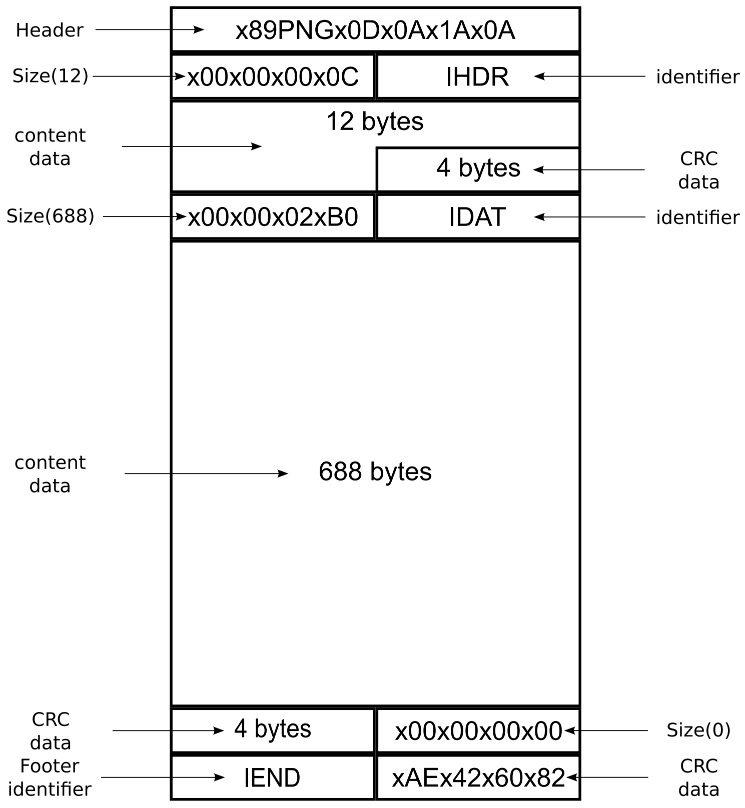


Figure 3.4: PNG file structure example

Figure 3.4 shows the file structure of a very small PNG file:

1. The PNG file starts with a header byte string.
2. The next piece is a 4 byte string which states the size of the next section. It is a big-endian hexadecimal string which in this case calculates to 12 bytes ($0 * 4096 + 0 * 256 + 0 * 16 + 12$).
3. “IHDR” is the identifier of this next section.
4. The 12 bytes that make up the “IHDR” section have no fixed structure elements and are therefore seen as unstructured data.
5. Then there are 4 bytes of CRC[3] data over the “IHDR” section.
6. Then next 4 byte string declares the size of the “IDAT” section, in this case 688 bytes.
7. “IDAT” is the identifier of the next section, which is the section that contains the actual image data.

[3]`http://en.wikipedia.org/wiki/Cyclic_redundancy_check`

8. The next 688 bytes are image data.
9. Then there are 4 bytes of CRC data of the "IDAT" section.
10. Finally there are the size, identifier and CRC of the "IEND" section, which is always the final section of a PNG file.

File structure based carving uses the internal layout of a file to determine which data is part of which file. In the above example these are elements like the header, footer and identifier strings, but also the size information which indicates where these strings can be found.

This layout information is derived from file format specifications, which describe the official makeup of a file format. These definitions vary greatly in availability and precision.

If a file has not been fragmented and its file structure data is fully intact, then this technique produces extremely good results. It can even be used to detect and handle many cases of corruption and fragmentation, if the file structure data is detailed and extensive enough.

If a file structure based carver detects a partial file or a fragmented file it cannot reconstruct, then it can deal with this in a number of ways.

1. It can discard the result, but this might lead to a file in the dataset which is not carved. This is called a *False negative*.
2. It can simply carve the part(s) of the file it has recognised. If this is done without some indication that the file is incorrect, then we have a False positive, just like with header-footer carving. In this case however, the carver knows that the file is incomplete and can mark it as such. This type of result is known as a *Known false positive*.

Unfortunately not all fragmentation or file corruption can be detected using file structure based carving. The main situation which it cannot handle is when a file is fragmented or partially overwritten at a position at which little to no file structure is present. An example of a part of a file with very little file structure is the image data in a PNG file. This means that this technique by itself can not be used to handle all fragmented or partial files.

Block content based carving

One technique that can be useful in detecting fragmentation in those cases where file structure based carving is unsuccessful, is block content based carving.

To explain block *content* based carving, it is best to first explain block based carving. Block based carving is based on the principle that physical media like hard disks write their data in sectors (blocks of 512 bytes), which means that fragmentation will only occur on these sector boundaries. A block based carving approach uses this knowledge by checking each block of data to see if it is part of a file. It is possible to do this for file structure based carving, as is explained in detail in chapter 8.3, but it also allows for block content based carving.

Block content based carving works by calculating meta information like character counts or statistical information over the bytes in a block.

A basic example is to determine the different character types that are part of a block. If half of the bytes in a block are text characters and the other half are zeros, then that block almost certainly consists of Unicode text. The most common way to do this is to use the c-type library[4], which provides different methods to perform this character typing.

The winners of the 2006 carving challenge used block content carving techniques that were based on a statistical approach[5]. One of the techniques that they used was to calculate the information entropy of a block. The wikipedia page on information entropy[6] states that: "The entropy rate of a data source means the average number of bits per symbol needed to encode it". In practice this means that compressed data has a higher entropy rate than for example plain text data.

How can this knowledge be used for file carving?

Take for example the unstructured image data in the PNG example, but with a much bigger image data section, say 6880 bytes. This means that there are about 13 blocks during which a file structure based carver will not be able to detect fragmentation. However, each of the blocks contains roughly the same type of (compressed) data, which means that the entropy of each of these blocks usually stays within certain limits. If there is a sudden change in entropy, then this is a strong indication that the block with this different entropy is not part of the PNG image data.

This technique works well in combination with manual inspection, but by itself is not always precise enough to guide an automated carving tool. However, it is strong in areas where a file structure based technique would have too little structure to work with, like compressed data in a PNG or zip file. This means that combining these two techniques might create (much) better carving results than either of them can provide on their own.

The main problem with this technique lies in finding the calculations which can distinguish between a block of data that belongs in a file and one that does not.

[4]`http://www-ccs.ucsd.edu/c/ctype.html`

[5]`http://sandbox.dfrws.org/2006/bair/README.ANSWERS`

[6]`http://en.wikipedia.org/wiki/Information_entropy`

Chapter 4

Carving quality criteria

The datasets used in chapter 6 to test the different carvers all have an accompanying description of the layout of the files in the dataset. This information usually consists of a list of files, the (block) ranges they occupy in the dataset and their MD5 sums. The MD5 sum is a 32-character hexadecimal number which is calculated using a cryptographic hash function. This MD5 sum can be used to uniquely identify a particular file.

As the previous chapter showed, the combination of complete, partial and fragmented files and different carving techniques can lead to four different result types. To recapitulate:

Positive A file that is correctly carved from the dataset is called a *Positive*.

False positive A carving result which is not a Positive.

Known false positive A carving result of which the carver knows that it is not fully correct, and which it has marked as such[1].

False negative A file that is present in the dataset, but which was not carved.

To determine the type of each carving result, it needs to be compared to the layout information of the dataset it was carved from. Note that this means that the results are directly dependent on the quality of the layout description. Errors in the layout description should therefore be avoided at all cost, but this is the responsibility of the creators of these datasets.

The four result types require more specification, before they can be used as the basis for actual quality measures. Therefore each result type is explained in more detail in the following paragraphs.

Positive

The easiest way to check if a carving result is a Positive is to match its MD5 sum with the MD5 sums in the dataset layout information. If there is a match, then the file is definitely a Positive. However if there is no match, then this does not automatically mean that the result is a False positive. The reason for this is that most file formats have certain degrees of freedom in their file definitions,

[1]for example by placing it in a separate directory from the Positives.

especially at the start and the end. For example an html file may have a trailing newline that does not necessarily have to be carved to get a correct result, but which still leads to a completely different MD5 sum. Therefore the blocks that were occupied by a carving result are also compared to the block ranges described in the dataset layout information. If the block ranges are the same then the carving result is a Positive.

Note that this still leaves some room for error. If the trailing newline from the previous example were to occur exactly on a block boundary, then this method would still not recognise it as a Positive. Unfortunately testing for this possibility in an automated way is a hard problem, since the allowed extra characters differ per file type and situation. Therefore the decision was made to ignore this possibility when determining the number of Positives.

False positives and Known false positives

Known false positives are the set of carving results of which the carving tool has somehow determined that they are not Positives, for example because they are partial files or contain some sort of corruption. Once the Positives and the Known false positives have been determined, then False positives can be easily found; each result that is neither a Positive, nor a Known false positive, is an Unknown false positive. Adding the number of Known and Unknown false positives gives the total number of False positives.

False negative

The False negative definition needs more elaboration, since in carving the recovery of a file is not necessarily all or nothing. The most important question is how much of the relevant information in a file was recovered from the dataset.

There is no standard answer to this question, since it differs greatly per file type. If 10% of a text file is not recovered, then the other 90% might still provide a lot of information. On the other hand if 10% of a zip file is not recovered, then the other 90% is probably still unaccessible since the file can most likely not be extracted.

To be able to handle this problem in the carving quality method, the assumption is made that all data in a file is equally important. A False negative can then be specified as the fraction of a file that was not recovered. So a file that was fully recovered leads to a False negative score of 0, a file that was not recovered at all gives a False negative score of 1 and a file of which, for example, a quarter of the data was carved leads to a False negative score of 0.75.

For the quality calculations described in the rest of this chapter, there is no need for the False negatives to be a round integer number. Therefore the definition of a False negative can be updated to:

False negative The fraction of a file that was not correctly carved from the dataset.

There are two main reasons why tools produce False negatives, which leads to a further specification:

1. If a tool has no support for a specific file, then files of that type will not be carved and are therefore always False negatives. These files are called *Unsupported false negatives.*

2. As described in the previous chapter, False negatives may also be produced for file types that are officially supported by a tool. The most common cause is if the file is fragmented in a way that the carver cannot handle properly. These results are called *Supported false negatives* and are much worse than Unsupported false negatives, since they reduce the reliability of a tool.

4.1 Quality criteria measures

Now that the different types of results have been defined, the main question to be answered is: "How can these result types be translated into measurable quality criteria?"

Originally a quality system was created based on the number of Positives, Unknown false positives and False negatives. It simply calculated the following two scores:

1. The main score was calculated by giving points for each Positive and subtracting points for the False negatives. The higher this score was, the better a tool performed.
2. A second score was calculated by counting the number of Unknown false positives, which should be as low as possible.

These scores performed reasonably well when comparing different tools, but there were three main problems:

1. There was no simple way to combine both scores to give an overall score of the tool quality.
2. There was no way to instantly see that a tool had (near) perfect results on a dataset, since the scores were not normalised for the number of files in that dataset.
3. There was no direct correlation between the two scores and specific quality aspects of a tool.

During this project the connection to statistical natural language processing was made at a number of occasions. The reason for this is that both fields deal with getting structured information out of (large) sets of seemingly unstructured data.

A book by Manning and Schütze [6] includes a method for measuring the quality of a natural language processor. This method also measures quality based on Positives, False negatives and False positives, but uses these numbers to calculate three quality scores.

The following three quality measurement functions are quoted from page 268 and 269 of [6], where "[ed.]" denotes an explanatory text which was not a literal part of the original quotation:

> *Recall* is defined as the proportion of the target items that the system selected:
>
> $$\text{recall} = \frac{tp}{tp + fn}$$
>
> [ed. ,where tp is the number of Positives and fn is the number of False negatives.]
>
> *Precision* is defined as a measure of the proportion of selected items that the system got right:
>
> $$\text{precision} = \frac{tp}{tp + fp}$$
>
> [ed. ,where fp is the number of False positives.]
>
> [ed. Together these two measures can be combined into a single measure of overall system performance, called the F measure.]
>
> $$\text{Fmeasure} = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}}$$
>
> where P is precision, R is recall and α is a factor between 0 and 1 which can be used to determine the weighting of precision and recall.

In this equation α can be used by a user of this quality method to indicate their relative preference for recall compared to precision.

These equations form the basis for a quality measurement system whose goal it is to answer the following three carving quality questions:

1. What proportion of the available files was recovered?
2. What proportion of the recovered files was correct?
3. How reliable is the tool? If it claims to support a set of file types, then what proportion of these files does it recover?

The first quality question can be answered using a modified version of the "recall" equation. This equation needs to be modified since the original equation does not take into account that it is also possible to return partial solutions. If a carver manages to carve all but one block of every file in a dataset, then it will have recovered almost all relevant data, but still get a recall score of 0, since the number of Positives will be 0.

A tailored carving recall equation was created that takes partial results into account, based on two observations:

1. The denominator in the equation $(tp + fn)$ represents all relevant files present in a dataset. In case of a dataset with known content this can be determined by counting the number of files described in the layout. This means that in the new equation the denominator can be replaced by *all*, where *all* is the number of files in the dataset.

2. In the previous observation $tp + fn$ is replaced by *all*, i.e. $all = tp + fn$ and therefore $tp = all - fn$. This means that the numerator in the recall equation can be replaced by $all - fn$. Since our definition of False negatives (fn) is already defined with partial results in mind, no further change is necessary.

This leads to the following recall equation for carving:

$$\text{carving\_recall} = \frac{all - fn}{all} \tag{4.1}$$

where *all* is the number of files in the dataset and fn is the amount of False negatives.

The second quality question can be answered using a modified version of the "precision" equation. In the original equation there are right (tp) and wrong (fp) results, but in carving there are also semi-wrong results, namely the Known false positives. These are not as bad as normal False positives, since they can be easily distinguished from Positives, but they are still False positives. To take this into account the false positives are split into two groups, namely the Known and Unknown false positives. The number of Known false positives is multiplied with a factor which marks the relative weight of an Unknown false positive compared to a Known false positive.

This leads to the following precision equation for carving.

$$\text{carving\_precision} = \frac{tp}{tp + ufp + \frac{1}{\beta}kfp} \tag{4.2}$$

where ufp is the number of Unknown false positives, kfp is the number of Known false positives and β is a factor $(\beta \geq 1)$, which can be used to determine the relative weight of Unknown false positives compared to Known false positives.

When using this method to make claims about the quality of one or more carving tools, both α and β should be clearly stated and explained, since they can have a profound impact on the different scores.

For the quality tests performed in this thesis, the decision was made to set β to 2, meaning that known false positives are rated to have half as much negative impact as Unknown false positives. This is a subjective decision, based on the fact that even though the known false positives do not obscure the Positives like the Unknown false positives do, they still need to be manually investigated.

A carving specific variation on the F measure, named the cF measure, can now be used to give an overall score for a tool, using the updated "carving_recall' and "carving_precision" scores:

$$cF = \frac{1}{\alpha\frac{1}{cP} + (1 - \alpha)\frac{1}{cR}} \tag{4.3}$$

where ${}_cP$ is carving_precision, ${}_cR$ is carving_recall and α is a factor which determines the weighting of carving_precision and carving_recall.

For the tests performed in this project the choice was made to choose an α of 0.5, meaning that carving_precision and carving_recall are chosen to be equally important. This way the effect of both quality aspects can be most clearly seen.

Note that when using these quality measures in practice, the values of both α and β can be adjusted to more closely suit the goals of the quality test. For example, at Hoffmann carving_recall is generally deemed *much* more important than carving_precision. When testing the quality of tools for use at Hoffmann, the α is therefore set to 0.1 to reflect this preference. However in some investigations carving_precision might be very important, for example if time is severely limited and manual checking is simply too time consuming. In that case larger α can be used and a different tool might turn out to be better suited for the job.

This leaves the third quality measure, the reliability of a tool. Reliability does not state how well a tool works on all the files in a dataset, but only how successful it is at recovering the file types it claims to support. The assumption is that a tool only recovers the file types it claims to support, which means that focussing only on supported file types has no impact on the number of Positives and False positives. This means that the effect of only taking supported file types into account can only be seen in the amount of False negatives.

