Advanced Carving techniques

Author: Michael Cohen (Scudette@users.sourceforge.net)

Abstract

"Carving" is the term most often used to indicate the act of recovering a file from unstructured digital forensic images. The term unstructured indicates that the original digital image does not contain useful filesystem information which may be used to assist in this recovery.

Typically, forensic analysts resort to carving techniques as an avenue of last resort due to the difficulty of current techniques. Most current techniques rely on manual inspection of the file to be recovered, and manually reconstructing this file using trial and error. Manual processing is typically impractical for modern disk images which might contain hundreds of thousands of files.

At the same time the traditional process of recovering deleted files using filesystem information is becoming less practical because most modern file systems purge critical information for deleted files. As such the need for automated carving techniques is quickly arising even when a filesystem does exist on the forensic image.

This paper explores the theory of carving in a formal way. We then proceed to apply this formal analysis to the carving of several different file types based on the internal structure inherent within the file format itself. Tools are introduced to automate carving for a number of important file types. Specifically this paper deals with carving from the Digital Forensic Research Work-Shop's 2007 carving challenge (DFRWS Challenge).

Part 1: Formal Analysis

This section explores the carving process formally and introduces a number of definitions which will be used throughout the paper. In the following discussion the term "file" refers to the file to be recovered (pdf, mp3 etc) while the term "image" refers to the image the file is to be recovered from.

In essence, the process of carving can be seen as extracting all the bytes belonging to a file from an image. This is achieved by means of a "mapping function".

Definition: *Mapping Function*
A mapping function is a function which maps bytes in the image to bytes in the file.

An example of a typical mapping function is shown below:

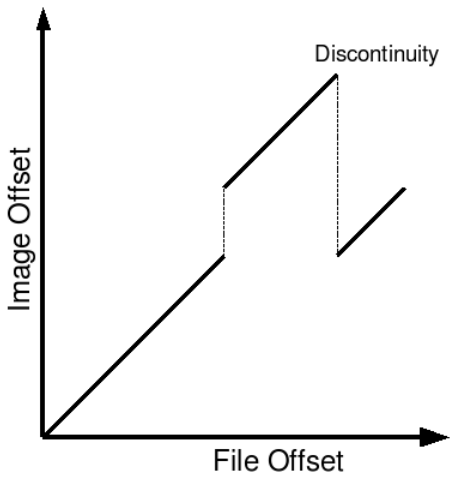

Illustration 1: Example of a Mapping Function

The mapping function has a number of interesting properties:

1. The slope of the function is always 1 (because there is a 1-1 mapping for the image with the file).
2. There are a number of discontinuities in the function at various places.
3. The function is invertible - i.e. there is at most one file offset for each value of the image offset (if such a value exists). In practice this means that once a sector from the image has been used it can not be used again.
4. The DFRWS challenge imposes a further restriction on the mapping function in that discontinuities can only occur on sector boundaries (512 bytes). (This is not always the case since some filesystems may allocate files in arbitrary sized chunks. This assumption is explored further below in the discussion of fragmentation models)

The process of carving essentially boils down to estimating the mapping function based on deductions which can be made from knowledge of the internal file structure.

For a finite sized image, there are actually only a finite number of mapping functions, resulting from the permutations of all sectors with each other (this number may be absolutely huge but it is actually finite). A naive approach to carving might be to create a generator of all possible mapping functions and examine the resulting file for each generated function. Aside from the fact that the number of combinations is impractical, this process highlights another critical component of a carving system:

Definition: *Discriminators*
A discriminator is a process which estimates the validity of a mapping function.

In effect the discriminator can tell the difference between a recovered file which is corrupt and one which is likely to be correct. A good discriminator is able to detect corruption in the recovered file

with a great level of certainty, and in particular localise the corruption within the file. Depending on the file type, it may or may not be possible to build such an effective discriminator.
There can often be several different discriminators for the same file type, each with different efficiency/cost trade-offs. For example a fast discriminator may be less accurate, while a very good discriminator may be too slow to run (e.g. a very good discriminator for English text is a human which can read and make sense of the text – but using this discriminator is slow and must be done sparingly).

Definition: *Mapping Function Generator*
A mapping function generator is a process which generates new mapping functions. These functions are tested by the discriminator and the results are fed back into the generator.

An efficient discriminator can be used to drive the choice of new mapping functions by isolating the parts of the file which are deemed correct, and trying different sectors for those parts of the file which are corrupted. The function generator can then use this information to select better mapping functions. Note that the function generator may have a number of different discriminators available to it and may use different discriminators under different situations.

The process of carving can be viewed as a mathematical discrete optimisation problem – a problem which is usually NP complete (i.e. can only be solved by trying every single possibility). Clearly a purely combinatorial mapping function generator is infeasible due to the large number of possibilities. In order to find the correct mapping function, a series of deductions and algorithms may be used to impose constraints on the mapping function based on knowledge of internal file structure. There are a number of assumptions that can be used to reduce the total number of possible mapping functions, and assist the generator in optimising the mapping function. Some file types may lend themselves more easily to some techniques while others may not.

Simplifying assumptions

The first kind of assumption may arise from analysing the internal file structure and thereby identifying constraints upon the mapping function itself. The constraints may be positive or negative constraints:

Definition: *Positive Constraints*
Positive constraints are deductions which identify a likely point on the mapping function. This likely point is referred to as an *identified point*.

A positive constraint might actually deduce a number of possible points on the function - only one of which belongs to the true function. A good positive constraint reduces the number of possibilities to just one or very few such points.

For example, a positively identified file header places a constraint on the first sector of the file. We see more examples of positively identified points in the specific file analysis below. Sometimes, depending on the file format, positive constraints may be identified throughout the file.

Definition: *Negative Constraints*
Negative constraints are deductions which exclude certain points from the mapping function.

A negative constraint can be defined as an observation that a certain sector does not belong within the sequence observed. This might occur, for example, if the sector clearly does not exhibit the characteristics required from this file type (e.g. sector with binary bytes following a HTML sector etc). Typically the discriminator will exclude a sector anyway, but the use of negative constraints can be invaluable for efficiency, since obviously incorrect sectors may not be chosen as potential candidates.

Fragmentation Models

Fragmentation occurs in forensic media as a result of filesystem allocation strategy. These strategies are typically designed to optimise some characteristic of the filesystem (e.g. faster access speed, better storage efficiency etc) and are not deliberately designed to make recovery difficult. It is often possible to make assumptions about the type of fragmentation present based on information about the file system which was previously on the disk and its allocation strategies.

Definition: *Fragmentation Model*
A Fragmentation model is a set of assumptions derived from an observation of how fragmentation occurs in practice. This may depend on filesystem characteristics or other simplifying assumptions.