This led to a modified version of the "carving_recall" measure, in which only supported files are taken into account:

$$\text{supported\_recall} = \frac{supported - sfn}{supported} \tag{4.4}$$

where *supported* is the number of supported files in the dataset and sfn is the amount of Supported false negatives. Comparing the supported_recall score of different tools can give an insight into the reliability of each tool.

4.2 Normative scores

The previous section provides four descriptive scores that can be used to compare the quality of different tools on different aspects, namely:

1. The precision of a tool's results using *carving_precision*
2. The amount of data recovered using *carving_recall*
3. The reliability of a tool using *supported_recall*
4. The overall quality of a tool using the *cF measure*

These scores can be used to describe that, for example, the carving_precision of tool X is higher than that of tool Y.

There is however no rule that determines whether a specific quality score for a tool is good or bad. For this a mapping of the descriptive quality scores to normative quality scores is needed, since normative statements describe the value that can be attached to a score.

There are two reasons why defining normative quality rules is useful in this project:

1. It allows for statements like: "X's carving_recall is *very good*, but due to its *bad* carving_precision it only has a *mediocre* overall cF measure." This statement gives a much quicker insight into the quality of a tool than: "X's carving_recall is 0.87, but due to its carving_precision score of 0.44 it only has an overall cF measure of 0.58."

2. By creating a fixed mapping of descriptive to normative quality scores, different interpretations of the quality scores of tools on different datasets can be avoided.

In order to map descriptive to normative scores, a norm has to be defined based on the descriptive scores. Since the four quality scores have only recently been defined, there is not yet a common consensus of what, for example, a "good" carving_precision score is. This means that the following mapping is a personal interpretation of the different descriptive quality scores to normative quality scores:

Descriptive score range	Normative score
$0 \leq$ descriptive score < 0.5	Bad
$0.5 \leq$ descriptive score < 0.75	Mediocre
$0.75 \leq$ descriptive score < 0.85	Good
$0.85 \leq$ descriptive score < 0.95	Very good
$0.95 \leq$ descriptive score < 1	Almost perfect
descriptive score $== 1$	Perfect

For the sake of simplicity this mapping is used for all four descriptive quality scores.

There is one final normative score that is not based on one of the descriptive scores of the previous section, but which was already defined in section 2.1. This score is based on the speed of a carver.

The speed of a carver is a normative quality score with the following mapping:

MB per second	Normative score
less than 0.58	Unusable
between 0.58 and 1.16	Usable for testing purposes
more than 1.16	Usable in practice

Chapter 5

How to measure carving result quality

In chapter 4 quality measures that can be calculated for a tool were provided, assuming that we know the results a tool produced on a specific dataset. But how to get these results?

5.1 Datasets

First of all, these measures only work if the layout of a dataset is known, since the tool results have to be compared to the files present in that set. Fortunately a number of these datasets have already been created, for the express purpose of testing carving tools.

In 2005 Nick Mikus released two datasets, one based on a FAT32 file system and one based on an EXT2 file system, which are meant to test carving tools. These images, along with their internal layout, can be downloaded from the following locations:
`http://dftt.sourceforge.net/test11/index.html` and
`http://dftt.sourceforge.net/test12/index.html`

The FAT32 image, from now on referred to as the 11-fat image, contains fifteen files. Of these files twelve are normal uncorrupted files, two files (a ppt and a wmv) have been deleted and one is a JPEG file which has been corrupted. None of these files is fragmented.

The EXT2 image, from now on referred to as the 12-ext2 image, contains 10 files. Of these files eight are normal files and two (a bmp and a doc) are deleted files. Due to the nature of the ext2 file system, almost all files are fragmented by sectors that are not part of the file itself. These are the 'indirect blocks' used by the ext2 file system to handle large files. One file is sufficiently small that it does not contain an indirect block, eight files contain one indirect block and one file contains three indirect blocks and one double indirect block.

The last tested dataset is the image that the DFRWS used for their carving challenge, which can be downloaded from `http://dfrws.org/2006/challenge/submission.shtml`. The layout was presented after the challenge and can be found on the following page: `http://dfrws.org/2006/challenge/layout.shtml`.

This image is not based on a specific file system, but on a 48 MB raw file filled with random data, in which (fragments of) files have been placed at specific offsets. It contains 32 files of six different file types, with varying degrees of linear fragmentation.

These three images are used in chapter 6 to test the quality of five commonly used carving tools.

5.2 Testing procedures

Each tool is tested by running it on the different datasets and comparing the results to the layout provided for that set.

The comparison is done in four phases:

1. The MD5 sums are calculated over the carving results and compared to the MD5 sums of the files in the image, if provided in the layout description. Matching files are marked as Positives.
2. The remaining carved files are checked against the remaining image files by comparing occupied block ranges. Files with exact matching block ranges are marked as Positives.
3. The remaining results, which have not been marked as Known false positives by the carver itself, are marked as False positives.
4. Finally the block ranges occupied by the not fully carved image files are compared to the block ranges carved by the carver. This is used to determine the False negatives.

The time needed by each tool to carve a specific dataset is also noted to determine its average carving speed for that set.

Note that the Known false positives are also taken into account when determining the number of Positives. The reasoning here is that in those rare cases when a Known false positive is a precise match for a file in the dataset, i.e. a Positive, then there is usually a problem with the original file. An example is the “domopers.wmv” file in the dataset, which contains corruption. If this file is correctly carved from the dataset, but due to its corruption is marked as a Known false positive, then it should still be counted as a Positive[1].

5.3 Score interpretation

Using the results of these comparisons, the quality scores that were specified in chapter 4 can be created for each combination of a tool and a dataset. These scores each give a different insight into the quality and improvement possibilities of the tools.

[1]This is a subjective and personal choice, but made in agreement with Joachim Metz of Hoffmann

Overall quality

- carving_precision: Low scores in carving_precision mean that relatively many False positives are produced and that the tool could be improved by improving its ability to detect and avoid, or mark, incorrect results.
- carving_recall: Low scores in carving_recall mean that relatively few files are retrieved correctly. Either because the tool supports too few file types, or because it fails to (fully) carve the files it does support.
- cF measure: This gives an overall score for a tool on a dataset and can be used to compare different tools.

Reliability

- supported_recall: Low scores in supported_recall mean that the tool had many problems carving the files it claims to support.

If the carving technique(s) used by a tool are known, then these scores can also be used to assess the quality of different carving techniques. Note however that one should be careful to assess the quality of techniques on this basis, due to two reasons:

1. Each tool has its own file definitions, and these may differ widely in extensiveness and even accuracy.
2. Even though two tools use the same basic carving technique(s), their implementation may be very different. This can also have an impact on the results.

Therefore results for multiple tools using the same technique(s) should be taken into account and even then making claims about the quality of a technique should be done with caution.

Chapter 6

Quality of current tools

6.1 Tested tools

To test the quality of existing tools, the tools most commonly used by researchers at Hoffmann have been tested. These include both open and closed source tools.

The open source (GNU/Linux-based) tools that have been tested are:

- PhotoRec
- Foremost
- Scalpel

These tools are especially interesting, since the used techniques are known. This means that, as stated in chapter 3.2, results for these tools can give an indication of the quality of different techniques. Both PhotoRec and Foremost use a combination of file structure based carving and header-footer carving, whereas Scalpel uses a combination of header-footer and header-“maximum file size” carving.

The closed-source (Microsoft Windows based) tools that have been tested are:

- Recover My Files
- Forensic Toolkit (FTK)

For more information on these five tools, see appendix B.

6.2 Scores on the different datasets

This section shows the scores that the different tools get on the datasets described in chapter 5.1. For each table an interpretation is given of the most relevant results.

To keep these tables readable and relevant, not all information has been depicted. This sometimes means that an interpretation is based not just on the values in these tables, but also on information from the full tables in appendix A.

The tests were performed on a Dell notebook with an Intel Core2 2x1.66GHz CPU and 2048 MB of RAM. Note that the MB per second values are not always 100% precise, since not all tools accurately log the time that was needed to process a dataset. In these cases the elapsed time was logged manually.

Tool	Carving Precision	Carving Recall	Supported Recall	cF Measure	MB per second
Foremost	0.786	0.933	0.933	0.85	20.67
FTk	0.75	0.462	0.703	0.57	1.07
PhotoRec	0.857	0.931	0.931	0.89	15.5
Recover My Files	0.923	0.975	0.975	0.95	2.48
Scalpel	0.003	1.0	1.0	0.01	3.1

Table 6.1: Current tool quality score for 11-fat

Interpretation of the results on the 11-fat image by Nick Mikus (table 6.1):

- Recover my files gets very good carving_precision and almost perfect carving_recall, so its cF measure score is almost perfect. The techniques it uses cannot easily be determined, since the program is closed source. Therefore nothing can be said at this moment about the effectiveness of its technique(s).

- Foremost and PhotoRec also get very good cF measure scores and manage to carve almost all relevant data. Both use hard coded file structure based carving for a limited amount of file types and header-footer based carving for the rest, which on this image appears to be an effective combination.

- FTK is the only tool that does not support all file types in this image. Even if the unsupported file formats are ignored, by examining the Supported recall score instead of the full carving_recall score, then it still performs worse than any of the other tools.

- Scalpel is the only tool with perfect carving_recall, but still manages to get by far the lowest cF measure score. There are so many False positives, more than a 100 False positives for each file in the dataset, that this tool is simply unusable in practice. The main reason for this poor performance is that it carves based on headers-footer or header-“maximum file size”, without performing any checks on the intermediate data. Exploratory tests showed that this behaviour is even worse on the other datasets, so Scalpel has not been tested further.

Interpretation of the results on the 12-ext2 image by Nick Mikus (table 6.2):

- First of all there are two extra results that are shown, namely Foremost-ext2 and PhotoRec-ext2. Both Foremost and PhotoRec have the ability to take the file system information present in an ext2 image into account to improve their carving results. These are not pure carving results, but they have been added for comparison to show the effect that this support can have on the quality of the results.

Tool	Carving Precision	Carving Recall	Supported Recall	cF Measure	MB per second
Foremost	0.455	0.7	0.7	0.55	11.27
Foremost-ext2	0.6	0.988	0.988	0.75	11.27
FTk	0.273	0.741	0.741	0.4	1.02
PhotoRec	0.333	0.692	0.692	0.45	12.4
PhotoRec-ext2	1.0	1.0	1.0	1.0	12.4
Recover My Files	0.333	0.898	0.898	0.49	3.1

Table 6.2: Current tool quality score for 12-ext2

- PhotoRec with ext2 support turned on manages to get perfect results on this image. However, with ext2 support turned off it got the second worst results of all tools.
- Foremost with ext2 support turned on has almost perfect carving_recall results, but only mediocre carving_precision. Apparently PhotoRec's implementation in this case is better than Foremost's.
- The best result produced by only using carving methods, were created by Foremost. However, both carving_recall and carving_precision are at most mediocre for Foremost, and even worse for the other tools. This is because they cannot (fully) deal with the indirect blocks that fragment the larger files.

Tool	Carving Precision	Carving Recall	Supported Recall	cF Measure	MB per second
Foremost	0.269	0.721	0.721	0.39	24
FTK	0.25	0.586	0.644	0.35	0.15
PhotoRec	0.633	0.875	0.875	0.73	1.78
Recover My Files	0.37	0.83	0.83	0.51	1.78

Table 6.3: Current tool quality score for dfrws2006

Interpretation of the results on the DFRWS 2006 dataset (table 6.3):

- This image has many fragmented files, but no file system information like the 12-ext2 image. None of the tools get more than mediocre F measure scores, with both Foremost and FTK simply producing bad results.
- Even though this is the smallest dataset of the three, the amount of files present in it has a clear and negative effect on the speed of the tools. This means that a measure of algorithm speed based on the size of an image may not be the best choice.
- PhotoRec has very good carving_recall results, but only mediocre carving_precision results. Still, it has a much better overall performance (cF measure) than any of the other tools.
- The other tools are reasonably reliable on this set, as can be seen in their mediocre to good supported_recall scores, but get (very) bad carving_precision scores.

- Recover My Files has carving_recall results that are almost as good as PhotoRec, but has a much lower overall score due to its carving_precision.
- FTK is once again the only tool that did not support all file types. Besides that, it is also *much* slower than the minimum speed required for a tool to be considered usable for testing.

6.3 Overall conclusions

On the 11-fat image the results for Foremost, PhotoRec and Recover My Files are very good, but this image had no fragmentation. The main lesson to be learned is based on Scalpel's results, namely that header and footer carving by itself is too imprecise to get usable carving results.

The 12-ext2 image gives us much more information. First of all it shows that file system specific support can greatly increase the efficiency of a carver. However, this is not a carving technique, so it falls outside the scope of this project.

If file system support is not present or disabled, then the indirect blocks can be seen as a form of fragmentation. In this case even the file structure based carvers get bad cF measure scores. So better fragmentation detection is needed if ext2 file systems are carved with generic carving methods.

The DFRWS 2006 image confirms the findings of the 12-ext2 image, namely that current tools have great problems handling fragmentation. Improving this can lead to more Positive and fewer False positives, which leads to better carving_precision.

There are two observations that can be made for all datasets.

All tools except FTK support all file types in these images. This means that their carving_recall rates are also a measure of their reliability. Both the 12-ext2 and the DFRWS 2006 image show that the reliability of current tools on fragmented files is still far from perfect. The reliability of FTK is low to mediocre for all images, even when only taking the supported file formats into account.

The tools have another common problem, namely that they do not detect False positives. Making even part of these 'known' can have a very positive effect on the carving_precision, but this depends on the size of the β factor.

Chapter 7

How to improve carving quality

Chapter 3 to 6 focussed on the first part of the goal of this project:
"Define meaningful criteria and a method to measure the quality of carving tools."

However, the goal of this project is twofold. The second part reads:
"Based on the quality of current tools, develop a carving tool which achieves better results."
This chapter is the start of the part of the thesis that focusses on fulfilling this second part of the goal.

The chapter is divided into two sections. The first section describes the specific improvement goals, whereas the second section describes the approach taken to meet these goals.

7.1 Specific improvement goals

Chapter 6.3 shows that improvements can be made both in carving_recall and carving_precision quality.

Higher carving_recall

Maximising carving_recall is all about detecting as much useful information as possible and not discarding interesting results. Therefore the following specific carving_recall goals are:

- Support many file types to decrease the number of Unsupported false negatives.
- Do not discard partial results, since they still might contain useful information. Carve them, but mark them as Known false positives. This increases the carving_recall score, but decreases the carving_precision.
- Carve corrupted files as Known false positives. If possible continue carving a file even when corruption has been detected, to recover as much of the

file as possible. This increases the carving_recall score, but decreases the carving_precision.

Higher carving_precision

Carving Known false positives instead of discarding them will have a negative impact on carving_precision, so increasing carving_precision becomes even more important than before. The specific carving_precision goals are:

- Detect False positives produced by the carver and automatically mark them. This will not avoid False positives, but by making them known their negative impact will decrease.
- Better fragmentation handling than current tools. If a fragmented file can be carved as a full file, instead of as one or more partial files, then this increases the number of Positives and decreases the number of False positives.

7.2 Improvement approach

This section describes the approach that was taken to fulfill each carving goal. Individual goals:

- *Support for many file types*: Investigate many file formats and create definitions for as many file types as possible. If possible separate the file format definitions from the carving code, so both can be developed relatively independently. This is common for header-footer based carvers, but not for more complicated carvers, like file structure based carvers.
- *Better fragmentation handling*: Combine a file structure based carver with content carving techniques, to improve fragmentation detection and handling in little or unstructured areas. Create extensive file format descriptions, to maximise the amount of guidance provided to the carving algorithm.
- *Carve partial files*: Carve partial files and mark them as Known false positives.
- *Carve corrupt files*: File formats often contain redundant information, add this to the file format description as well to help detect corrupt files and carve as much as possible. Carve corrupt files and mark them as Known false positives.
- *Detect false positives*: Create a validator which checks the carved files and marks them as 'valid', 'viewable' or 'unviewable'. Valid files are marked as Positives, (un)viewable files are marked as Known false positives. In this context a valid file is a file that can be opened using a corresponding tool[1], without that tool producing any errors. A viewable file is a file that is viewable using a corresponding tool, but for which the tool provides errors or is unable to show all contents.