The model can be applied to the generator by reducing the number of mapping functions which need to be examined, or having the mapping function generator try out those mapping functions which are more likely to be correct based on the fragmentation model before those less likely to occur.
Having good knowledge of filesystem structures may be used in guiding the function generator by guessing more likely mapping functions.
The DFRWS challenge presents a fragmentation model which states that discontinuities may only occur on sector boundaries and sectors are 512 bytes long. The image does not have a filesystem at all (so we do not attempt to find any filesystem structures).
We adopt this model for the remainder of the paper.

Traditional Carving Techniques

Traditionally, carving is done by a start of file/end of file method. It is instructive to describe the traditional method in terms of the present theory:

1. The image is scanned for start of file signatures (File headers). This in effect identifies a point on the mapping function by fixing the file start (file offset of zero) with the image offset of the signature.
2. We adopt a simple fragmentation model which assumes no fragmentation at all. This implies that the mapping function can have no discontinuities and be of unit slope.

3. Often using internal file information, the file size may be deduced. This places another identified point for the tail of the file. Alternatively, we can search the mapping function for a tail signature, and use it to identify the tail point. Failing these, an arbitrary tail point is chosen resulting in a sufficient length of file.
4. Due to the simplicity of the model, the total number of possible functions is small, and an automatic discriminator is not often used – leaving the task of discriminating to the human investigator previewing the results manually.

Fragmentation Order

Consider the following possible mapping function:

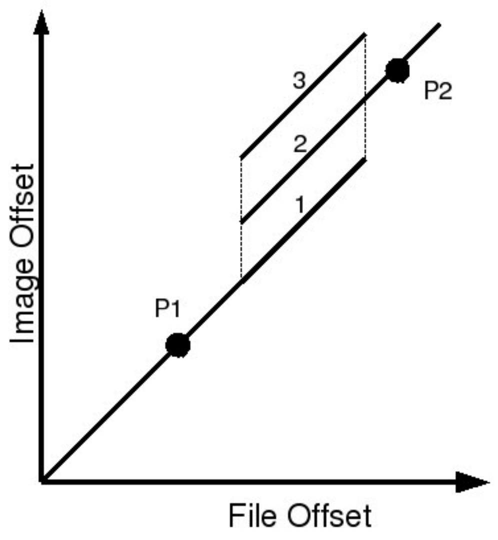

Illustration 2: Fragmentation modes

This function has 2 positively identified points (P1 and P2). Since the points do not lie on the same line of slope one, there must be at least one discontinuity between them. The figure shows 3 possible mapping functions numbered 1,2,3.

Possibilities 1 and 2 have a single discontinuity on a sector between P1 and P2. However, possibility 3 has 2 discontinuities. We say that possibilities 1 and 2 exhibit first order fragmentation, while possibility 3 exhibits second order (or higher order) fragmentation.

Definition: *Fragmentation order*
Fragmentation order is the number of discontinuities which occur between any two identified points.

Although in theory there can be as many discontinuities between any identified points as there are sector boundaries, in reality fragmentation is typically kept to a low level with modern filesystems. This implies that the smaller the file offset difference between identified points the more likely the fragmentation will be of first order. A good simplifying assumption is to assume that all fragmentation is of first order.

Interpolation

In order to estimate the mapping function we must predict values occurring on the mapping function which do not correspond to identified points. This process is called interpolation. There are 2 possible ways to interpolate between two identified points:

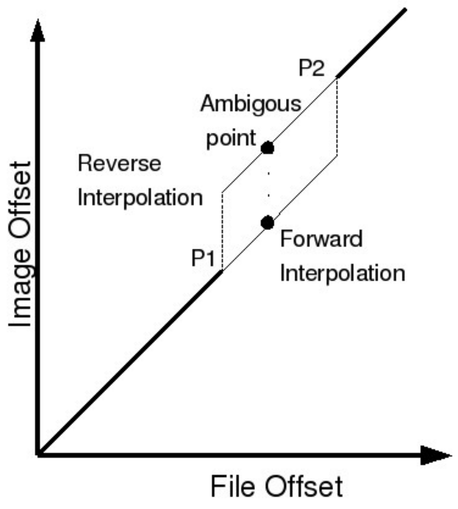

Illustration 3: Possible interpolation strategies

Definition: *Forward Interpolation*
The value is interpolated forward from P1 through a straight line of unit slope which goes through P1. The line terminates where the file offset is the same as that of P2.

Definition: *Reverse Interpolation*
The value is interpolated in reverse from P2 through a straight line of unit slope which goes through P2. The line terminates where the file offset is the same as that of P1.

Note that if P1 and P2 lie on the same line, forward interpolation and reverse interpolation yield the same value. Note also that the true mapping function will only coincide with the forward or reverse interpolation if there is first order fragmentation.

Definition: *Ambiguous Points*
Ambiguous points are those points on the mapping function which have different image offsets for a given file offset depending on the manner of interpolation.

Given first order fragmentation, if P1 and P2 are both on the same line of unit slope, there are no ambiguous points between them because interpolating forward and backwards yield the same points.

Projection Line

Consider the following identified points P1 and P2:

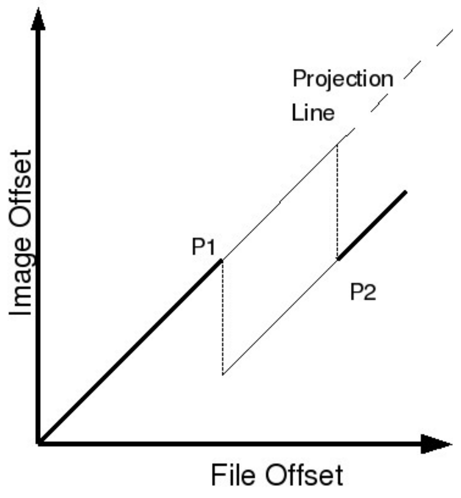

Illustration 4: An impossible mapping function

In the above figure, the projection line is a line of unit slope which passes through P1. The second identified point P2 lies below the projection line. As can be seen it is impossible to interpolate a valid mapping function (assuming only first order fragmentation) between P1 and P2 in this case because the function is not invertible[1]. This property of mapping functions can be used in many cases to immediately discount certain mapping functions without needing to test them.
This condition (P2 is below the projection line) implies that fragmentation order is higher than first order and that more identified points are required (or an exhaustive search must be conducted).

The above analysis applies equally well to fragmentation which is not on sector boundaries but the algorithms can be simplified and optimized by taking into account the specific fragmentation model.

Overview of a carver

The above analysis describes the general theory of carving. Presently we discuss a general construction for carving algorithms. Depending on the specifics of the file format examined, this design might vary, but generally the carvers presented in this paper follow this overarching design.