[1] A corresponding tool is a tool which was created to handle the type of file being checked.

The following chapters describe the different steps that were taken to create a carving framework that can fulfill these different goals.

Chapter 8

MultiCarve

MultiCarve is a file carver that was developed with two goals in mind:

1. Help to acquire the required knowledge about the carving problem domain. This is a very complex domain of which relatively little is known or documented, so it is important to get hands on experience.

2. Provide a testbed to try out (new) techniques and carving approaches.

This means that MultiCarve is not meant to directly meet any of the improvement goals stated in the previous chapter, but to gain the insight needed to effectively address these goals.

In order to meet the goals set for MultiCarve, a flexible framework was designed and implemented in Ruby[1]. The choice for Ruby was a very pragmatic one, it is a very flexible language that I feel comfortable developing in.

The part of this design which deals with tasks like handling the user interaction and reading the dataset is rather straightforward and not very relevant to this thesis. The part that *is* relevant to this thesis is the "carving method handler".

8.1 Carving method handler

The goal of the carving method handler is to support multiple different carving methods in one program. It was designed to minimise the impact of implementation differences when testing the quality of different file carving methods. This means that the different supported carving methods can:

- all use a number of common carving functions and

- all share a single file format definition.

A graphical representation of the carving method handler is shown in figure 8.1. This image shows the main components:

[1]`www.ruby-lang.org`

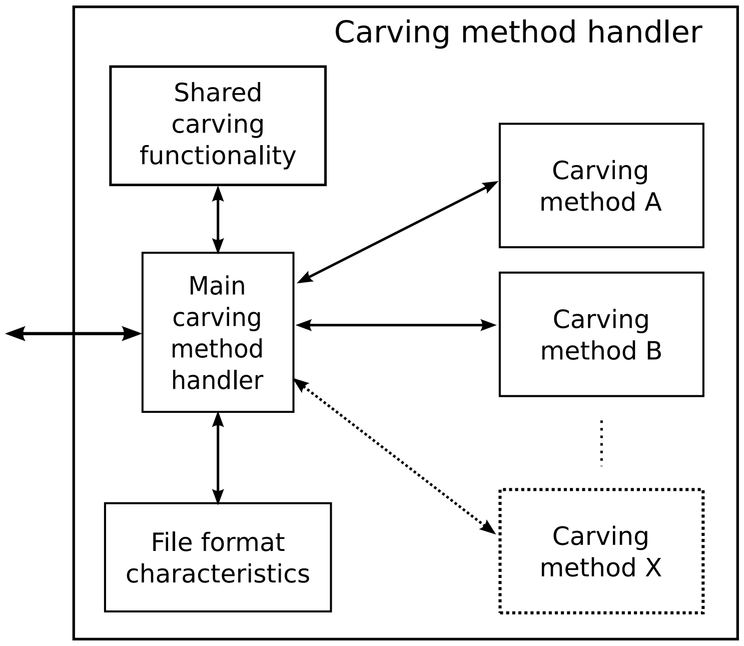

Carving method handler
Shared carving functionality
Carving method A
Main carving method handler
Carving method B
File format characteristics
Carving method X

Figure 8.1: Carving method handler

- Shared carving functionality, which provides functions that can be used by all different carving methods.
- File format handler, which provides the different carving methods with the file format information that is relevant to that specific method.
- A number of different carving methods, for example a header-footer and a file structure based carving method.
- Main carving method handler, which handles the communication between the different components in the carving method handler. This module also handles the information exchange between the carving method handler and the rest of the MultiCarve program.

Shared carving functionality

This module provides the functions that are relevant to multiple carving methods. Examples are:

- read data from a specific position in the dataset.
- search for a byte string in a specified range.
- calculate the CRC or MD5 value over a specific range of data.
- calculate the entropy of a specific range of data.

Implementing common functionality in one shared module decreases the complexity and increases the readability of the individual carving methods.

File format handler

Different carving methods need different elements of a file format definition. A file structure based carver is interested in the entire file structure, while a header-footer carver is only interested in the start and the end of a file. Neither are interested in the different entropy values that are allowed for a specific part of a file, but this may be needed by a block content based carver.

The file format handler handles these different interests by providing different views of the same file definition. This means that one definition for each file format can be used, without forcing the different carving methods to deal with information that is not of interest to them.

Carving methods

Two carving methods were designed and implemented in MultiCarve:

1. Header-footer carving
2. File structure based carving

The header-footer carving method is relatively simple and produces the same results as the Scalpel carver. It was used as a way to learn basic file carving principles and has no further relevance to this thesis.

The main carving technique developed for this framework is a file structure based carving method. It uses much more information from a file format than the header-footer based carver does, which means that there are more points at which it can check whether the data being carved is still correct.

To show the type of structures that can be present in a file format, the next section gives an example of the buildup of a PNG file. Section 8.3 describes MultiCarve's file structure based carver and shows how it is able to deal with PNG files.

8.2 PNG file structure

To explain the PNG file structure in more detail, the example from chapter 3.2 is repeated here.

Figure 8.2 shows the file structure of a very small PNG file:

1. The PNG file starts with a header byte string.
2. The next piece is a 4 byte string which declares the size of the next section. It is a big-endian hexadecimal string which in this case calculates to 12 bytes.
3. "IHDR" is the identifier of this next section.
4. The 12 bytes that make up the "IHDR" section are content data, which have no fixed file structure.
5. Then there are 4 bytes of CRC data over the "IHDR" section.
6. Then there is again a 4 byte string which declares the size of the next section, in this case 688 bytes.

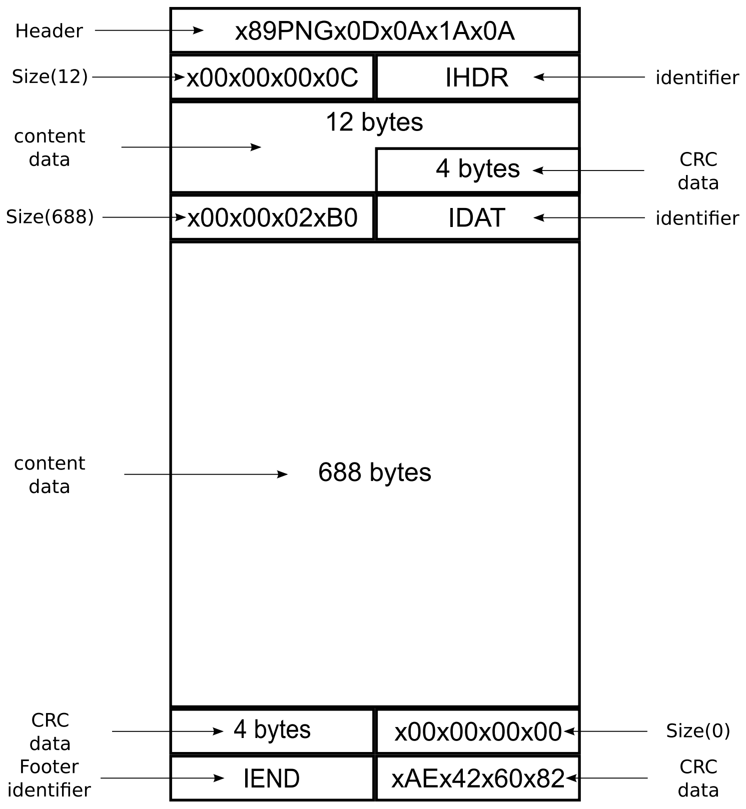


Figure 8.2: PNG file structure example

7. “IDAT” is the identifier of the next section, which is the section that contains the actual image data.
8. The next 688 bytes are image data.
9. Then there are 4 bytes of CRC data of the “IDAT” section.
10. Finally there are the size, identifier and CRC of the “IEND” footer section, which is always the final section of a PNG file. Since this section has no content, the CRC is always the same.

As this example shows there is quite a lot of repetition and structure in this file format. The header byte string and the different identifiers are elements that can be used to recognise specific parts of a file. Layout information like the size data can be used to identify the position where these byte string elements can be found. Besides the structural elements, a PNG also contains content data, like the image data, which has no fixed file structure. The CRC data can be used to verify the validity of the content data.

These different information elements can be used to create a file structure based carving approach for PNG.

A hard-coded file structure based carver, like PhotoRec, contains an algorithm for PNG carving which follows the same structure as the file format itself. The different fixed strings like the header and section identifiers are either hard coded or read from a configuration file.

This solution works rather well, but suffers from a number of problems:

1. Big algorithm changes, like fragmentation support, probably have to be implemented for each file type individually.
2. File structure information cannot easily be shared between different programs or communicated between different persons.

In MultiCarve the file format definition and the carving algorithm are separated, so another approach is needed. This means that the carving algorithm needs to get both element and layout information from the configuration file.

The file format definition used by MultiCarve describes both the content and layout of a file in sufficient detail to guide a single carving algorithm that can handle all file types.

8.3 File structure based carving

MultiCarve’s file structure based carving approach is explained in two phases.

Subsection 8.3.1 describes a very simple file format, the definition MultiCarve would use to describe this format and the way the file structure based carver uses this definition to carve a file of this format.

Subsection 8.3.2 describes how PNG files are defined in MultiCarve, to show how this definition works on a real file format.

8.3.1 Simple file format carving

The “foobarbaz” file type is an extremely simple file type. In fact there are three possible instances of this type:

- “foobaz”: A file consisting of the string “foobaz”
- “foobar*****foo”: A file consisting of the string “foobar”, followed by 5 unspecified bytes, followed by the string “foo”.
- “foobar*****baz”: A file consisting of the string “foobar”, followed by 5 unspecified bytes, followed by the string “baz”.

To write this file type in MultiCarve’s definition, two main parts must be described.
First of all byte strings which are present in the definition are defined as “elements”. These are written using the following syntax:
“KEY: {BYTESTRING: TYPE}”
In this case this can be written as:

```
elements:
  A: {"foo": String}
  B: {"bar": String}
  C: {"baz": String}
```

This describes the byte strings present, but not the order in which they can occur or the presence of the five unspecified bytes. For this a description of the layout is needed.
This basic layout definition is built up of references to the previously defined elements, i.e. A, B, C, and operators which define the position and order in which these elements may occur. The following three operators are used in this layout description:

1. ‘.’: A.B means that element A is followed directly by element B.
2. ‘OR’: A OR B means either element A or element B.
3. ‘rel_offset(X)’: rel_offset(X) describes the fact that there are X bytes for which no file structure data is known, which is handled as a relative offset.

Using these operators the “foobarbaz” file type can be defined as follows:

```
elements:
  A: {"foo": String}
  B: {"bar": String}
  C: {"baz": String}

layout:
  X: A.C OR
     A.B.rel_offset(5).Y
  Y: A OR C
```

As the example shows, a layout definition may include both the keys of element definitions as well as links to other layout definitions.

File structure based carving based on this definition

The file format definition describes both content and layout, so the file structure can use this information to guide its decisions. This means that the basic idea behind a file structure based carver can be rather simple. This paragraph describes the file structure based carving algorithm used by MultiCarve. Note that this algorithm does not take file fragmentation into account.

The MultiCarve file structure based carver design works according to the principle of block based carving. Block based carving is based on the observation that physical media like hard disks write their data in sectors (blocks of 512 bytes), as already mentioned in chapter 4. Since only one file can occupy the same space on a drive, each block belongs to at most one file. This also means that each file starts at the start of a block.

In the following example the "foobarbaz" definition of the previous paragraph is used. The carver is used to investigate a special dataset that is made up of blocks of 3 bytes large, which contains the file **"foobarquuxxfoo"**. This file starts at the start of a block, i.e. it is block aligned.

To check the dataset for the presence of a "foobarbaz" file, the carver calculates all possible starting blocks of this file type, which in this case is only one block: "foo". Now each block of the dataset is checked until a "foo" block is found.

Once the carver finds a "foo" block, a "foobarbaz" file state is created and the blocks that are allowed after the "foo" block are determined:

- "foobarbaz":
  - "foo": "baz"
  - "foo": "bar"

In our example the next block is "bar", so the *"foo": "baz"* sub-state is terminated and the *"foo": "bar"* sub-state is extended with the next possible blocks, leading to the following file state:

- "foobarbaz":
  - "foobar": "***"

The next block in our example is "quu", leading to:

- "foobarbaz":
  - "foobarquu": "**b"
  - "foobarquu": "**f"

The next block in our example is "xxf", terminating one sub-state and leading to:

- "foobarbaz":
  - "foobarquuxxf": "oo"

The next block is "oo" followed by a byte that is of no interest to the carver. At this time the "foobarbaz" file definition is complete and the file "foobarquuxxfoo" can be carved.

If the final block had been incorrect, for example "aax", then there would be no more possible sub-states left. In this case "foobarquuxxf" would have been carved as a partial result. This is not just the case for the final block; if the last sub-state is removed before the file is finished, then that sub-state is carved as a partial result.

This can be generalised to create the following block based carving method, which can handle multiple file types.

1. Create a list of all the possible starting blocks of each file type.
2. Read blocks from the input until a block matches the starting block of one of the file type definitions, which then becomes the active file state.
3. As long as each block from the input matches the active file state and the definition layout is not yet completed, it is assigned to that state. Otherwise:
   (a) if the definition layout is completed, then the active file state is carved as a complete result and the algorithm goes to step 1.
   (b) If the block does not match the active file state, then that state is carved as a partial result and the algorithm goes to step 1. In this case no new block needs to be read from the input, since the current block may be the start of a new file and should be checked as such.

This algorithm runs until all blocks have been handled. If there is still an active definition state at that time, then it is carved as a partial file.

8.3.2 PNG file definition

A real file format definition is much more complex than the simple definition from the previous subsection and the full definition syntax needs to be more extensive than the components described so far.

This subsection describes how the PNG file format can be defined using MultiCarve's definition format.

The PNG file depicted in figure 8.2 consists of a header, two internal data structures and a footer data structure. Each of these data structures, which in a PNG is called a segment, is made up of 4 bytes of size data, an identifier, a content part and 4 bytes of CRC data.

Besides the terms already introduced in the previous subsection, three more extensions to the layout definition need to be introduced to describe MultiCarve's PNG definition:

1. EMPTY: This keyword is used as a means to end a recursive definition and is interpreted as a "do nothing" instruction. For example to describe that a file is made up of zero or more repetitions of an A element, the following recursive syntax would be used:

- X: A.X OR EMPTY

2. DATA: Sometimes it is necessary to read and interpret data from the file being carved. For example the PNG file shown in figure 8.2 contains a number of 4 byte areas that describe the size of a specific part of the file. This data is in big-endian byte order and the command to get the data from the file and save it in the "size" variable is:

   - DATA(in: Num, 4, big, out: size)

3. (sub-)definition references: The layout cannot just refer to elements or other layout definitions, but also to other (file) definitions. In the PNG file definition there is a definition called Segment, which can be referred to in the layout as "PNG::Segment". Note that it is even possible to refer to full file definitions, which can be used to define the presence of an embedded file.

The following definition is the PNG definition used by the MultiCarve carver, which uses the extra components just described. Note that this definition does not use the CRC information, but treats it as unstructured information.

```
PNG:
  elements:
    start: {"\x89PNG\x0d\x0a\x1a\x0a": String}
    end: {"IEND????": Pattern}
  layout:
    l_main: "start.PNG::Segment.l_segment.REL_OFFSET(4).end"
    l_segment: "PNG::Segment.l_segment OR EMPTY"
  Segment:
    layout:
      l_main: "DATA(in: Num, 4, big, out: size).
               PNG::Segment::Start.
               REL_OFFSET(size).REL_OFFSET(4)"
    Start:
      elements:
        start1: {"IHDR": String}
        start2: {"PLTE": String}
        start3: {"IDAT": String}
        ...
        start15: {"iTXt": String}
        start16: {"tEXt": String}
        start17: {"zTXt": String}
      layout:
        l_main: "start1 OR start2 OR start3 OR ...
                 start15 OR start16 OR start17"
```

The main parts of this definition are explained using the terms introduced in the first part of this subsection.

At the highest level is the PNG definition itself, which contains three main components: *elements*, *layout* and a sub-definition named *Segment*.

The "start" and "end" elements specify the header and footer byte strings. The "end" element is a Pattern, instead of a String, which means that the question marks are wildcards which represent an unspecified byte.

The layout shows how its different components can be combined to describe the main structure of a PNG file:

```
l_main: "start.PNG::Segment.l_segment.REL_OFFSET(4).end"
```

describes the fact that a "start" element definition is followed by a "PNG::Segment" sub-definition, followed by an "l_segment" layout definition, followed by 4 unstructured bytes and finally the "end" object definition.

The "l_segment" layout definition shows the use of the 'OR' keyword and the 'EMPTY' keyword.

```
l_segment: "PNG::Segment.l_segment OR
           EMPTY"
```

This definition specifies that there may be zero or more PNG::Segments, by using a recursive definition and the 'EMPTY' keyword to break out of the loop.

The main point of interest in the Segment component is that it uses the DATA function in its layout to read the data which is later used to specify the size of the REL_OFFSET.

Note that even with this much more complicated definition, the basic carving algorithm remains the same. The only difference is that determining the possible blocks becomes more complicated, since there are more definition elements to be handled.

8.4 Does MultiCarve meet its goals?

To recapitulate, MultiCarve was created with two main goals in mind:

1. Help to acquire the required knowledge about the carving problem domain. This is a very complex domain of which relatively little is known or documented, so it is important to get hands on experience.
2. Provide a testbed to try out (new) techniques and carving approaches.

The first goal worked out very well. Developing a file structure based carver based on a completely new definition principle means that a lot of research had to be done on both the common elements and the differences between file formats, as well as the problems posed by the datasets that contain these files. This knowledge was both invaluable when designing the fragmentation support for the Revit carver described in chapter 10 and when adding new file format definitions to that carver.

As a testbed for techniques and carving approaches, MultiCarve also achieved rather good results. It is a file structure based carver that achieved the separation between file format definitions and a single generic carving algorithm.

Revit07, a carver which was developed parallel to MultiCarve, is the only other carving tool which supports this[2].

The next logical extension to MultiCarve would be to add support for handling fragmentation. The following chapter explains the problems that are caused by file fragmentation and describes an algorithm that uses a combination of file structure based carving and content based carving to deal with these problems.

[2]Based on examination of (open-source) carvers and configuration files

Chapter 9

Fragmentation handling algorithm

MultiCarve meets two of the goals set in chapter 7.1, namely the "Support for many file types" and the ability to "Carve partial files". This chapter describes an algorithm that was developed to meet another one of the main goals:
"Better fragmentation handling".

The main goal of this algorithm is to be able to carve fragmented files, while still handling unfragmented and partial files as well or better as the file structure based carver described in the previous chapter.

The decision was made from the start to only support linear fragmentation[1], for three main reasons:

1. Fragmentation is a very hard problem to handle, therefore it will be handled one step at a time: first understand and be able to handle *linear* fragmentation, before undertaking support for *nonlinear* fragmentation.

2. Little is known about the nonlinear fragmentation scenarios that occur on normal systems. In [4], Simson Garfinkel investigated fragmentation on 449 disk images and found that linear fragmentation is very common. There is unfortunately no such paper on the presence and frequency of nonlinear fragmentation. Handling of nonlinear fragmentation will probably take a lot of time to design and implement correctly, so knowing whether or not nonlinear fragmentation is common on normal file systems is very important to decide whether it is worth the effort. Due to time constraints the decision was made that, for now, the effort involved is too great.

3. The algorithm is designed to limit the time needed to process a dataset, by only making a single run over the data. For linear fragmentation this poses no real extra problems, since information from the already carved fragment(s) of a file can be used to recognise subsequent fragments. With nonlinear fragmentation this information is not available, making it much harder to detect and match all fragments.

[1]Described in section 3.1

9.1 Specific problems to solve

To summarise the start of this chapter, an algorithm needs to be designed that is able to handle:

1. Complete, contiguous files
2. Partial files
3. Linearly fragmented files

A method for block based carving of complete and partial files has already been discussed in the previous chapter, but linear fragmentation introduces a completely new set of problems.

First of all the location in a file where the fragmentation occurs influences how hard it is to handle that fragmentation. Take for example the PNG depicted in figure 8.2 in the previous chapter. If the fragmentation occurs halfway down the header, then a file structure based carver knows that the next block should start with the second part of the header. If it does not, then the carver will detect that the block does not match its expectations. It can then continue processing the data in the dataset, while putting this file 'on hold'. Once it detects the start of a matching block, it can continue carving the fragmented file.

Fragmentation handling depends very heavily on one crucial point, namely the ability of a carver to determine whether a block does or does not match the file definition. If the fragmentation in the previous example had occurred in the image area, where there is little to no file structure information, then this would have been much more difficult. In this case a carver has to use more information than file structure information alone, for example content data.

The following section contains a number of scenarios that a carver might encounter in a dataset. They contain different combinations of complete, partial and linearly fragmented files and have been specially designed to highlight some of the difficulties a carver has to be able to handle.

9.2 Scenarios

Each scenario consists of a graphical representation of a block-based dataset that contains 2 files and a description of how the files in the dataset would be carved if a block based carver could perfectly detect the start and end of complete files and fragments in the dataset.

Each dataset is made up of two types of data: the data blocks that belong to a file and the rest of the data in the dataset, called *non-file* blocks. Non-file blocks can for example be blocks that contain file system information. These non-file blocks are checked by the carving algorithm, but since they do not contain relevant information for the carving process they are otherwise ignored.

The following three subsections each describe a scenario.

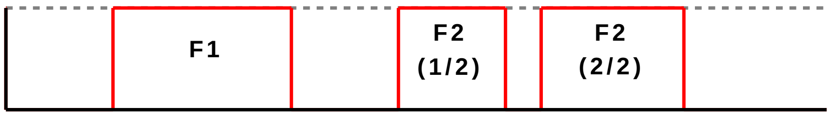


Figure 9.1: Complete file and fragmentation

9.2.1 Complete file and fragmentation

This first scenario depicts a dataset with one complete file F1, followed by a fragmented file F2.

It can be handled as follows:
The data before F1 consists of *non-file* blocks, which can be ignored by the carver. At some point the carver will detect the beginning of F1 and start carving it until it detects the end of the file. At this time the carver may extract the file, or save the relevant information to allow extraction at a later time. The carver can then continue processing the rest of the data, but will do nothing with the blocks until it detects the start of F2. While processing F2 it will detect the fragmentation and the processing of F2 is put on hold while it handles the non-file blocks in its middle. Processing of F2 is continued once the carver detects its second fragment, and stops when the end of the file is reached, after which it can be extracted. The final non-file blocks are checked and ignored.

The biggest difficulty in this scenario is detecting where the fragmentation of F2 starts and ends. As described in the previous section, this is very much dependent on a carvers ability to determine which blocks do or do not belong to F2.

9.2.2 Intertwined files

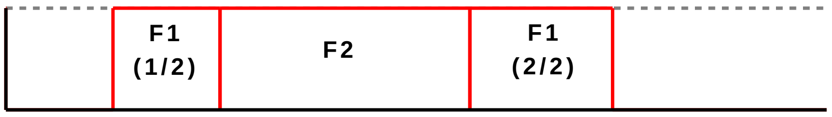


Figure 9.2: Intertwined files

This scenario depicts two intertwined files, where the first part of the fragmented file F1 is directly followed by the complete file F2, which in turn is directly followed by last part of F1.

It can be handled as follows:
After handling a number of non-file blocks, the carver will detect the start of F1 and start processing it. When the presence of F2 is detected, processing of F1 is put on hold while F2 is carved. After handling F2 the carver will detect that F1 continues, so processing of F1 is continued until its end, after which it may be extracted. The final non-file blocks are checked and ignored.

The difficulty in this scenario is basically the same as in the first scenario, i.e. detecting where the fragmentation of F1 starts and ends, but with one major difference. The start of F2 is a clear indication that F1 is fragmented at that point, so this can be used by a carving algorithm to detect the fragmentation of F1(1/2). While in this scenario F1 starts directly after F2, there is no guarantee that this will always be the case. So the start of the second fragment is just as difficult to detect as in the first scenario.

9.2.3 Partial file and fragmentation

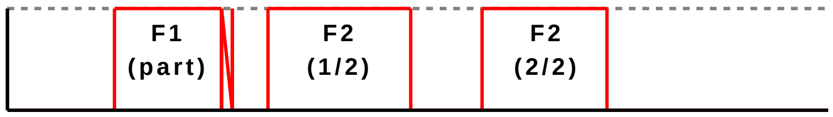


Figure 9.3: Partial file and fragmentation

This final scenario depicts a dataset which contains both a partial file F1 and a fragmented file F2. This means that the algorithm has to (at some point) carve F1 as a partial result.

After a number of non-file blocks the carver detects the start of F1 and starts processing it. When F1 ends abruptly, since it is only a partial file, the carver will put processing of F1 on hold in case the file continues further on in the dataset. After a number of non-file blocks the start of F2 is detected and the carver starts to process it. Just like with F1 there will come a point where the carver detects the premature end of the file, so F2 is also put on hold. After a number of non-file blocks the carver detects the start of a second part of a file, but it has to determine whether it is part of F1 or F2. F2(2/2) matches F2(1/2), so F2 is reactivated and can be fully carved. F1 is still on hold once the end of the dataset is reached, so it is carved as a partial file.

9.3 Algorithm description

The previous section described the main types of scenarios a carver might have to deal with. To handle these types of scenarios effectively, the algorithm described in this section was designed.

This section is set up as follows:

1. The first half of this section describes the fragmentation handling algorithm.
2. The second half of the section describes how the algorithm handles the different scenarios.

9.3.1 High-level description of the fragmentation handling algorithm

Figure 9.4 shows a very high-level representation of the fragmentation handling algorithm.

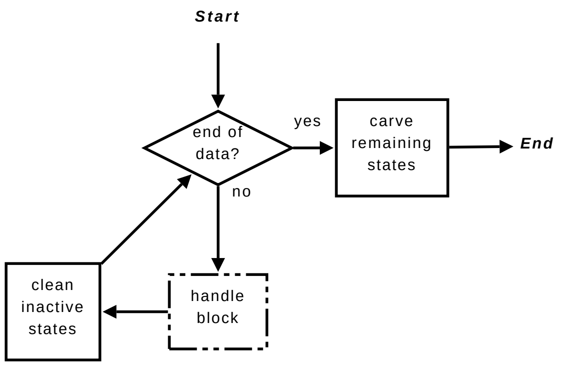


Figure 9.4: Fragmentation handling algorithm

The goal of this algorithm is to carve as much relevant data from the dataset as possible in a single run. Where possible parts of a linearly fragmented file are recombined and carved as a single file. In the cases where the algorithm is unable to detect and match all parts of a fragmented file, the detected parts are carved as partial results.

The algorithm works by processing one block at a time, just like MultiCarve's file structure based carver, as long as the end of the dataset has not been reached. For each block the *handle block* component is executed, which examines each block to determine whether it is the start of a new file, part of an already detected file or not part of a file.

When the *handle block* component detects the presence of a file, this is marked by the creation of a *file state* for that file. The task of this file state is twofold:

1. Keep track of the blocks that the carver has marked as being part of the file it represents. The carving algorithm marks a block as being part of a specific file by *assigning* a block to its file state. A block is never assigned to more than one file state, since a block can only be part of one file.

2. Provide the functionality to test whether a block may be part of the file it represents[2], based on the file format definition and the blocks that have already been assigned to the file state. If a block passes this test, then it is said to *match* the file state.

As the carver processes the dataset, it may encounter multiple files and create multiple file states. If the carver is processing blocks of a specific file, then its file state is called the *active* file state. If non-file blocks or blocks from another file are encountered before a file is finished, then its file state becomes *inactive.* This happens if a file is fragmented or partial. Once a file has been fully processed it is carved and its file state is removed.

If the dataset contains partial files or fragmented files that the carver was unable to fully recover, then these files may lead to inactive file states that will never be completed. After executing the *handle block* component the main algorithm "cleans" the inactive states. This means that the main algorithm checks to see if

[2]For example by using a file structure based check like in MultiCarve

there are any inactive states that meet a pre-specified condition[3]. These states are carved as partial results.

Once all data has been processed, a final check is performed to see if there are any inactive file states left. If so then these are carved as partial results.

9.3.2 Block handling algorithm

The whole algorithm is based around the “handle block” component, which consists of the block handling algorithm described in this section. The overall goal of this algorithm is to determine for each block whether it is part of an existing file state, the start of a new file or not part of any file at all. To do this a file state must be able to check whether a block matches it.

Check whether a block matches a file state

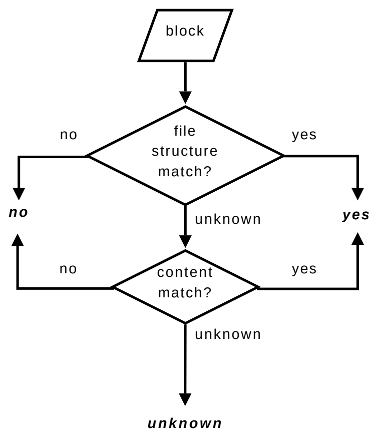


Figure 9.5: Block matches file state?

The file state uses a combination of file structure and block content based carving techniques, to increase the precision with which a block can be matched to the file it represents.

Figure 9.5 depicts how file structure and content based carving techniques are combined to check if a block matches a file state. First a check is performed using a file structure based technique. If this is able to produce a clear “yes” or “no” answer, then this answer is returned as the result of the overall check. If a file structure based technique cannot answer this question with certainty, i.e. the check returns “unknown”, then one or more content based carving techniques are used. In some cases one of the content carving techniques will be able to produce a definite “yes” or “no” answer, but the possibility remains that the final result of the check is “unknown”.

The file structure based technique in this method is comparable to the technique described in chapter 8. The block content carving technique, however, is not described in detail in this thesis, for a number of reasons:

[3]For example that none of the last 512 blocks that have been handled was assigned to that inactive state

1. First of all, for this algorithm, it is not necessary to know how the content carving technique works. When designing this algorithm a content carving “black box” was imagined which, when presented with a block and a file state, returns “yes”, “no” or “unknown” for the question whether the block matches the file state. This means that this algorithm does not depend on one or more specific content carving techniques.

2. None of the different carving techniques that may be used by this method have been developed by me. In the final implementation of this algorithm a combination of techniques was used that was either developed for the DFRWS2006 [2] or was researched and implemented by Joachim Metz as part of his work on the Revit07 carver, described in chapter 10.

3. Finally there are many different possible content carving techniques, of which most have barely been researched. Investigating and describing them did not fit the timeframe available for this project.

Main decision points

Figure 9.6 shows the total design of the block handling algorithm. As explained earlier its goal is to determine whether the block being checked is the start of a new file, part of an already detected file or not part of a file.

To do this the algorithm contains multiple points where a check may be performed to see if a block belongs to a certain file state or is the start of a new file. These can be divided into three specific checks:

1. valid first block?

2. match active?

3. #inactive matches (yes)

valid first block? checks whether a block matches one of the known file format definitions by using only a file structure based technique.

The reasoning is that the first block of a file is always clearly detectable, since it contains the header and often many more fixed strings. The chances that a block that does not mark the start of a file contains these strings in these exact positions is assumed to be negligible. Therefore a block either contains the relevant strings in the correct position, in which case the answer is “yes”, or it does not, in which case the answer is “no”.

match active checks whether a block matches one of the next possible blocks of the active state, using the method depicted in figure 9.5. The answer can be “yes”, “no” or “unknown” and the rest of the algorithm has to take these different possibilities into account.

#inactive matches (yes) checks a block to determine whether it belongs to one of the inactive file states. It uses the same techniques as *match active*, so the results for each inactive file state might be “yes”, “no” or “unknown”. It then counts the number of inactive states for which the block is a certain match, i.e. the result was “yes”. The reason for ignoring inactive file states for which the result is “unknown” is to avoid assigning incorrect blocks to a file state.