The diagram below illustrates some of the major components of a semantic carver:

1 Mapping functions must be invertible because there is a 1-1 relation between file offsets and image offsets. The above example has regions where image data is used twice in the file – which is impossible to occur in reality.

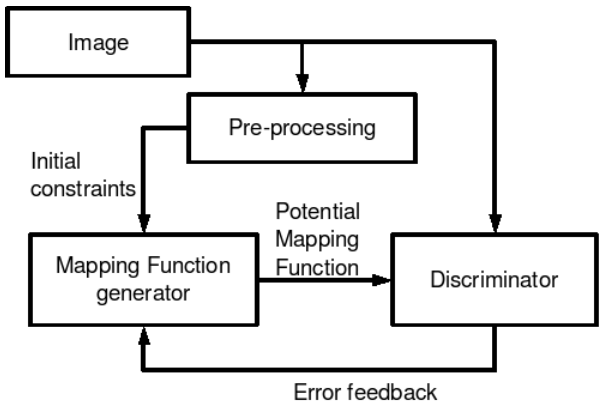

Illustration 5: Overview of Carving algorithms

1. The image is first fed into a preprocessing system which extracts information about the file being carved. This might include the identification of points on the mapping function.
2. This information is then fed to a mapping function generator. The generator can use this information to constrain the mapping functions produced.
3. For each prospective mapping function, the generator feeds the resultant file to the discriminator. The discriminator attempts to detect errors in the file and most importantly, the discriminator attempts to estimate the point where the file had been corrupted. This information is fed back to the generator which uses it to further reduce the number of functions tested.
4. The generator returns a number of possible mapping functions which are considered good by the discriminator. The user may use some other discriminator (e.g. human judgement) to further reduce this set.

The discriminator

In the above algorithm, the discriminator is used to guide the generation process as well as validate the file. Although general purpose programs may be used as discriminators (e.g. use xpdf or Adobe Acrobat as a discriminator for PDF files, WinZip as a discriminator for ZIP files etc), there are a number of problems with this approach:

- The general purpose programs expect the file not be corrupted, this means that often corruptions are not even detected because a full integrity check of the file is not performed. For example, the PDF file format allows for random seeking throughout the file for arbitrary pages. The xpdf program does not verify the integrity of pages which are not rendered, which means that it is time consuming to verify the entire PDF file because it all needs to be rendered.
- A general purpose program might attempt to recover from errors rather than report them.
- Most general purpose programs fail to indicate the precise location and cause of the errors - this information is crucial for the generator function to identify which sector is likely to be

out of place. Without this information all permutations need to be generated which could become intractable.

Clearly specialised parsers for the file types of interest must be employed to build a good discriminator. Of course, if the specifics of the file format are not known we are left with using general purpose programs as a final resort.

Discriminators do not generally need to understand fully the file format in question, they just need to understand enough of the overall structure to be able to discriminate between corrupted files and good files. The mapping function generator may use more thorough discriminators (e.g. human examiners) to further narrow down a small list of possibilities.

Part 2: PDF File Carving

The PDF format is describe in *The PDF Reference* 6th edition available from the Adobe web site.

The format is essentially a line oriented text based format with embedded binary data. An implementation of a tokeniser and parser for PDF can be found in the file PDF.py. Here we just describe some high level properties of the PDF format.

The Objects

PDFs contain a sequence of objects within the file. Objects are high level containers for other data and in particular stream data (encoded binary data) may be contained in the object. Objects are numbered by an object number and a generation number and followed by the object tag which looks like:

`x y obj`

where *x* is the object number and *y* is the generation number. A PDF document contains an incrementing sequence of objects which generally appear immediately after one another. Objects can be large if they contain streams or only a few bytes if they contain more simpler data types.

XREF Tables and Trailers

For fast access into the file, the standard defines XREF tables. These tables contain runs of offsets into the file for each object ID specified. An XREF table is followed by a trailer section which describes a number of different properties of the table. Some of the more useful properties include the document ID (which can be used to collate XREF tables belonging to the same document together) and the /Prev tag which specifies the offset to the previous XREF table.

Example

For example the following is an example of an xref table extracted from a typical PDF file:

```
xref
0 28
0000000000 65535 f
0000369883 00000 n
0000000019 00000 n
...
trailer
<<
/Size 280
/Info 219 0 R
/Root 235 0 R
/Prev 309490
/ID[<efff3bba84c3e54573acfe1fae914a45>
<efff3bba84c3e54573acfe1fae914a45>]
>>
```

```
startxref
0
%%EOF
```

After the *xref* keyword, the range of object IDs is given (in this case 0 to 28). Then for each object ID in this range, the file offset is given (e.g. in this case object 1 is at offset 369883, object 2 is at offset 19 etc). This means that in this PDF file at offset 19 for example we expect to find the string, **2 0 obj**

The trailer part in this example indicates that there is another xref table at offset 309490 which is indicated using the /PREV keyword. This information may be used in coalescing this *xref* table with the one found at that offset.

The *xref* tables provides us with a large number of positively identified points, because they satisfy the basic properties for identified point sources:

1. They reference an exact offset within the reassembled file. (In terms of the previous discussion, the *xref* tables provide a fix on the *x* coordinate of the identified point on the mapping function – where the *x* axis is the file offset).
2. It is possible to determine with a high level of confidence if the point referred to is in fact the point expected.
3. If the point referred to is not the point expected, it is possible to locate the expected point within the image with a high level of confidence. This essentially provides us with a fix on the *y* coordinate of the identified point – where y is the image offset axis.

For example, assume the *xref table* claims that object 3 is at offset 100 into the file. This provides a positively identified *x* coordinate on the mapping function. We then search the image for all objects with an id of 3, and choose the one which is most likely to be correct (This decision may be based on the fragmentation model). This provides us with a fix on the *y* coordinate of the identified point on the mapping function (i.e. image offset). Using both coordinates we can positively identify the point on the mapping function.

Using the fragmentation model given by the DFRWS challenge, we can see that the following must hold:

file offset **mod** 512 = image offset **mod** 512

This is true since fragmentation can only occur on sector boundaries and sectors are 512 bytes. In addition, an implied assumption is that files must also be aligned to sector boundary – a constraint which is not actually stated by the DFRWS challenge, but is common in many filesystems. This relation is termed the **modulus rule**, and is used throughout this paper as a property of the fragmentation model.

An example of this process is shown in the figure below:

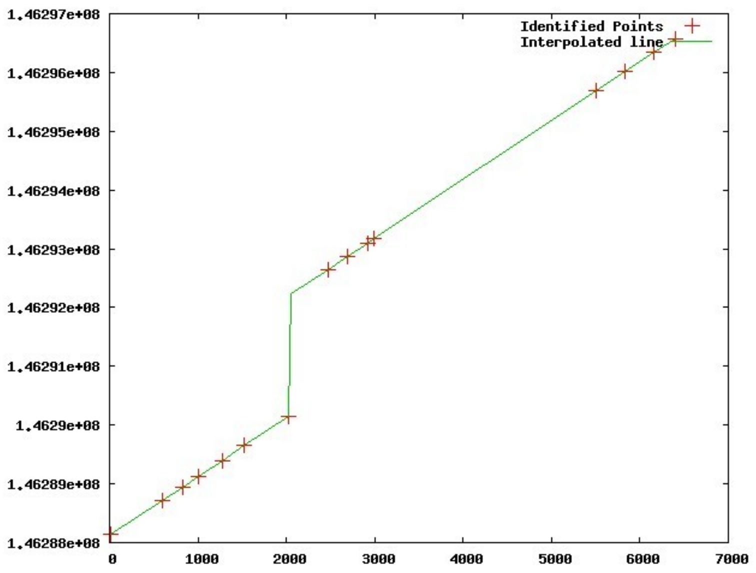

Illustration 6: Mapping function for a PDF file. x axis is file offset, y axis is image offset.

In this figure, the red crosses indicate PDF objects as obtained from the *xref tables*. Their image offsets are obtained using the modulus rule above. The green line is the correct mapping function which is obtained by interpolating between the identified points using the discriminator described below.

Preprocessing

Carving for PDFs is performed in 2 passes. The first pass is the preprocessing pass, where we attempt to locate all PDF components present within the image. In particular we use regular expressions to index the image offsets of all *xref tables*, and PDF objects. This is saved to an index file for rapid subsequent loading.
Once the index is build we coalesce the *xref tables* which appear to belong together.
We build a series of map files corresponding to the different *xref tables* found. The user can the apply the carver for each of these map files to attempt to recover each PDF file in turn.

The discriminator

It is important for a good discriminator to localise errors within the file, so that discontinuities can be determined more accurately. Using an external program like xpdf for example is inadequate due to the following reasons:

- Xpdf reads the document in accordance with the standard. The standard requires the reader to follow the following steps:
  - Check the header for a PDF comment.
  - Seek to the end of the document and locate the trailer and *xref tables*.
  - Read the page structure.
  - Open the objects that describe the page the reader wants to render by seeking directly to

their offsets (as obtained from the *xref tables*).

- This is a problem because often a mapping function will start off without XREF tables, PDF comments or even basic paging objects. This will cause Xpdf to error out immediately without attempting to open the document at all – giving no feedback of where the file is corrupted.
- Xpdf is not consistent in its error reporting – often an offset of where the error occurs will be reported, other times errors will not be reported (for example if Xpdf gives up).

There are a number of other PDF implementations available – for example panda is a lex/yacc based implementation. In the present case, a specially written Lexer based parser was written.

PDF Lexer Implementation

The parser presented in this paper is a stateful feed tokeniser. The parser is a state machine which transitions from state to state based on matching tokens in the input stream. More details of the implementation can be seen in the source file PDF.py. The parser presents the following properties:

1. The PDF file is read sequentially from the start.
2. Depending on the current state a set of regular expressions is tried to find tokens which are valid in the current state.
3. If valid tokens are found, an appropriate callback is called and the parser state changes to the next state. These callbacks maintain high level semantic meaning of each token (i.e. we do not apply formal grammars, preferring instead to manage syntax in a more flexible way through the tokeniser state machine).
4. Sometimes a valid token is not found for the current state, in which case the lexer is said to be stuck. An error callback is used to indicate this condition and the input is flushed by a single byte to try to re-sync the parser.
5. The parser has a running tally of errors – and different errors can be weighed differently. Since the parser reads the file sequentially from the start watching this error tally can indicate when a discontinuity occurs as the number of errors encountered at that point increases very quickly.
6. PDF streams may be compressed or encoded in many ways. The parser is able to detect errors in stream encodings (and in particular compressed data). Because stream objects are typically much larger than other types of objects, it is more likely that errors occur in the compressed data.

PDF function generator

The *Reassembler* class defined in *Carver.py* implements an abstracted file like object based on the mapping function. The object can accept positively identified points using the add_point() method and automatically interpolates between them. It presents a standard file like interface (i.e. seek, read methods) which hides the interpolation of the mapping function itself. This abstraction makes it easy to pass a mapping function directly to the discriminator parser and presents a simple interface for direct manipulation of the mapping function by the generator – by allowing the adding and removing of points.

Whenever there are ambiguous points in the mapping function, the generator resolves the ambiguity by inserting test points within the range of identified points. The discriminator is then called on the disambiguated mapping function and if errors are present, the point is removed, and re-inserted one sector further to the right. Assuming first order fragmentation only, one of the test points should result in no errors occurring within the P1-P2 range.

This approach is illustrated in the figure below:

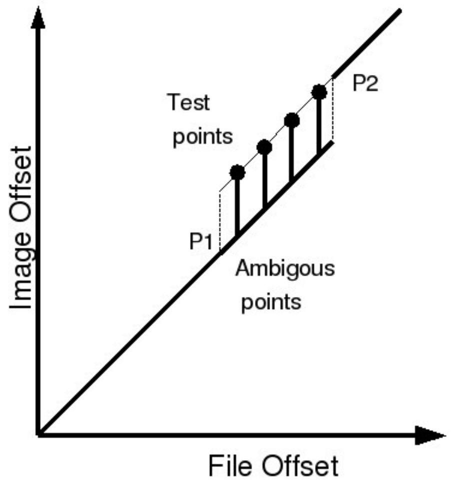

Illustration 7: Test points added to the Reassembler to disambiguate the mapping function

Optimisations

The generator works by inserting test points in sequence, and then reparsing the resulting mapping function using the discriminator. This can be slow, especially if the PDF file recovered is large, and the ambiguous points are close to the end – since the parser starts from the beginning of the file each time, and parses the file (which is mostly the same each time) until reaching the permuted region.

Since the Lexer is a state machine, it is possible to fully define its future evolution in terms of its current state. Furthermore, its current state can be fully saved and restored by saving a number of critical state variables. The PDF object automatically saves its state at convenient points in the PDF file. This essentially allows the discriminator to resume from a point just before the region being tested instead of re-parsing the file from the start each time. This optimisation can lead to very fast performance.

All together now

The algorithm then proceeds in the following steps:

1. All objects and xref tables are found and indexed on the disk.

2. Possible objects are located from the index for each xref table by assuming fragmentation on sector boundaries (i.e. offset required % 512 = object offset % 512).
3. The *xref* tables belonging to the same file are merged together (using the /PREV document property). This process is called *coalescing the xref tables*. This step is required because many PDF files have several xref tables within them.
4. For each file, a carver object is instantiated and identified points are added to the object.
5. For each ambiguous region test points are added and the carver attempts to re-parse the result using the PDF parser.
6. Once all discontinuities are resolved, the parser is used to locate the end of the document.
7. The document is saved to disk. The resulting map is saved to disk as well.