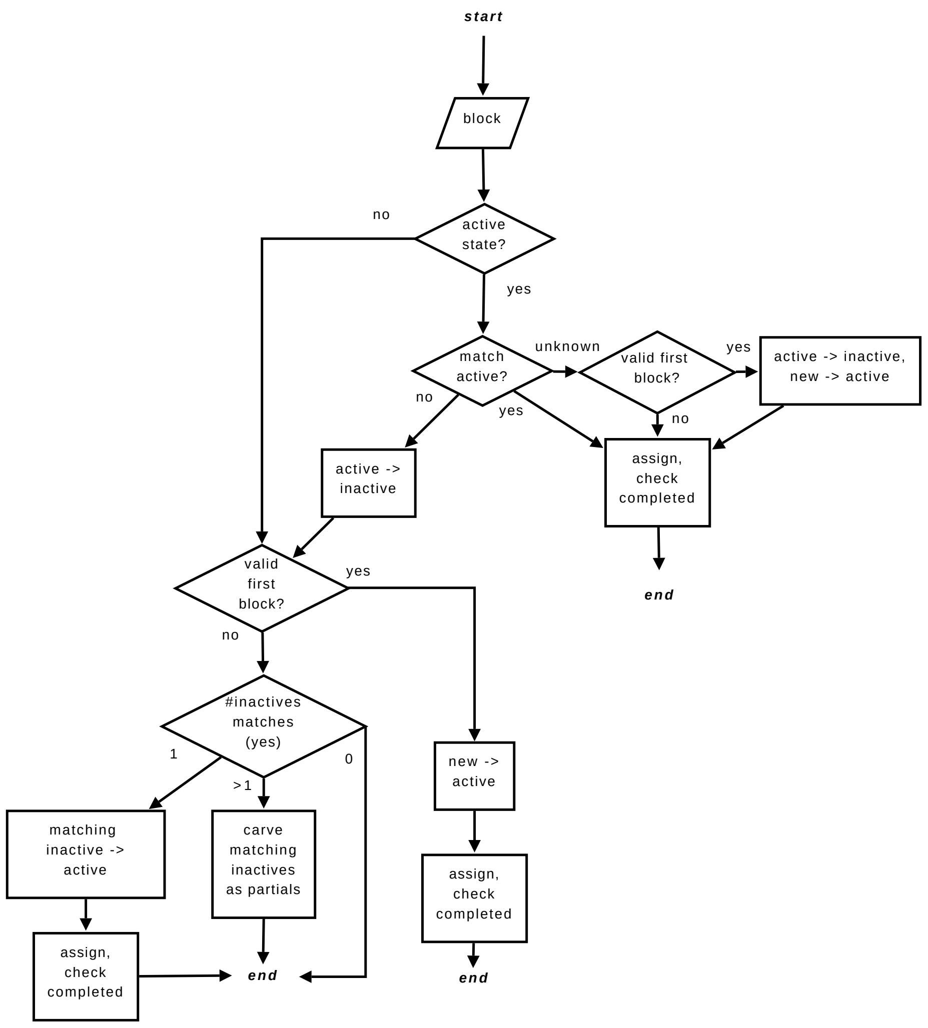


Figure 9.6: Block handling algorithm

Note that in theory a block can be a certain match for more than one inactive state, since many file types contain some degree of freedom in their definitions. For example take the JPEG file format:

The image data in a JPEG file can roughly be defined as "start of scan marker", followed by an undefined[4] amount of image data, followed by an "end of scan" marker. This basically means that after detecting a "start of scan marker" the carver starts searching for an "end of scan" marker. If there is a dataset which contains two fragmented JPEG files, both with the fragmentation somewhere in their image data, then the following can happen:

1. A carver detects the start of JPEG 1 and creates a file state
2. A carver detects the start of JPEG 2 and creates a file state
3. At some point both file states have become inactive
4. the carver detects a block that contains an "end of scan" marker
5. This block is now a perfect match for both inactive files

Main ideas behind the block handling algorithm

The block handling algorithm shown in figure 9.6 is based on two main ideas:

1. Be 'greedy' when matching blocks to the active file state, i.e. a block matches the active state unless there is clear evidence to the contrary. The reason for this is that, even in unfragmented files, there are blocks that cannot be matched for certain to that file. The decision is that it is better to risk adding a wrong block to a fragmented file than to risk rejecting a correct block for an unfragmented file.
2. The total running time of the algorithm must always be kept in mind, since if the tool is too slow then it is not viable for practical use. This was the main reason to have the algorithm perform a single linear run on a dataset and it also means that in some cases in the algorithm speed is chosen over precision.

High-level explanation of the block handling algorithm

With the main terms and decisions explained in the previous paragraphs, all that remains is explaining the overall functioning of the block handling algorithm depicted in figure 9.6. In this explanation the active file state is often simply referred to as the "active state".

The first step of this algorithm, when presented with a block, is to check if there is an active file state. If so, then it will check to see if this block matches this active state. There are three possible outcomes of this check:

1. "yes", in which case the block is assigned to the active state.
2. "unknown", in which case there are again two possibilities:

[4]Or defined in a way that the carver cannot process

   (a) The block marks the start of a new file, so the active file state becomes inactive and a new file state is created for the detected file. This new state then becomes the active state and the block is assigned to it.

   (b) The block does not mark the start of a new file, in which case the 'greedy' choice is made to assign the block to the active state. A check could have been performed to see if the block matches one of the inactive states, but these checks are relatively time consuming. Therefore the decision was made to choose speed over precision in this case.

3. "no", in which case the active file state becomes inactive and the algorithm continues as if there was no active file state in the first place.

If there was no active state or the block did not match the active state, then a check is performed to see if the block marks the start of a new file. If so, then a new file state is created, made active and has the block assigned to it. This check is relatively time efficient and provides a simple "yes/no" answer, which is why this check is done before checking whether the block matches any of the inactive states.

In case the block did not mark the start of a new file, it is matched against the inactive states. If there are no inactive states or if matching the block against the inactive states never results in a clear "yes" answer, then the block is not assigned to any file. If it matches exactly one inactive state, then that state becomes active and has the block assigned to it. The final possibility is that somehow the block matches more than one inactive state, in which case this algorithm has no way of choosing between them. To resolve this conflict the block is not assigned to any of them and the conflicting inactive states are carved as partial results.

In the previous explanation one element was not mentioned, namely the fact that "assign" also checks for completeness of the active file state. If the file is completed with the addition of this latest block, then the assigned blocks are carved and the file state is removed.

9.3.3 Scenario handling makeup

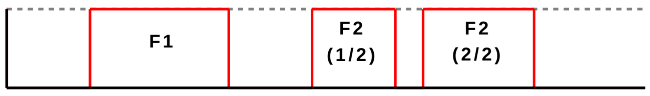
Subsections 9.3.4 to 9.3.6 describe how this carving algorithm handles the three scenarios described in section 9.2, but to fully understand these descriptions their basic makeup needs to be explained.

A scenario description is always made up of two parts: the difficulties that the scenario poses to the fragmentation handling algorithm and a description of a run of the algorithm on this scenario.

The run of the algorithm focusses mainly on the block handling algorithm, where the outcome of the different checks is depicted using an arrow. For example a positive check whether a block matches the active state leads to the following line: Match active? $\rightarrow$ yes

This is used to describe a path through the algorithm for the relevant points in the different scenarios.

9.3.4 Complete file and fragmentation



F1 is a complete and unfragmented file, so it should not pose any problems. The main difficulty in this scenario is to deal with the fragmentation of F2.

This algorithm run is based on scenario of subsection 9.2.1.

Scenario

1. The non-file blocks before F1:
   (a) Active state? → no
   (b) Valid first block? → no
   (c) No inactive states, so #inactive matches → 0
   (d) *Result:* the blocks are not assigned
2. start of F1:
   (a) Active state? → no
   (b) Valid first block? → yes
   (c) *Result:* New state for F1, becomes the active state, block is assigned
3. middle blocks of F1:
   (a) Active state? → yes (F1)
   (b) Match active? →
      i. yes: assign block
      ii. unknown: (Valid first block? → no), assign block
   (c) *Overall result:* block assigned to the active state
4. Last block of F1:
   (a) Same as for the middle blocks, but completed check after the assign returns yes. *Result:* the file is carved and the active state is removed.
5. non-file blocks: no active state, no valid first block, no inactive states, *Result:* the blocks are not assigned
6. F2(1/2): same as start/middle of F1
7. first non-file block after F2(1/2):
   (a) Active state? → yes (F2)
   (b) Match active? → no
   (c) Active state becomes inactive
   (d) Valid first block? → no
   (e) 1 inactive (F2), but 0 matches
   (f) *Result:* block not assigned

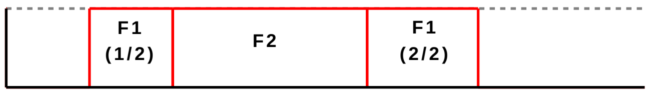
8. non-file blocks: no active state, no valid first block, 1 inactive state (F2), but no sure matches, *Result:* block not assigned

9. First block of F2(2/2):
   (a) Active state? → no
   (b) Valid first block? → no
   (c) #inactive matches → 1 (F2)
   (d) *Result:* The inactive file state for F2 becomes active again and the block is assigned.

10. Rest of F2(2/2), handled the same as the middle/end of F1.

11. non-file blocks: no active state, no valid first block, no inactive states, *Result:* the blocks are not assigned

9.3.5 Intertwined files



Though this scenario looks more complicated than the previous one, it is actually somewhat simpler to handle. The starting block of F2 can be easily recognised, so the fragmentation of F1 can be better detected than that of F2 in the previous scenario. Matching the second part of F1 to its first part is just as difficult as matching F2(2/2) to F2(1/2) in the previous scenario.

This algorithm run is based on scenario of subsection 9.2.1.

1. non-file blocks: no active state, no valid first block, no inactive states, *Result:* the blocks are not assigned

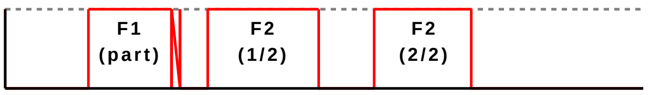
2. F1(1/2): Handled the same as the start and middle of F1 (the complete file) in the first scenario.

3. First block of F2:
   (a) Active state? → yes (F1)
   (b) Match active? → (Block not part of F1, so answer will not be 'yes', but the carver may not know for sure that this block does not belong to F1):
      i. no:
         A. Active state becomes inactive
         B. Valid first block? → yes
         C. New state for F2, becomes the active state, block is assigned
      ii. unknown:
         A. Valid first block? → yes
         B. Active state becomes inactive
         C. New state for F2, becomes the active state, block is assigned

(c) *Overall result:* New state for F2, becomes the active state, block is assigned

4. Rest of F2: Same as the middle and last of F1 (the complete file) in the first scenario.
5. F1(2/2): Same as F2(2/2) (second part of the fragmented file) in the first scenario.
6. non-file blocks: no active state, no valid first block, no inactive states, *Result:* the blocks are not assigned

9.3.6 Partial file and fragmentation



The difficulties of this scenario are the same as for the first scenario, plus two more. First of all the partial file F1 needs to be dealt with, but this is done by the main algorithm. Secondly, if F1 has not been carved as a partial file by the time the second part of F2 is handled, then its presence as an inactive file state might interfere with the matching between F2(2/2) and F2(1/2).

This algorithm run is based on scenario of subsection 9.2.1.

1. non-file blocks: no active state, no valid first block, no inactive states, *Result:* the blocks are not assigned
2. F1(part): Same as F2(1/2) of the first scenario
3. first non-file block after F1: same as the first non-file block after F2(1/2) of the first scenario
4. non-file blocks: no active state, no valid first block, 1 inactive state (F1), but no sure matches, *Result:* the block is not assigned
5. F2(1/2): Same as F2(1/2) of the first scenario
6. non-file blocks: Same as the non-file blocks between F2(1/2) and F2(2/2) in the first scenario.
7. First block of F2(2/2):
   (a) Active state? → no
   (b) Valid first block? → no
   (c) Two inactive states, the block is a correct match for F2(1/2), but it might theoretically be a match for F1(1/2) as well.
      i. 1: (Matches F2(1/2)) → *Result (correct):* The inactive file state for F2 becomes active again and the block is assigned.
      ii. 2: (Matches both F1(part) and F2(1/2)) → *Result (incorrect):* No way for the carver to make a certain correct choice, the safe choice is to carve both as partials. After this there are no more inactive states.

(d) The rest of this scenario description continues as though a match was made between the block and F2(1/2)

8. middle and end of F2(2/2): Same as middle and end of F2(2/2) in the first scenario.

9. non-file blocks: no active state, no valid first block, 1 inactive state (F2), but no sure matches, *Result:* the block is not assigned

At some point the carver must decide to carve F1 as a partial. The amount of time that inactive files are retained can effect the results of a carver. If the time is short, then the first part of a fragmented file might be carved before the second fragment is handled, so the file cannot be fully carved. On the other hand, the possible match to both F1(part) and F2(1/2) would have been avoided if F1(part) had been carved before the start of F2(2/2).

9.4 Algorithm weak points and improvements

The algorithm described in this chapter was implemented by Joachim Metz in the Revit07 file carver, as will be described in chapter 10. During this implementation and the further development of Revit07, both his and my knowledge of file formats and different carving scenarios increased significantly. Because of this new knowledge, a number of weaknesses and incorrect assumptions were identified in the fragmentation handling algorithm. This section described these weaknesses and their (partial) solutions.

9.4.1 Algorithm weak points

In some situations the information available to the carver might be insufficient to guarantee correct decisions. These are weak points in the algorithm, which may lead to a number of incorrect carving results. To explain these weak points and the possible results, the scenario descriptions from the previous section are used again.

Weakness 1

The first point where a wrong decision might be made can be seen in scenario 1, at the end of F2(1/2). In case of fragmentation with non-file data, the carver may not always be able to decide for certain that the non-file block is not a part of the active state F2(1/2). In this case “Match active?” returns “yes” or more likely “unknown” and “Valid first block?” will always return “no”, since the blocks contain non-file data. This means that one or more blocks may be added to the active state that should not have been added.
Results:
There are three possible outcomes of a wrong decision:

1. None of the non-file blocks between F2(1/2) and F2(2/2) are recognised as not being part of F2(1/2) and F2 is carved as an unfragmented file. This is possible, for example, if the fragmentation occurred in the image data of a JPEG file. The reason for this is the same as the reason that a

block may match two inactive file states, as explained earlier. In this case the extra blocks are interpreted as part of the content and a corrupt file will be carved.

2. The fragmentation was detected, but not immediately, so extra block(s) were incorrectly assigned to F2(1/2). This leads to the two other possible outcomes:
    (a) F2(2/2) can somehow still be matched to F2(1/2), in which case a corrupt file will be carved.
    (b) F2(2/2) can no longer be matched to F2(1/2), in which case F2(1/2) will (eventually) be carved as a partial file. F2(2/2) will probably not be carved, since the algorithm cannot use the information from F2(1/2) to determine where it starts.

Weakness 2

F2 in scenario 1 highlights another possible weakness in the algorithm, specifically when deciding whether the first block of F2(2/2) is a match for F2(1/2).

This check has been made rather strict to avoid adding wrong blocks to F2. This means that F2 will only be made active again and the block assigned to it, if the carver is certain that the first block of F2(2/2) matches F2(1/2).

The carver will not always be able to answer this question with certainty, especially if the fragmentation occurred in a part of the file where little (structural) information is available. In this case the second part of a fragmented file may not be matched to the first.

Results:
F2(1/2) will be carved as a partial file (eventually). F2(2/2) is not carved, since this algorithm cannot deal with partial files that are not the start of a file.

9.4.2 Incorrect assumptions

When extending the file formats supported by Revit07, a number of file formats were encountered that did not fit the original notion of what a file format looks like. This means that two assumptions from section 9.3 turned out to be incorrect in some cases:

First block of a file is always clearly detectable This is not the case for text files, in which there is no clear structure that can be used to define a starting block. Another file type that may pose problems is MP3, which starts with a very small header, a number of bytes that define the size of the following section and then a (large) amount of file structureless data. This means that the first block of an MP3 file may contain only a few bytes of fixed data that can be matched against and a block that is not the start of an MP3 file will sometimes match the beginning of the MP3 file format definition.