PDF Carving Tool

The following illustrates how to use the PDF caving tool published with this paper. There are 3 main steps to carving PDFs. The index creation step, the map creation step and finally the recovery of the PDF files themselves.

The first step is to create an index of all possible PDF components within the image:

```
pdf_carver.py -i pdf.idx -c dfrws-2007-challenge.img
Found OBJECTS in 936
Found OBJECTS in 1222
Found OBJECTS in 2333
Found OBJECTS in 2382
```

The program creates the index file pdf.idx by recording the locations of objects, xref tables and startxref tags.

Next the initial maps are created by identifying points for the objects:

```
pdf_carver.py -i pdf.idx -m dfrws-2007-challenge.img
Reading XREFs table from 38267254
[]
Output of PDF file
Xref Table:
[0, 0, 65535, 'f']
[1, 388895, 0, 'n']
[2, 389047, 0, 'n']
[3, 389297, 0, 'n']
[4, 398708, 0, 'n']
[5, 416173, 0, 'n']
[6, 469803, 0, 'n']
...
251 - 0 f
252 - 0 f
253 - 0 f
254 - 0 f
255 - 0 f
256 - 0 f
[38315692, 305551242, 300917133, 146306832, 304689779, 304512789, 236510367,
103537905, 147103916, 146288243,
 61335724, 304516723, 40306803, 148733892, 103228588, 300187251, 304655475,
300966897, 300968051, 38267254, 3
00214444, 40320638]
Node 38315692 has the following xrefs:
```

```
Checking 482167.0 against xref at 38315692 (375.0,172)
Checking 482167.0 against xref at 38267254 (375.0,374)
Xref at offset 38267254 is possibly related to xref at offset 38315692
My range is [0, 10], their range is [10, 160]
Ranges for xref tables match - coalescing...
10 - 16 n
11 - 3348 n
12 - 3812 n
13 - 4020 n
```

As can be seen, the xref tables are coalesced if their ranges match. The result is a set of mapping files:

```
103228588.map
103537905-103228588.map
103537905.map
146288243.map
146306832-146288243.map
146306832.map
147103916.map
148733892-147103916.map
148733892.map
236510367.map
300187251.map
300214444.map
300917133-300187251.map
300917133.map
300966897-300214444.map
300966897.map
300968051.map
304512789.map
304516723-304512789.map
304516723.map
304655475.map
304689779.map
305551242-304689779.map
305551242.map
38267254.map
38315692-38267254.map
38315692.map
40306803.map
40320638-40306803.map
40320638.map
61335724.map
```

Both the coalesced maps and the individual maps are created in case the coalescing process made a mistake. Note that the user can manually coalesce any map functions by just catting them together. The map files simply consist of 3 columns: the first column is the file offset, the second is the image offset and the third column is a comment to indicate what this identified point is.

The final step is to actually carve each of these maps out. This may or may not be successful so users need to do it separately for each file. For example:

```
PDFForcer.py -m 148733892-147103916.map -r 148733892-147103916.map.final -o
148733892-147103916.pdf ../dfrws-2007-challenge.img
Sector 2481 is ambiguous - testing
Checking until 1274843
Error: Lexer Stuck, discarding 1 byte ('\xa5\x93F\x15\x019enH,') - state ARRAY
...
Error: Lexer Stuck, discarding 1 byte ('\xa4p\x1e\xa5\x00v\r\r%\x00') – state
```

```
ARRAY
Error count is 10
Sector 2482 is ambiguous - testing
Checking until 1274843
Found a hit at 2482
Error count is 0
Writing reconstructed file: 148733892-147103916.map.final
Writing file: 148733892-147103916.pdf
Total reconstructed file error count: 0
```

As can be seen in this example we test the file up to the first ambiguous point, and then continue testing one sector at the time until we have no errors. The use of the -v switch increases debugging output and helps determine any problems which might occur closer to the discontinuities.

Part 3: Zip Format

ZIP files are described in the application note http://www.pkware.com/documents/casestudies/APPNOTE.TXT. Although the application note discusses a Zip64 standard with different format, it seems to suggest that much of that standard is covered by patent claims. This means that in practice it is uncommon to see and most zip files use the old structures for compatibility. The following is a brief overview of the file format.

A ZIP file is an archive which contains a number of different files embedded within it. The general structure is illustrated below:

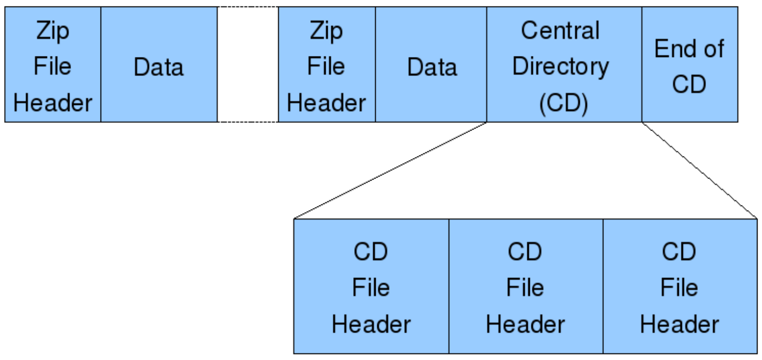

Illustration 8: Zip File Format Overview

Data Structures

The following data structures are used in the ZIP file format. Note that not all fields are listed here, for a full listing of all fields see the code in Zip.py. Here we only list fields relevant to the discussion.

When a ZIP file is opened, the reader searches backwards from the end of the file for the End of Central Directory magic – which yields the EndCentralDirectory structure:

```
EndCentralDirectory
        magic , ULONG_CONSTANT, expected = "PK\x05\x06"
        number_of_this_disk, USHORT
        disk_with_cd, USHORT
        total_entries_in_cd_on_disk, USHORT
        total_entries_in_cd,USHORT
        size_of_cd, ULONG
        offset_of_cd,ULONG
```

This structure reveals important information about the central directory (CD) itself, namely the size of the CD, the number of entries in the CD and the file offset of the CD.

Readers then read the CD itself which consists of an array of CDFileHeader structures:

```
CDFileHeader
        magic , ULONG_CONSTANT, expected = "PK\x01\x02"
        compression,USHORT
        lastmodtime,USHORT
        lastmoddate,USHORT
        crc-32,ULONG
        compressed_size,ULONG
        uncompr_size,ULONG
        file_name_length, USHORT
        extra_field_length, USHORT
        file_comment_length,USHORT
        relative_offset_local_header, ULONG
        filename, STRING, length = file_name_length
        extra_field, STRING, length = extra_field_length
        file_comment, STRING, length = file_comment_length
```

The most important field in the CDFileHeader is *relative_offset_local_header* which is the file offset of the ZipFileHeader for the specific file archived. The reader can then use this to read the ZipFileHeader structure for the archived file:

```
ZipFileHeader:
     magic, ULONG_CONSTANT, expected = 0x04034b50 (\x04\x03PK)
     version, USHORT
     flags, USHORT
     compression_method, USHORT
     lastmodtime, USHORT
     lastmoddate, USHORT
     crc32, ULONG
     compr_size, ULONG
     uncompr_size, ULONG
     name_len, USHORT
     extra_field_len, USHORT
     zip_path, STRING, length is specified in field name_len
     extra_field, STRING, length is specified in field extra_field_len
     data , STRING, length is specified in field compr_size
```

We notice that there are many fields which are repeated in the CDFileHeader and the ZipFileHeader. Depending on the ZIP file creator, some of these may be empty. For example the Linux Zip program does fill the file name in both the CDFileHeader and the ZipFileHeader. This can be used when carving to identify a ZipFileHeader as belonging to the CDFileHeader.

Carving of Zip Files.

From the point of view of a semantic carver, many of the structures in the ZIP file may be useful for positively identifying points. In particular the CDFileHeader structures refer back to the ZipFileHeader for each file in the archive, specifying their file offsets (the *x* coordinate of the mapping function). The problem remains to provide a fix on the image offset (the *y* coordinate). Some of the following constraints can be placed given the DFRWS fragmentation model:

1. The **modulus rule** must hold. This may present a number of potential ZipFileHeader structures – use the other constraints to limit them further.
2. The filename, compressed/uncompressed sizes, date timestamps and other properties which are repeated in both the ZipFileHeader and CDFileHeader must agree if they are not empty. Note that often the ZIP file creator may not have filled the information in both spots.
3. The compressed size of the ZipFileHeaders must agree from the size in the header, and the implied size from the CD. (Archived files are normally saved sequentially so even if the compressed size is not filled in the CD it is possible to tell what it should be.)

Once the correct ZipFileHeader is found on the image for the relevant CDFileHeader a positively identified point may be added to the mapping function.

Preprocessing

Carving is done in 2 steps – first the image is scanned for signatures of possible structures. The following magic signatures are searched: EndCentralDirectory, CDFileHeader, ZipFileHeader. The image offsets for all those are indexed and stored for rapid loading.

Positively identified points are then derived by matching the structures together using the modulus rule and the other rules mentioned above.

Discriminator for Zip Files:

The vast majority of data in the ZIP file is compressed data which means that any fragmentation is likely to occur within the compressed region. The discriminator can use any error detection mechanism for the compression scheme to detect errors within the compressed data.

The ZipFileHeader has a field `compression_method` which specifies the specific compression method employed for each file. Most commonly zlib compression is used, but the standard specifies a large choice of algorithms (some patented or proprietary), as well as *Stored* (no compression). The discriminator needs to support the compression method used in the file, and the compressed data needs to have some kind of error correction built in.

This is not always the case, e.g. the *Stored* compression method (no compression at all) does not provide for a running checksum of the data. The only form of error detection in that case is the CRC32 value which may not be localised enough (i.e. may not detect errors soon enough to determine where the discontinuity may have occurred).

In the DFRWS challenge the files are AES encrypted[2]. This makes it impossible to build a discriminator for the data without access to the pass-phrase. It is simply impossible to tell if the data is valid or corrupted due to the fact that encrypted data is psuedo-random and therefore error detection is not possible (unless the key is known) – Even the CRC32 fields can not be used until the data is decrypted.

There are only 2 ZIP files in the DFRWS image File 1 and File 2:

FileOffset	ImageOffset	Identified Point
0000004	183104516	File file1.dat
1048826	184153338	Central Directory
1048892	184153404	End Central Directory

FileOffset	ImageOffset	Identified Point
0000004	241807876	File file2.dat
1048826	241807610	Central Directory
1048892	241807676	End Central Directory

File 1 appears to be non fragmented although its impossible to be positive of this. An inspection of the data payload in the encrypted file data shows that the entire file appears psuedo-random, adding to the presumption that it is not fragmented.

Future work

When dealing with encrypted files, the discriminator is difficult to build, and probably must use statistical techniques. However, for standard zlib compressed files, a discriminator would be simple to construct, and the carving of unencrypted ZIP files should be as accurate and powerful as PDF files above. Unfortunately, in the present work a complete discriminator was not completed in time.

Use of tool

The tools provided with the paper can be used to recover ZIP files in a similar fashion to the PDF file carver above. The first step is to create an index of the image offsets of important ZIP structures:

```
zip_carver.py -c -i zip.idx dfrws-2007-challenge.img
Found ZipFileHeader in 45182679
Found ZipFileHeader in 61998274
Found ZipFileHeader in 149901830
Found ZipFileHeader in 183104516
Found EndCentralDirectory in 184153404
...
```

Next, the initial map files are constructed:

```
zip_carver.py -m -i zip.idx dfrws-2007-challenge.img
End Central Directory at offset 184153404:
No fragmentation in Central Directory at offset 184153338 discovered... good!
Possible File header at image offset 183104516
Will use Zip File Header at 183104516.
```

2 Details of the encryption scheme used can be found in http://www.cse.ucsd.edu/users/tkohno/papers/WinZip/.

```
End Central Directory at offset 241807676:
Possible Central Directory Starts at 184153338
No fragmentation in Central Directory at offset 241807610 discovered... good!
Possible File header at image offset 183104516
This ZipFileHeader is for file1.dat, while we wanted file2.dat
Possible File header at image offset 241807876
Will use Zip File Header at 241807876.
```

This produces a set of map files:

```
184153404.map  241807676.map
```

When discriminator is written for the ZIP carver, a brute forcer will need to be run at this point. (Just like the PDF carver).

Now it is possible to use the maps to extract the zip file itself:

```
zip_carver.py -e 184153404.zip -M 184153404.map dfrws-2007-challenge.img
```

Part 4: Email formats

This paper deals with two email file formats:

1. The mbox format which is described by the mbox man page - or on http://www.qmail.org/man/man5/mbox.html.
2. The maildir format which is described by http://www.qmail.org/man/man5/maildir.html or the maildir man page.

The following is an overview of these file formats. Both these formats are based around the MIME specification RFC2822 which defines a way for email to contain attachments and deal with multiple language encodings. The mbox format keeps all messages in the same file with a UUCP style "From sender@domain" header before each message. The maildir format keeps each message in its own file within the same directory (The From header is optional for maildir files).