File completion can be easily checked An MP3 file is made up of one or more MP3 frames, which in themselves are also valid MP3 files. This means that if a valid MP3 file is divided in two parts on a frame boundary, then the result is two valid MP3 files. This also means that if the carver detects the end of an MP3 frame, then this *may* be the end of the MP3 file, but it does not have to be. Text files pose a comparable problem, they have no fixed file structure and therefore no way to check for completeness.

9.4.3 Revit07 solutions to the different weaknesses

The approaching DFRWS2007 deadline meant that there was not enough time to redesign the fragmentation handling algorithm to deal with the discovered weaknesses and incorrect assumptions. Joachim Metz did however implement a number of (partial) solutions in Revit07:

Weakness 1: Start of fragmentation not detected If there are doubts about assigning a block to a file state, perform a provisional assignment. Once the carver encounters a block that confirms the provisional assignment was correct then it can be made permanent. If a block is encountered that proves that the provisional assignment was incorrect, then it is undone.

Weakness 2: Fragments not matched to each other Extend the "#inactive matched" check to also allow for an "unknown" match to lead to a provisional assignment. This is possible due to the solution to weakness 1.

Incorrect assumption 1: First block not clearly detectable Distinguish between *certain* new file states and *possible* new file states. Extend the algorithm to take these new possibilities into account.

Incorrect assumption 2: File completion not easily checked Mark a file as possibly completed, but only carve it when no new blocks can be added to it.

These solutions are not able to avoid and detect all problems, but they still manage to greatly improve the effectiveness and precision of the fragmentation handling algorithm.

Chapter 10

Revit07

This chapter describes the Revit07 file carver, which was developed by Joachim Metz at Hoffmann to participate in the DFRWS 2007 file carving challenge[1].

The reason that this carver is of interest to this project is threefold:

1. Revit07 implements a separation of file format definitions and a single carving algorithm, that is partly based on the design done for MultiCarve as described in chapter 8
2. The fragmentation handling algorithm of chapter 9 was developed specifically for use in Revit07
3. Revit07 was combined with the validator described in chapter 11 to form a carving framework that accomplishes the goals stated in chapter 9.4.

Revit07 is a file carver that uses a combination of file structure and content based carving techniques. It is the successor of the Revit06 file carver submitted by Hoffmann for the DFRWS 2006 carving challenge[2]. Revit06 was a file structure based carver that already had an extensive file format configuration file, but had not yet achieved separation of the carving algorithm from the file format definitions.

Development of Revit07 was done parallel to the development of MultiCarve, so Joachim Metz and I had frequent discussions on different design issues. This means that many of the ideas on the separation of file format definitions and a single carving algorithm, as developed for MultiCarve can also be found in Revit07. Note however that the implementation of this separation is very different, mainly due to two reasons:

1. Revit07 is based on Revit06, so many internal structures were already defined.
2. Revit07 is written in C[3] and uses lex and yacc [1] for parsing its file format configuration. This means that the MultiCarve file definition was impractical to implement for Revit07 and has been implemented differently.

[1]`http://dfrws.org/2007/challenge/`

[2]`http://dfrws.org/2006/challenge/`

[3]http://www.open-std.org/jtc1/sc22/wg14/

10.1 Revit07 and MultiCarve

Even though development of Revit07 and MultiCarve started roughly at the same time, Revit07 was ultimately developed for a much longer period. This was due to the decision to focus development efforts on one program, because of the DFRWS 2007 deadline. Since Revit07 was more mature than MultiCarve at that time, the choice was made to focus all efforts on the Revit07 development. This also means that the current version of Revit07 contains a number of key improvements over MultiCarve:

1. Revit07 supports linearly fragmented files, by implementing (and extending) the algorithm described in chapter 9.

2. Revit07 supports a number of content carving techniques, as described in subsection 10.1.1

3. Revit07 has the ability to detect file corruption in specific cases, as described in subsection 10.1.2

10.1.1 Content carving techniques

Joachim Metz implemented a number of content carving techniques in Revit07 to help it deal with fragmentation. The ideas behind these techniques came mostly from three sources:

1. The winning submission of the DFRWS 2006 challenge [2]

2. A pseudo random number sequence testing tool called "ent"[4]

3. A book on statistical natural language processing[6]

The following three techniques currently show the best promise:

Character counts: The number of times each different byte occurs in a block is counted. This can be used in multiple ways:

1. Some bytes may not occur in specific parts of some file types. A block that contains these bytes cannot be fully contained in those parts of a file.
2. If a block contains 50% zeroes and 50% text characters, then it is almost certainly a block filled with Unicode text.

Entropy Information entropy is a statistic that provides insight into the information density of a range of data. In some parts of some file types, for example in the image data in a JPEG file, this value stays relatively constant. A sudden unexpected drop in entropy is therefor usually a sign of fragmentation.

[4]`http://www.fourmilab.ch/random/`

10.1.2 Detecting corrupted files

File format definitions sometimes contain redundant information. For example two things might be known about a data structure in a file:

1. The size
2. Its internal elements

The carver can then choose to check both the internal elements and the size of the structure. If there is a discrepancy between these two, for example because the combined size of the internal elements is larger than the defined size of the data structure, then the file is most probably corrupted.

10.2 Improvement goals handled by Revit07

By implementing the techniques described in chapters 8 and 9, as well as numerous other extensions and adaptations, Revit07 (partially) meets the following improvement goals set in chapter 7.1.

Support for many file types: Due to the separation of file format definitions and the carving algorithm, file formats could be added while the carving algorithm was still under heavy development.

Fragmentation support: Revit07 implements the fragmentation handling algorithm of the previous chapter, combined with support for a number of content carving techniques. Together this should be able to produce good results on linearly fragmented files.

Deal with corrupted files: By using redundant information in the file format definitions, Revit07 is able to detect corruption in some cases.

Unfortunately Revit07 cannot detect all forms of fragmentation and file corruption. This means that the possibility still exists that False positives are produced, so validation of Revit07's carving results is still required.

Chapter 11

Validator

As described in the previous chapter, Revit07 focusses on meeting three of the four improvement goals stated in chapter 7, namely the support for many file types, better fragmentation handling and support for carving partial and fragmented files. However, this leaves one final improvement goal, the detection of False positives.

The chapters on file fragmentation handling and Revit07, i.e. chapters 9 and 10, highlight why this detection of False positives is required. In some situations the information available to a carver is simply insufficient to guarantee correct carving results.

The decision was made to perform the detection of False positives in a separate validator program, so it could be developed and used independent of Revit07.

To fully automate the task of carving and validating files, the carver and validator are combined in a very simple framework, as shown in figure 11.1.

The carver investigates a dataset and produces a number of carving results. These results are then tested by the validator, which divides them into the following three groups:

1. Valid files: The validator has detected no problems with these files.

2. Viewable files: The validator has detected that there were problems with these files, but has determined that they are viewable using a corresponding tool.

3. Unviewable files: The validator has determined these files cannot be viewed using a corresponding tool.

The post-processor combines the output of the carver and the validator as follows:

Carver	Validator	End result
Complete	Valid	Valid
Complete	Viewable	Viewable
Complete	Unviewable	Unviewable
Partial/Corrupt	Valid/Viewable	Viewable
Partial/Corrupt	Unviewable	Unviewable

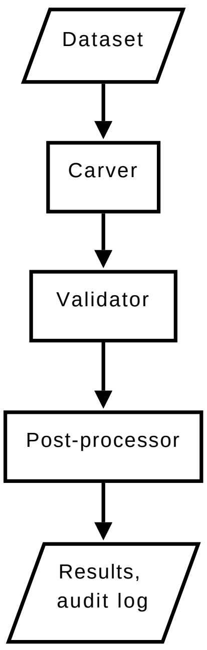


Figure 11.1: Carving framework

It then moves the carved files into a valid/viewable/unviewable directory and produces a combined audit log for both the carver and the validator.

The main question to answer related to these groups is:
Why divide the results into valid, viewable and unviewable, instead of just into valid and invalid?

If the sole purpose of the validator was to increase carving quality scores, then this would indeed be enough, but the validator also has a secondary objective: find all carving results that are viewable using a corresponding tool. These files are of interest in a forensic investigation, because the information in them is readily accessible and can therefore be processed further.

With the grouping described above both goals can be met:

- For the quality scores the valid files are presented as Positives, whereas (un)viewable files are interpreted as Known false positives.
- For further processing the valid and viewable files are combined to form the total group of viewable files.

11.1 Validator framework

The validation program shown in figure 11.2 is a modular framework where separate modules can validate one or more file types and the main validator handles command line interaction and combines the results to create a validation audit log.

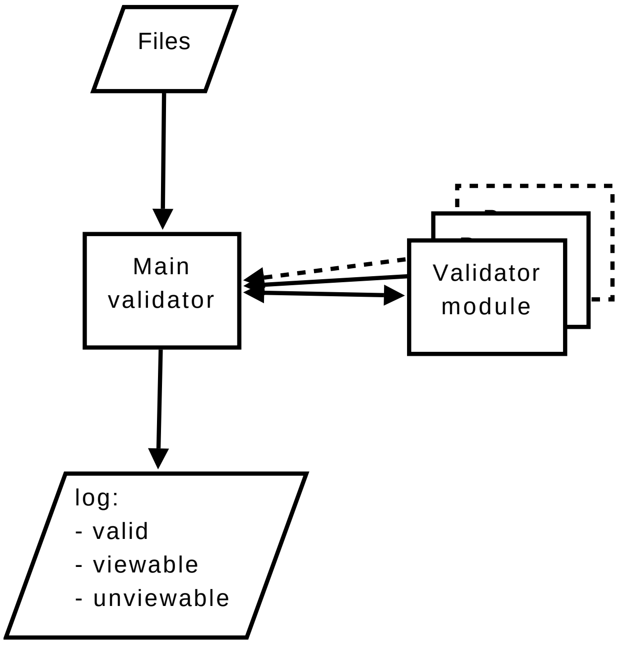


Figure 11.2: Validator framework

Each validation module is based on a different tool. For example most image file formats are validated using the “convert” tool. The validity of a file can be determined using one or more of the following criteria.

- Some tools will crash or return an error code on invalid inputs.
- Tools often produce output that can be analysed for error messages.
- Some file types are validated by trying to convert them into a different type. In case of an invalid input the target file may not be produced.

Appendix C provides a list and description of the programs used in the different validator modules and the files they support, as well as a global description of their validation approach.

The program interfaces analyse the tool results and provide the results to the main validator.

11.1.1 Validator problems

The main reason for using existing tools is that the validity or viewability of a file is defined as to whether a corresponding tool can (correctly) open it.

The use of existing tools enables the creation of a validator that can handle many file types in a short development time. This also means that validation of a new file type can usually be added rather easily, by adding a new module or by extending an existing module. This does however depend on finding a tool that returns enough useful information when opening a file, to be able to check whether the file was opened successfully.

Unfortunately there is also a major downside to this approach, namely that most tools were not created with file validation in mind. They are often able to deal with corrupted or partial files without producing error messages.

Another problem is related to the file type being validated instead of the type of validation. Specifically the validation of html and (Unicode) text files is much more difficult than that of other file types.

The problem with html files is that there are multiple standards, but that most html files found on the Internet conform to none of them. This means that html files will rarely be marked as valid.

With text files the problem is of a very different nature. Here the problem is that there is no real definition of a valid text file and therefore no way to accurately test text validity.

11.2 Validation examples

There are many different ways to perform file validation using standard tools. This section provides two examples, to give an idea of the different possible approaches.

For some file types there are standard tools that are able to test the validity of a file. For example the validator used for "zip" files works as follows:

1. The zip file is checked using `unzip -t ZIPFILE` and the exit status of `unzip` is examined.
2. If `unzip` exited without error, i.e. the exit status is 0, then the file is *valid*
3. If `unzip` exited with exit status 1 or 2, then there is something wrong with the zip file, but it is considered *viewable.*
4. Any exit status higher than 2 leads to the result *unviewable.*

The validation of a zip file is rather straightforward, but many other file formats require more effort. An example is image files, which are hard to validate without human interaction. There are plenty of tools that are able to view different image file types, but few provide enough feedback to be used for validation. Another issue is that it is preferable if the validator works without depending on the availability of a graphical environment. The chosen solution was to use the "convert" tool from the ImageMagick toolset[1]. Validation is done as follows:

1. `convert FILENAME TMPFILE`
2. If `convert` produced no error messages, then the image file is *valid* and the temporary file is removed.
3. If `convert` produced one or more error messages, but the temporary file was created, then is was able to extract enough relevant information. In this case the result is *viewable.*
4. If `convert` produced one or more error messages, and the temporary file was not created, then the file is *unviewable.*

[1]`http://www.imagemagick.org`

Chapter 12

Quality of the carving framework results

This chapter tests the quality of the newly developed carving framework and compares this quality to the goals stated in chapter 7.

To get an idea of the effectiveness of the different techniques, the carver has been tested in a number of different configurations, as described in section 12.1. Section 12.2 gives the actual scores and analyses the effectiveness of the different techniques. It also compares the results to relevant results of the tools tested in chapter 6. Finally section 12.3 analyses to what degree the goals stated in chapter 7 have been met.

12.1 Carving framework configurations

To test the effectiveness of the different techniques combined in the Revit carving framework, this framework has been tested in the following four configurations:

1. Revit-none: Validation and content based carving have been disabled, so the framework acts like a pure file structure based carver.

2. Revit-val: Content based carving has been disabled, so the framework acts like a file structure based carver with added validation.

3. Revit-content: Validation has been disabled, so the framework acts like a combined file structure and content based carver, but which does not validate its results.

4. Revit-all: All techniques are enabled to create a combined file structure and content based carver, with file validation.

This provides insight into the contribution the different techniques made to the overall framework quality.

12.2 Scores on the different datasets

This section shows the scores that the different framework configurations get on the datasets described in chapter 5.1. Each table also shows the most interesting results from the other tools tested. In case of the 11-fat and dfrws 2006 images, these are the top two results, in case of the 12-ext image this is the top two results and the best result that did not use ext2 support.

For each table an interpretation is given of the most relevant results and a comparison is made between the carving framework and the other tools tested.

To keep these tables readable and relevant, not all information has been depicted. This sometimes means that an interpretation is based not just on the values in these tables, but also on information from the full tables in appendix A. Also the Supported Recall column is not shown, since the carving framework supports all files in the datasets. This means that all the results depicted have the same scores for both Carving Recall and Supported Recall.

Configuration	Carving Precision	Carving Recall	cF Measure	MB per second
Revit-all	0.966	0.935	0.95	5.17
Revit-content	0.966	0.935	0.95	5.64
Revit-val	0.897	0.937	0.92	5.17
Revit-none	0.897	0.937	0.92	5.64
Recover My Files	0.923	0.975	0.95	2.48
PhotoRec	0.857	0.931	0.89	15.5

Table 12.1: Carving framework quality score for 11-fat

Interpretation of the results on the 11-fat dataset by Nick Mikus (table 12.1):

- Revit-none acts like a pure file structure based carver, which gets good results on this image. It does produce a False positive result, however, as can be seen in the appendix.
- Revit-val shows that the validator is not perfect and that it does not detect all False positives. Therefore the precision score is equal to that of Revit-none.
- Revit-content and Revit-all show that the added content carving techniques manage to correctly carve the Unknown false positive as a Positive, increasing the carving_precision to near perfect. Adding validation makes no difference in the scores, since there are no more Unknown false positives.
- Revit-content and Revit-all both have the same F-measure score as Recover My Files, the tool that performed best in the initial tests. The difference is that Revit has slightly better carving_precision and Recover My Files has slightly better carving_recall.

Interpretation of the results on the 12-ext2 dataset by Nick Mikus (table 12.2):

- Revit-none confirms the idea that pure file structure based carvers have great difficulty in carving fragmented files.