An example of a MIME message is shown below:

```
Return-Path: <owner-tct-users@porcupine.org>
Delivered-To: tct-users@porcupine.org
...
MIME-Version: 1.0
Content-Type: multipart/related;
 boundary="238D813505"

--238D813505
Content-Type: text/html; charset=ISO-8859-1
Content-Transfer-Encoding: 7bit

.....
--238D813505
Content-Type: image/gif;
 name="decipher.gif"
Content-Transfer-Encoding: base64
Content-ID: <BA258A7E43836@e-muzic.net>
Content-Disposition: inline;
 filename="decipher.gif"

R0lGODdhrgJNAYQAAP///09cPIGQ0wieizTPWbRlxGk9JZG6Fb1PBu6Z3sAMITIbCTAf0VsxKzxC....
--238D813505 --
```

The message consists of a group of property/value headers. The **Mime-Version** property specifies that this is a mime message. The **Content-Type** property specifies what type the message is – in this case it is a multi-part message (i.e. it contains other parts in it). The boundary specified is a random string of characters which forms the start, separating and end boundaries of the parts contained in the message.

Once a boundary is encountered, a new part is formed. Each part may have its own **Content-Type** property and may embed other parts within it (in which case a new boundary is specified for the parts which belong to it).

A part may specify its charset – this allows parts to be written in foreign languages. The **Content-Transfer-Encoding** can also be specified. This defines any transformations done to the data before it has been placed into the message. Common encodings include Base64, quoted-printables, 7bit among others. Parsers need to reverse this encodings in order to attain the original data.

Finally the boundary followed by two dashes (--) represents the end of the parts.

Carving MIME messages

MIME messages present a number of difficulties for carving for two basic reasons:

1. The identification of points is difficult due to the lack of explicit file offsets (more on this below).
2. The construction of a discriminator is difficult because each part may contain any other file type. This means that the discriminator needs to support many other file types – to be able to verify the integrity of each part separately. Sometimes the part may contain free text with no enforced structure – making it difficult to detect errors in the text.

Identification of Points

The MIME format does not explicitly specify file offsets for any points, instead points within the file are specified in terms of a boundary. This boundary must be searched for (even when parsing normal mime messages). This makes it difficult to get a fix on the *x* coordinate of the point on the mapping function. In addition the same boundary is used for all parts of the message – making it possible to match several points for the same part.

This problem is illustrated below:

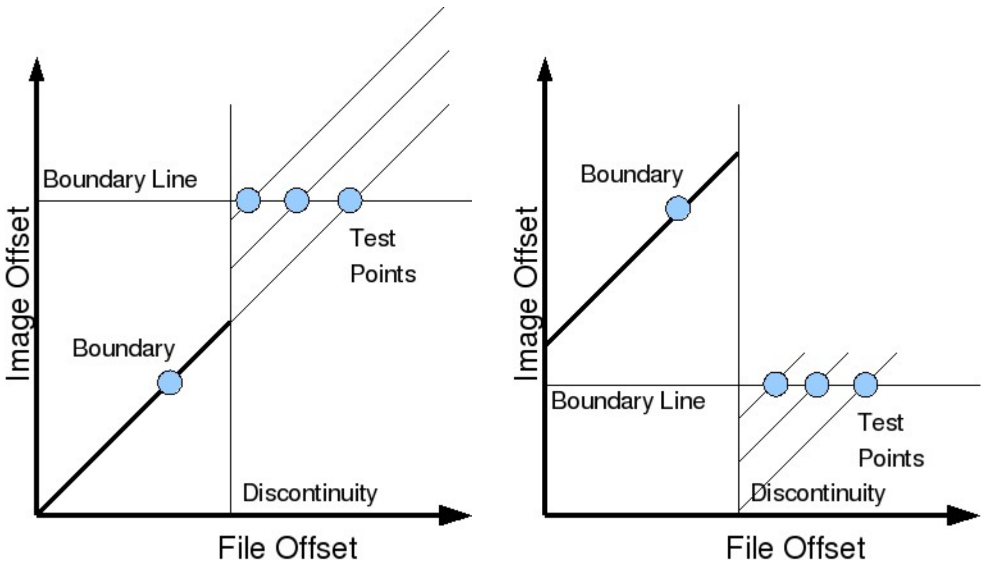

Illustration 9: Carving strategy for Mime messages. Left – where boundary is after discontinuity. Right – where boundary is before start of file.

In the above diagram a mime message is being carved. A discontinuity is detected at a certain point within the file. We know that the file must contain a boundary and we locate this boundary within the image (i.e. we have a fix on its y coordinate – the image offset).

However we do not have a fix on the file offset where the boundary is. This is because the exact offset of the boundary point within the file is not specified (i.e. we don't usually know how large the mime part is). This leads to a number of possible mapping functions as shown in the figure.

Assuming the DFRWS fragmentation model we can further reduce this set to those functions which are sector aligned (because fragmentation can only occur on sector alignment)[3].

Preprocessing

The preprocessing step requires the creation of an index of all the boundary strings within the image. A regular expression is first used to locate all the boundary strings. A second pass is then performed to locate these strings within the image. Due to the large number of strings that need to be indexed (about 1000 unique boundary strings) we make use of the PyFlag indexing engine, although a direct string matching method is also possible. We save the index file to disk for rapid processing.

The Discriminator

Building a discriminator for MIME messages requires the ability to hand off to other discriminators (such as HTML parsers) when needed. The MIME parser itself must be able to handle the various encodings (such as Base64) and code pages. (Some data is not allowed in certain code pages which

3 We are effectively appending the sectors before the boundary line after the discontinuity for each test point, moving backwards and appending more data in turn.

may indicate an error).

In the previous figure we saw how to derive possible mapping functions given a point of discontinuity. But the determination of this discontinuity is difficult. It is often difficult to determine precisely the sector which is out of place – especially if the MIME part is HTML or free flowing text.

Unlike the more structured file types that were presented earlier (PDF, ZIP) the errors in MIME messages are not as localised. This means that typically we need to evaluate the errors of all the possibilities and return those mapping functions with the least errors (unlike PDF where we optimise the mapping function on the fly). This approach is adequate for MIME message which are typically small, so the number of possibilities is not as high.

Because we use boundaries as identified points, we are unable to handle higher order fragmentation within the same message. This is not a problem for smaller messages but can be a problem for large attachments. Future work may use the sub parsers to add additional identified points.

HTML discriminator

Many Mime messages contain HTML parts. This makes it necessary to construct a HTML parser to attempt to detect errors in the HTML. Unfortunately, much of the HTML included in the challenge is invalid anyway, making it difficult to detect a real errors. Existing parsers tend to be more focused on HTML validity, failing completely when invalid HTML is parser (see for example the HTML parser in Python's standard library).