Configuration	Carving Precision	Carving Recall	cF Measure	MB per second
Revit-all	0.778	0.986	0.87	6.89
Revit-content	0.737	0.986	0.84	9.54
Revit-val	0.533	0.753	0.62	7.75
Revit-none	0.4	0.753	0.52	11.27
PhotoRec-ext2	1.0	1.0	1.0	12.4
Foremost-ext2	0.6	0.988	0.75	11.27
Foremost	0.455	0.7	0.55	11.27

Table 12.2: Carving framework quality score for 12-ext2

- Revit-val performs slightly better, since all False positives are detected. Note that the revit framework in this configuration already performs better than any of the carvers that did not use the ext2 information, even though the overall results are still only mediocre.
- Revit-content shows the positive effect of content carving on fragmented datasets. There is a clear increase in both carving_precision and carving_recall and the overall cF measure score is even better than Foremost with ext2 support enabled.
- Adding validation to Revit-content creates a slight improvement in carving_precision. All in all the cF measure for Revit-all is 58% better than that of Foremost, which had the highest score of any tool not using explicit ext2 support.

Configuration	Carving Precision	Carving Recall	cF Measure	MB per second
Revit-all	0.731	0.98	0.84	0.83
Revit-content	0.613	0.98	0.75	1.07
Revit-val	0.585	0.754	0.66	0.91
Revit-none	0.464	0.754	0.57	1.2
PhotoRec	0.633	0.875	0.73	1.78
Recover My Files	0.37	0.83	0.51	1.78

Table 12.3: Carving framework quality score for dfrws2006

Interpretation of the results on the DFRWS 2006 dataset (table 12.3):

- Revit-none performs very mediocre, due to the high fragmentation in this image. This means that it produces a high number of both False negatives and (Known/Unknown) false positives, leading to a low carving_recall score and a very low carving_precision score.
- Revit-val manages to turn all Unknown false positives into Known false positives, making a big improvement in carving_precision.
- Adding content carving makes a big difference on this image in both carving_recall and carving_precision. Carving_recall becomes almost perfect,

mostly due to a huge drop in False negatives. The 7 added Positives mean that the carving_precision is also much better. Unfortunately there are still 10 Unknown false positives and 4 Known false positives, which keeps the carving_precision score down to a very mediocre 0.66.

- Revit-all truly shows the positive effect of combining file structure based carving, content carving and validation, with a near perfect carving_recall and a very reasonable carving_precision. The final cF measure score is 15 percent higher than the best score of the other tested tools.
- Note that carving speed (MB per second) is lower than would be preferred for a tool to be usable in practice. This is because of the complexity and high number of files in the dataset.

Both the 12-ext2 and the DFRWS 2006 image show that the revit framework can achieve very high carving_recall rates, but the extra carved (Known/Unknown) false positives have a negative effect on the carving_precision. Improving the framework to correctly carve these results would have a very significant impact on this carving_precision score and make the overall cF measure score even better.

The reliability of the framework on all datasets, as measured by the supported_recall score, was between very good and almost perfect.

12.3 Does the framework meet the improvement goals?

Chapter 7.1 stated a number of improvement goals, based on the results of the tested tools. This section investigates how well our carving framework has met these goals.

The stated goals can be divided into two groups:

1. Higher carving_recall
2. Higher carving_precision

The following two subsections repeat each of these goals and investigate how well the carving framework (Revit-all) fulfilled them.

12.3.1 Higher carving_recall

Support many file types

The framework supports 24 different file types, enabling it to carve for all the different file types in the dataset. The file format definition is separate from the carving algorithm, so support for new files can be added with relative ease. The main reason for this goal was to minimise the amount of Unsupported false negatives, and since this number is 0 on all datasets this goal is clearly met for these datasets. Note that different investigations call for different supported file types, so increasing the number of supported file types is always a good idea.

Carve partial/corrupted results as Known false positives.

Revit07 has the ability to carve partial and corrupted files and save them in a separate directory. The validator then marks them as Known false positives. The main reason for this goal was to increase the carving_recall score by decreasing the number of Supported false negatives. The fact that on all datasets the Supported false negative score is less than one, means that of all files in all the datasets at least a part has been recovered. Therefore this goal has clearly been met.

The overall goal of increasing carving_recall has definitely been met. On the 11-fat dataset the carving_recall is very good, but so was the score for all other tools (except FTk) on that dataset. The real improvement can be seen on the other two datasets, where the carving_recall is almost perfect.

12.3.2 Higher carving_precision

Mark False positives as Known false positives

Increase the carving_precision by turning Unknown false positives produced by the carver into Known false positives[1]. In the carving framework this is the task of the validator. On 11-fat the carver (when using content carving) did not produce any Unknown false negatives, so the validator made no difference. On the 12-ext2 dataset the validator managed to mark one of the two Unknown false positives as Known false positives, but missed the other one. On the dfrws2006 dataset its score was perfect, it managed to catch all 10 Unknown false positives.

All in all this goal has been met almost perfectly for these datasets, but the use of general purpose tools for validation purposes means that there is always a certain chance that an Unknown false positive will be missed.

Better fragmentation handling

The main field of improvement, mostly for carving_precision, but also affecting carving_recall, is better fragmentation support.

First of all the current carving framework will never be able to carve all fragmented files, since it only supports linear fragmentation, so this paragraph only tries to determine how well it supports linearly fragmented files.

There is no way to read the level of fragmentation support directly from the results, but a reasonable indication can be inferred in a number of steps.

1. First of all, only the 12-ext2 and dfrws2006 datasets include fragmented files and all fragmentation in these images is linear. Comparing the difference between Revit-content and Revit-none on all datasets shows a precision increase of 7,7% on 11-fat, compared to an increase of 84% and 33,8% on 12-ext2 and dfrws2006 respectively. This indicates that adding content carving most likely translates mainly into the framework's ability to handle fragmented files. This observation is strengthened by the knowledge that content carving was added to the framework for that exact reason.

2. On 12-ext2 the carving_precision of all tools that use pure carving techniques is below 0.46, this is including revit-none. Revit-content and its

[1]This depends on β being greater than 1

better fragmentation support manage to get a carving_precision of over 0.73, which is a huge increase. A comparable effect can be seen in the carving_recall score. This seems to indicate that Revit-content, and therefore Revit-all, can handle the fragmentation due to the indirect blocks much better than the other pure-carving tools.

3. On the dfrws2006 dataset the Revit framework gets real competition when it comes to fragmentation handling, in the form of PhotoRec. This is mostly because the Revit-content does not achieve the same level of carving_precision as on the other two datasets. One possible reason for this is that content carving is less precise on this image, since it is based on a raw file filled with random blocks. Carving_recall still increases dramatically when content carving is enabled. Based on the results of this dataset no clear difference in fragmentation support can be detected between the Revit framework and its direct competitor PhotoRec.

Combining the analysis of the 12-ext2 and the dfrws2006 dataset leads to the following assessment of the increase in fragmentation support:
Since the Revit framework handles fragmentation as well or better than the other tools on the dfrws2006 dataset and *much* better than all other tools on the 12-ext2 dataset, the goal of increasing fragmentation support has been reached.

Chapter 13

Supporting tools and dataset analysis

Not all tools that have been created for this project were created to directly influence the quality of carving results. Some tools were created to provide better insight into specific carving techniques or to help find problems in (the results of) other tools. The best example of this is the MultiCarve tool described in chapter 8, but there are two other tools that are worth mentioning.

13.1 Block content analysing tool

First of all a "block content analysing" tool was created to try and find information that could be used in a content carving technique.

This tool is based on the fact that Revit07 is able to calculate and export a number of statistics for each block of data it processes. These statistics are based on the "ent" pseudo random number sequence tests[1].

The block content analysing tool takes these statistics and marks sudden increases or decreases in their values. The idea behind this is that these changes will often be the result of changes in the layout of the dataset, like the start or the end of a file, but also as a result of fragmentation.

The result of the analysis is a list of possible points of interest. These are manually investigated to see if there is indeed something of interest, like fragmentation, happening there.

The main goal of this approach was to establish rules that could be used by a content carver. Unfortunately the data under examination was too complex to manually recognise the patterns that would form the basis for this rule. Another problem is the lack of real-world datasets that can be used to test the different rules. Both datasets by Nick Mikus contain very little files and the datasets by the DFRWS are based on random blocks, which leads to many false interesting points.

Both a larger number of real-world datasets and an automated means of generating likely content analysis rules are required for this tool to be truly useful.

[1]`http://www.fourmilab.ch/random/`

13.2 Dataset topology

If a carver does not get perfect results on a dataset, then the main question is usually: Why? One method to gain more insight into the problems that may have occurred is to compare the results of the carver to the layout of the dataset. The topology tool is one of the ways that this comparison can be performed.

The topology tool is able to combine multiple sources of layout information in a single text-based topology of a dataset.

In the following example a part of the DFRWS 2007 layout is shown, in which the location of a carved quicktime (qt/mov) file can be compared to the location of that file as provided by the layout of the dataset.

```
-------------|---------------||
             | 277 blks      ||
-------------|---------------||
011fe600     | 011fe600      || start_qt(V)     | start_1.mov(#)  |
-------------|---------------|| vvvvvvvvvvvvvvv | ############### |
             | 16660 blks    || vvvvvvvvvvvvvvv | ############### |
-------------|---------------|| vvvvvvvvvvvvvvv | ############### |
01a21000     | 01a21000      || vvvvvvvvvvvvvvv | end_1.mov(#)    |
             | 01a211ff      || end_qt(V)       |
-------------|---------------||
```

The first column depicts the start offset of a 512 byte block (in hexadecimal notation), whereas the second column has a double function. It either depicts the exact offset of a point of interest or the number of blocks that contain no change in information. In the example there are 277 non-file blocks, after which at `011fe600` both the carver (start_qt) and the official layout (start_1.mov) agree that a quicktime file starts. Then there are 16660 blocks of quicktime file, after which according to the layout the file ends at `01a21000`, whereas according to the carver the file ends at `01a211ff`. If the layout information is correct, then this means that the carver has carved more data than it was supposed to do. This information can then be used to more specifically look for errors in the carving algorithm.

Besides this example there are a number of other ways this topology tool can be used. For example the carving results of different tools on an image with an unknown layout can be combined to get an indication of the actual layout. Or the interesting points from the previous section can be combined with the layout of a dataset to see if these points of interest indeed coincide with changes in the layout.

Chapter 14

Conclusion

This project had a combined goal:
"Define meaningful criteria and a method to measure the quality of carving tools. Based on the quality of current tools, develop a carving tool which achieves better results"

The first part of this project, described in chapters 3 to 6 focussed on determining how carving quality can be defined and measured. The rest of the project was about gaining a deeper understanding of the intricacies of file carving and creating a better carving tool.

Section 14.1 analyses how well the first part of the goal, namely measuring the quality of carving results, has been met. Section 14.2 analyses whether the newly developed carving framework produces carving results and states a number of possible further improvements.

14.1 Measuring the quality of carving results

During the course of this project a method was developed which provides a clear insight into the overall quality of a tool. Besides the overall quality, the method is also able to give specific information about the following three quality aspects of a tool:

1. How good is a tool at recovering files, i.e. how much relevant data is missed?
2. How precise is a tool, are the recovered files correct?
3. How reliable is a tool, how much of the data that it claims to be able to recover does it actually recover?

Using the scores for these three quality aspects it is possible to identify concrete areas need improvement.

This method also provides the ability to adjust the relative importance of the individual quality aspects when determining the overall quality. This can be useful since the quality of a tool is related to the specific goals of the user of that tool.

There is, however, also a downside to this way of measuring quality. This method relies on the availability of datasets with a known layout. These datasets are man made, so they may not accurately reflect the data that can be found on the average computer system. The future work chapter provides a possible approach to overcome this problem.

Another problem of measuring quality based on the carving results is that some quality questions cannot be answered directly, but have to be inferred from the available data. An example of this is determining whether the Revit framework handles fragmentation better than the competing tools.

In conclusion, the quality measuring method clearly meets its goal, but there are still possible areas of improvement.

14.2 Improving the quality of carving results

The improvement goals stated for the new carving framework can be divided into two categories: improve carving_recall and improve carving_precision.

Improving carving_recall

Carving_recall is a measure for how good a tool is at recovering files. To improve this as much relevant information as possible must be extracted from a dataset. The newly developed carving framework is much better at recovering files than the other tools tested, as section 12.3 shows. Therefore this area shows a clear improvement.

Improving carving_precision

Carving_precision is a measure of the quality of the results produced by a carving tool. The new carving framework produces a much lower number of incorrect results than existing tools, but there is still plenty of room for improvement.

Improving content carving support can lead to even better fragmentation support, which mostly increases carving_precision, but may also improve carving_recall scores.

Secondly the use of generic tools for file validation may be an efficient choice, but special purpose validators could probably produce more reliable results.

Chapter 15

Reflection

As the introduction stated, carving is an area of forensics that has been somewhat neglected in the development of new tools. Only recently has interest in this technique increased, for a large part due to the 2006 and 2007 DFRWS forensic challenges. This new period of interest has led to new insights into carving techniques, but even more so, highlighted the many aspects of carving that are still unknown.

The upside of this great number of unknown aspects is that there is very much to be explored. For every question answered, two more interesting questions come up. Some of these questions had to be answered directly to further this project, the rest of them are interesting for future work. It never becomes boring, there is always more to discover, but this also means that there is very little solid information to hold on to.

To deal with this lack of solid information it was vital to get hands on experience with carving by creating a carver myself. This provided me with a great deal of insight into both the possibilities and the difficulties of creating a carving tool. It also taught me that I much prefer designing a complex tool, over solving all the implementation problems related to it.

Many different aspects of carving were investigated during this project and every new insight led to a (minor) change in the direction of the project. Therefore the hardest part of this project for me was not the main project itself, but describing all the separate parts as a logical whole in this thesis. This took much more time and effort than I had expected, but in my opinion the extra effort has led to a thesis that accurately describes how all parts of this project are related.

One event that is mentioned on multiple occasions throughout this thesis, yet was not used for any measurements on carving quality, is the DFRWS 2007 carving challenge. The reason for this is that the layout information was not available in time to accurately investigate and include in this thesis. However, this part of the thesis provides the possibility to describe some of the results without having to go into too much detail. First of all, our carving framework was not one of the winners of the challenge. Initial tests show that the carver carved one or two blocks too many on about 12 files, leading to a huge reduction in the number of Positives. The reason for this is still under investigation. Regardless of these results there was again a discussion about the way in which the quality of the different tools was rated. This shows the need for a universally accepted way for measuring carving tool quality.

Chapter 16

Future work

During the course of this project a great deal of unknown aspects about the field of carving were encountered, as well as possible ways to improve this situation.

This chapter describes a number of areas in which more work can be done to further develop the field of carving. These have been divided into three sections:

1. Fundamental areas of improvement
2. Possible improvements to the quality testing method
3. Possible improvements to the carving framework

16.1 Fundamental areas of improvement

Fragmentation of real-life datasets The major challenge for a carver is how to handle different forms of fragmentation. One of the major obstacles in this is that very little is known about the types and amount of fragmentation that occurs on computers that are used in different situations. A good start for this has been made by professor Garfinkel in one of his papers [4], where a large number of disks was investigated for the amount of fragmentation present. This paper gives a first insight, but still leaves many questions unanswered:

1. How often does nonlinear fragmentation occur compared to linear fragmentation?
2. What types and amount of fragmentation occur for different combinations of file systems and operating systems?
3. What is the impact of defragmentation of the allocated data on the unallocated data?

Increase the number of datasets with a known layout One of the main objections that can be had against the quality testing method developed in this project is that it has only been fully tested against three datasets. Unfortunately, until the very end of this project, that was all there was available. Adding the DFRWS 2007 dataset leads to four datasets, which is still much too few. Another problem with these datasets is that they have all been hand-crafted for specific purposes, so they may not reflect real-life situations and can therefore provide a skewed view of the quality of different tools. Somehow the number of available real-life datasets with a known layout needs to be greatly increased. Simson Garfinkel is currently writing a paper [5] which also highlights this lack of realistic datasets for carving and other forensic purposes. Section 16.2 of this chapter provides a possible method for creating these datasets.

Investigate content characteristics of different file types The biggest obstacle to be overcome when using content carving is to find the meta information to differentiate one part of a file from another part of a (different) file. At this time very little is known about the content characteristics of different parts of different files. Analysing a large number of files of different types could provide the information needed to greatly increase the accuracy of content carving techniques.

16.2 Improvements to the quality testing method

The main problem with this quality testing method[1] is the dependency on datasets with a known layout, of which very few exist. This problem might be solved in two ways:

Extend the quality method to provide meaningful information when tested on unknown datasets.

Increase the availability of known datasets, for example by using different file recovery techniques and tools on a dataset with an unknown layout and combine the results to create a (likely) layout for this dataset.

If either of these methods is successful, then the problem is transformed to finding useful datasets for which the layout is not (necessarily) specified.

Another way to improve the quality testing method, as suggested by Joachim Metz, is to extend it to provide a way to indicate how well a tool is able to carve a specific file type. Unfortunately, to give an accurate assessment of the quality of a tool for a specific file type, a carver needs to carve many files of that type. At this time the datasets needed for this are simply unavailable.

One final improvement to the quality measure would be to have it reviewed and hopefully accepted by forensic researchers worldwide. If this happens then the mapping of descriptive to normative quality scores can also be updated to reflect a more widely supported norm.

16.3 Improvements to the carving framework

During this project a number of possible improvements have been identified for our carving framework, for which simply not enough time was available.

[1] in my opinion

Handle non-linear fragmentation by using the information acquired during the linear carving run to combine non-linearly fragmented files in a post-processing step. Initial attempts indicate that this is a very hard problem to solve, so it may be wise to first find out how common non-linear fragmentation is in a real world situation, before spending too much time on this.

Extend the file formats supported by Revit07 The quality of carving results in a carver like Revit07 is directly related to the number and quality of its file format definitions. Extending and improving these is therefore always a good idea.

Include results of the DFRWS 2007 challenge A number of results from the DFRWS 2007 file carving challenge might prove useful for inclusion in the carving framework:

- A number of dedicated validators for specific file types were developed by different contestants. See if these can be integrated into the validator framework to increase its reliability.
- Investigate the carving techniques used by other teams and see if they can be used to improve our carver.
- Compare the file format definitions the different teams are using to our own definitions and see if they have found interesting information that we can use.

Use tools to repair broken carving results There are tools that can repair corrupted files of specific types. Investigate whether this can be useful to repair broken carving results.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] Andy Bair, Klayton Monroe, and Jason Smith. Readme.answers. Part of their 2006 DFRWS carving challenge submission, July 2006.

[3] Brian Carrier. *File System Forensic Analysis.* Pearson Education, Inc, 2005.

[4] Simson L. Garfinkel. Carving contiguous and fragmented files with fast object validation. In *Digital Forensics Workshop (DFRWS 2007)*, Pittsburgh, PA, August 2007. `http://www.simson.net/clips/academic/2007.DFRWS.pdf`.

[5] Simson L. Garfinkel. Forensic corpora: A challenge for forensic research. Work in progress, `http://www.simson.net/ref/2007/Forensic_Corpora.pdf`, April 2007.

[6] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing.* MIT Press, Cambridge (Mass.) and London, 1999.

[7] Nicholas A. Mikus. An analysis of disc carving techniques. Master's thesis, Naval Postgraduate School, March 2005. `http://handle.dtic.mil/100.2/ADA432468`.

Appendix A

Tool comparison tables

This appendix contains the full tables for the carving tool quality tests. Since these tables are very large they have been placed sideways on a page and the names for some of the different columns have been abbreviated. All tables use the following legenda:

Tool The tool (configuration) being tested

Present The number of files in a dataset

Positives The number of Positive results

uFNs The amount of unsupported False Negatives

sFNs The amount of supported False Negatives

uFPs The number of unknown False Positives

kFPs The number of known False Positives

Carving Precision The carving_precision score

Carving Recall The carving_recall score

Supp Recall The supported_recall score

cF Measure The cF Measure score

MB per second The average amount of data processed per second

Time The time needed to process the dataset

Size The size of the dataset

Tool	Present	Positives	uFNs	sFNs	uFPs	kFPs	Carving Precision	Carving Recall	Supp Recall	cF Measure	MB per second	Time	Size
Revit-all	15.0	14.0	0	0.97	0.0	1.0	0.966	0.935	0.935	0.95	5.17	12	62
Revit-content	15.0	14.0	0	0.97	0.0	1.0	0.966	0.935	0.935	0.95	5.64	11	62
Revit-val	15.0	13.0	0	0.95	1.0	1.0	0.897	0.937	0.937	0.92	5.17	12	62
Revit-none	15.0	13.0	0	0.95	1.0	1.0	0.897	0.937	0.937	0.92	5.64	11	62
PhotoRec	15.0	12.0	0	1.04	2.0	0.0	0.857	0.931	0.931	0.89	15.5	4	62
Foremost	15.0	11.0	0	1.0	3.0	0.0	0.786	0.933	0.933	0.85	20.67	3	62
Recover My Files	15.0	12.0	0	0.37	1.0	0.0	0.923	0.975	0.975	0.95	2.48	25	62
FTK	15.0	6.0	5	2.97	2.0	0.0	0.75	0.462	0.703	0.57	1.07	58	62
Scalpel	15.0	5.0	0	0.0	1597.0	0.0	0.003	1.0	1.0	0.01	3.1	20	62

Table A.1: Quality score for 11-fat

Tool	Present	Positives	uFNs	sFNs	uFPs	kFPs	Carving Precision	Carving Recall	Supp Recall	cF Measure	MB per second	Time	Size
Revit-all	10.0	7.0	0	0.14	1.0	2.0	0.778	0.986	0.986	0.87	6.89	18.0	124.0
Revit-content	10.0	7.0	0	0.14	2.0	1.0	0.737	0.986	0.986	0.84	9.54	13.0	124.0
Revit-val	10.0	4.0	0	2.47	0.0	7.0	0.533	0.753	0.753	0.62	7.75	16.0	124.0
Revit-none	10.0	4.0	0	2.47	5.0	2.0	0.4	0.753	0.753	0.52	11.27	11.0	124.0
PhotoRec-ext2	10.0	10.0	0	0.0	0.0	0.0	1.0	1.0	1.0	1.0	12.4	10.0	124.0
PhotoRec	10.0	3.0	0	3.08	6.0	0.0	0.333	0.692	0.692	0.45	12.4	10.0	124.0
Foremost-ext2	10.0	6.0	0	0.12	4.0	0.0	0.6	0.988	0.988	0.75	11.27	11.0	124.0
Foremost	10.0	5.0	0	3.0	6.0	0.0	0.455	0.7	0.7	0.55	11.27	11.0	124.0
Recover My Files	10.0	3.0	0	1.02	6.0	0.0	0.333	0.898	0.898	0.49	3.1	40.0	124.0
FTK	10.0	3.0	0	2.59	8.0	0.0	0.273	0.741	0.741	0.4	1.02	122.0	124.0

Table A.2: Quality score for 12-ext2

Tool	Present	Positives	uFNs	sFNs	uFPs	kFPs	Carving Precision	Carving Recall	Supp Recall	cF Measure	MB per second	Time	Size
Revit-all	32.0	19.0	0	0.65	0.0	14.0	0.731	0.98	0.98	0.84	0.83	58.0	48.0
Revit-content	32.0	19.0	0	0.65	10.0	4.0	0.613	0.98	0.98	0.75	1.07	45.0	48.0
Revit-val	32.0	12.0	0	7.87	0.0	17.0	0.585	0.754	0.754	0.66	0.91	53.0	48.0
Revit-none	32.0	12.0	0	7.87	11.0	6.0	0.462	0.754	0.754	0.57	1.2	40.0	48.0
PhotoRec	32.0	19.0	0	4.01	11.0	0.0	0.633	0.875	0.875	0.73	1.78	27.0	48.0
Foremost	32.0	7.0	0	8.94	19.0	0.0	0.269	0.721	0.721	0.39	24.0	2.0	48.0
Recover My Files	32.0	10.0	0	5.44	17.0	0.0	0.37	0.83	0.83	0.51	1.78	27.0	48.0
FTK	32.0	8.0	4	9.25	24.0	0.0	0.25	0.586	0.644	0.35	0.15	312.0	48.0

Table A.3: Quality score for dfrws2006

Appendix B

Tested carving tools

This appendix provides background information on the carving tools tested in this project.

For each tool the following information is provided:

- The version of the tool that was tested.
- The creator of the tool, or at least the company that created it.
- A description of the tool, literally copied from the tools own website.
- The different file types that the tool (claims to) support. This is either a link to an online list of supported types or a list of file type extensions.
- The carving techniques used by a tool, as far as they are known.

B.1 Foremost

Version: 1.5, 12-04-2007
Created by Nick Mikus

Description from the website:
`http://foremost.sourceforge.net`
Foremost is a console program to recover files based on their headers, footers, and internal data structures. This process is commonly referred to as data carving. Foremost can work on image files, such as those generated by dd, Safeback, Encase, etc, or directly on a drive. The headers and footers can be specified by a configuration file or you can use command line switches to specify built-in file types. These built-in types look at the data structures of a given file format allowing for a more reliable and faster recovery.

Supports file structure based carving for avi, bmp, doc, gif, hmlt, jpg, mov, pdf, png, rar, wav and zip files.

Supports header-footer carving for art, asf, chm, cookie, cpp, dat, dbx, fws, idx, java, lnk, mail, mbx, mp3, mpg, ost, pgd, pgp, ppt, pst, ra, rdp, rpm, tif, txt, wma, wmv, wpc and xls files.

B.2 Forensic Toolkit (FTK)

Version: 1.71
Created by AccessData

Description from the website:
`http://www.accessdata.com/catalog/partdetail.aspx?partno=11000`
The perfect tool for thorough forensic examinations. FTK has full text indexing, advanced search, deleted file recovery, data-carving & email graphics analysis.

Supports abl, aol, asd, bmp, doc, dot, emf, gif, html, jpg, mpp, one, pdf, png, ppt, pub, puz, vsd, vss, vst, xla, xls and xlt files.

Closed source, so internal techniques unknown

B.3 PhotoRec

Version: 6.9-WIP
Created by Christophe Grenier

Description from the website:
`http://www.cgsecurity.org/wiki/PhotoRec`
PhotoRec is file data recovery software designed to recover lost files including video, documents and archives from Hard Disks and CDRom and lost pictures (thus, its 'Photo Recovery' name) from digital camera memory. PhotoRec ignores the filesystem and goes after the underlying data, so it will still work even if your media's filesystem has been severely damaged or re-formatted.

Supports of 80 file formats, described in `http://www.cgsecurity.org/wiki/File_Formats_Recovered_By_PhotoRec`

PhotoRec uses a combination of file structure based carving and header-footer carving.

B.4 Recover My Files

Version: 3.9.8.5750
Created by GetData

Description from the website:
`http://www.recovermyfiles.com`
Recover My Files data recovery software will easily recover deleted files emptied from the Windows Recycle Bin, or lost due to the format or corruption of a hard drive, virus or Trojan infection, unexpected system shutdown or software failure.

According to the text on the website: "Recover My Files will find any type of deleted file" 200 file types have been specifically named, however, described in
`http://www.recovermyfiles.com/recover-deleted-file-types.php`

Closed source, so internal techniques unknown.

Note that Recover My Files recommends the user to select at most five file types to search for, to limit the execution time, but claims that all other file types will be recovered anyway. This is not the case, since on multiple occasions adjusting the list of file types to search for had a direct impact on the type and number of files that were recovered. This is in my personal opinion a grave reliability problem.

B.5 Scalpel

Version: 1.60, 12-08-2006
Created by Golden G. Richard III

Description from the website:
`http://www.digitalforensicssolutions.com/Scalpel`
Scalpel is a fast file carver that reads a database of header and footer definitions and extracts matching files from a set of image files or raw device files. Scalpel is filesystem-independent and will carve files from FATx, NTFS, ext2/3, or raw partitions. It is useful for both digital forensics investigation and file recovery. Scalpel resulted from a complete rewrite of foremost 0.69, a popular open source file carver, to enhance performance and decrease memory usage.

Supports art, avi, dat, dbx, doc, fws, gif, htm, idx, java, jpg, mail, max, mbx, mov, mpg, ost, pdf, pgd, pgp, pins, png, pst, ra, rpm, tif, txt, wav, wpc and zip files.

Scalpel uses a combination of header-footer and header-“maximum file size” carving.

Appendix C

Validator tools

This appendix provides a list of the programs used by the validator framework. For each file the following information is provided:

- The full program(suite) name and version
- A short description
- The file types that are validated using this tool, by extension
- A short description of how this program is used for validation

C.1 Tool descriptions

catppt

Name and version	Catdoc Version 0.94.2
Description	catppt reads MS-PowerPoint file and puts its content on standard output
File types	ppt
Validation method(s)	Run the program, scan the log for error messages. No error messages implies a valid file, otherwise it is unviewable.

convert

Name and version	ImageMagick 6.2.4
Description	convert between image formats as well as resize an image, blur, crop, despeckle, dither, draw on, flip, join, re-sample, and much more.
File types	gif, jpeg, png, bmp, tiff
Validation method(s)	Convert the target file to PNG and check the error messages. No error messages means valid, conversion succeeded but with error messages means viewable and a failed conversion means unviewable

eu-elflint

Name and version	elflint (Red Hat elfutils) 0.128
Description	Pedantic checking of ELF files compliance with gABI/psABI spec.
File types	elf[1]
Validation method(s)	Run the program, scan the log for error messages. If the log contains the message: 'No errors', then this implies a valid file, otherwise it is unviewable.

mp3check

Name and version	mp3check 0.8.0
Description	check audio mpeg layer 1,2 and 3 (*.mp3) files for consistency (headers and crc) and anomalies
File types	mp3
Validation method(s)	Run the program and check the programs exit status. 0 implies a valid file, otherwise it is unviewable.

mplayer

Name and version	MPlayer 1.0rc1-4.1.3-DFSG-free
Description	mplayer is a movie player for Linux
File types	asf, avi, flv, mpeg, qt, wav, wmv
Validation method(s)	Play the video with no audio or video output and at increased speed to limit the time required, scan the log for error messages. No error messages mean a valid file, files smaller than 512 bytes or for which mplayer exits instantly or gets a segmentation fault are deemed unviewable and all other files are deemed viewable.

pdfinfo combined with pdftops

Name and version	pdfinfo 3.02
Description	Portable Document Format (PDF) document information extractor
File types	pdf
Validation method(s)	Run the program and scan the log for error messages. No error messages mean a valid file, otherwise the file needs to be checked further using pdftops

Name and version	pdftops 3.02
Description	Portable Document Format (PDF) to PostScript converter
File types	pdf
Validation method(s)	Convert the PDF file to PostScript. If the conversion succeeded then the file is deemed viewable, otherwise it is unviewable.

tidy

Name and version	HTML Tidy for Linux/x86 released on 1 September 2005
Description	Utility to clean up and pretty print HTML/XHTML/XML
File types	html
Validation method(s)	Run the program and scan the log for error messages. No error messages mean a valid file, otherwise it is unviewable.

file

Name and version	file 4.18
Description	Determine file type of FILEs.
File types	txt
Validation method(s)	Run the program and scan the output for the word 'text'. This is too weak a validation to say that the file is valid, so if the word 'text' is found then the file is deemed viewable, otherwise it is unviewable.

unzip

Name and version	UnZip 5.52
Description	List, test and extract compressed files in a ZIP archive
File types	zip
Validation method(s)	Run the program and check the programs exit status. 0 implies a valid file, 1 or 2 imply viewable, otherwise it is unviewable.

link.exe, using wine

Name and version	Microsoft (R) Incremental Linker Version 8.00.40310.39
Description	link.exe or link process information for Microsoft Incremental Linker
File types	exe
Validation method(s) 0 implies a valid file, otherwise it is unviewable.	Run link.exe using wine and check the exit status.

Name and version	Wine 0.9.41
Description	Run Windows programs on Unix

wvText

Name and version	wvWare 1.2.4
Description	Convert msword documents
File types	doc
Validation method(s)	Run the program, scan the log for error messages. If the error messages include the line: 'Could not convert into HTML' then the file is unviewable, else it is deemed valid.

xlhtml

Name and version	xlhtml 0.5.1
Description	A program for converting Microsoft Excel Files .xls
File types	xls
Validation method(s)	Run the program and check the programs exit status. 0 implies a valid file, otherwise it is unviewable.