In this paper, a feed parser is developed using the same basic Lexer framework used in the PDF and MIME parsers. Errors within the HTML stream carry certain weights depending on how likely they are to occur in general badly formatted HTML text:

- The lexer maintains a stack of tags and flags errors when tags are not correctly closed (e.g. a sequence like `<a href=...><b>hello</a></b>` will flag an error). Some tags are not generally closed when they should strictly be (e.g. <br> or <p>) - these carry no penalty.
- If characters are unexpected within the current state – the lexer will be stuck, and releasing it will increase the error count – for example a sequence like `<a href=...    <b>`

Part 5: MPEG File Format

The MPEG2 standard is a file format which stores movie and sound information. The MPEG file format is covered by a large number of patents and trademarks. Therefore it seems to be very difficult to find information about the file format itself. Apparently much of the information can be found in ISO/IEC 13818 but that document is not freely available.

There are some partial references on the internet. The most detailed I found is:

http://dvd.sourceforge.net/dvdinfo/mpeghdrs.html

This covers MPEGPS which is the most common file format.

File Format

MPEG streams are broken into packets. There are many types of packets, they all start with 0x000001. The next byte signals which type of packet this is – the rest of the packet is parsed based on this type.

Carving of MPEG files

The MPEG format is very suitable for carving, mainly because MPEG was designed to be transmitted over unreliable media. This means that most players will be able to play an MPEG stream despite many errors. Carved streams are still useful even if they are not perfectly carved.

The following rules can be used to assist in the carving process:

- The packet sizes can be calculated by examining each header. Therefore it is possible to tell immediately when a discontinuity occurs since the next frames header is not present where we expect it to be. We can follow consecutive packets until we detect a discontinuity. We term the sequence of packets as a **run.** An MPEG sequence is a series of packet runs[4].
- The problem remains to associate packet runs to each other and order them correctly. Luckily the MPEG header contains useful fields called PTS or DTS (Presentation Time Stamp or Decode Time Stamp) which are timestamps spread across the file. The fact that timestamps must be monotonically increasing can be used to order runs. If a timestamp is found out of order, we know that a discontinuity is present.
- The modulus rule must hold for MPEG files.

Preprocessing

The carver has a preprocessing phase where runs are identified. The carver can make some initial analysis of each run to extract information which will be useful later on in the carving step. In this step we keep information such as:

1. Initial PTS for the start of the run. This is used to order the run with respect to other runs.

4 This technique is not new, it is already used in photorec for example.

2. Initial offset of the start of the run. This is used with the modulus rule to locate a possible match.
3. The length of this run in frames and bytes.

Here is an example of the output:

```
mpg_carver -s mpg.idx dfrws-2007-challenge.img
1: Start 299520(0), 161 frames (215040 bytes), LKG 512524/514560 (0) with 2036
PTS Range: 6980400 - 7113600, Sectors 585 - 1001
2: Start 1402880(0), 338 frames (471040 bytes), LKG 1871884/1873920 (0) with
2036
PTS Range: 7117200 - 7383600, Sectors 2740 - 3656
3: Start 1874432(0), 486 frames (675840 bytes), LKG 2548236/2550272 (0) with
2036
PTS Range: 7387200 - 7765200, Sectors 3661 - 4977
4: Start 3228160(0), 70 frames (102400 bytes), LKG 3328512/3330560 (0) with 2048
PTS Range: 7772400 - 7819200, Sectors 6305 - 6501
```

This example shows that run 1 starts at offset 299520 bytes which is 0 bytes into the sector. Its 161 frames, and the last known good frame (LKG) goes from offset 512524 to 514560 and is 2036 bytes long. The PTS for this range is given – and we also learn the sector range.

The carver can then be used to extract the runs and try to match them up:

```
mpg_carver -e -l mpg.idx dfrws-2007-challenge.img
carve:356 Checking 1
...
Possible run 1629 comes before run 1 - frame sizes 2036,2036 6422400,6980400
558000
carve:385 Old candidate 209 is better than 1629
...
Sequence: 1612 Out1612.mpg  (6990)
Sequence: 1619 Out1619.mpg Out1618.mpg Out1617.mpg Out999.mpg  (2880868)
```

We see that the carver checks each run and finds runs that can be placed before it. The frame sizes are compared, as well as the PTS difference. We aim to minimise the PTS difference between the end of one run and the start of the other.

Finally we produce a sequence of possible runs and their total size in bytes. The larger this sequence the more likely the video is to be reliably recovered.

Future work

This carver is not very accurate because it uses high level structure of the video files to order the runs together – there is no discriminator which is able to find the precise sector boundary where discontinuity may occur. This may need to be built using and MPEG decoding library. Unfortunately, most MPEG decoding libraries are designed to be resilient to errors and just skip bad frames without providing feedback to the program using the library. This makes the discriminator job much more difficult.

Conclusion

Carving is a difficult problem which is not always solvable. The DFRWS challenge is a particularly difficult image to carve because no filesystem information exists at all, and files are fragmented in an extreme manner. Despite this we were able to show that powerful semantic carvers can be built with varying levels of success depending on the file types.

The general theory of carving was presented – a theory which can be applied to any file type. When examining the design of a carver for different file types, it is possible to look for features of the file format which may be used as positively identified points, and for the construction of high quality discriminators.

The construction of a number of carvers was shown in light of the theory presented. Some file types make carving much simpler for example PDF and ZIP. Other file types are more difficult because they are inherently less structured, for example, MIME.

Results

MD5	Sectors	Size	File Type
3440a0169740b80f5fc ff4cb8641a210	201618-2022231	314300	PDF
8cc4aa0e3e918ed13c4 558c4101a587e	285719-285722, 285727-285761	19617	PDF
e2747d8e103fe6d4870 851ab339080be	287312-289793, 290411-290515	1323915	PDF
ccc56b8b20fcb9fbbf0 05ecac0ef4651	586303-586355, 586713-587729	547772	PDF
Partial	586356-586732, 587780-587828	217130	PDF
20d1dd4a484e992dd1e 439441217f76f	594759-594846, 594600-594759	126106	PDF
18514301d4f7a3df6af c2867b92d40bf	595097-595173, 596772-596790	48603	PDF
d89ba04ad6364330475 4bb6606f56c0b	74835-75107, 74008-74167, 74232-74742	482495	PDF
05e826af4322a28890e 5d42ff054eeed	357626-359674	1048914	ZIP
a11f7ec7abe170ab07a 48a3f69312b8e	102681-102702, 102754-102761	14694	MIME[5]
3b8934ab654d5864aac 2b28a1fb20091	202480-202528, 202449-202478	39507	MIME
8ecc24bb924c86ac302 cab790caeaa3f	290515-290535, 290566-290567	11100	MIME
7f98983f8502b6511f1 36bc3b8744f7f	611719-611745, 616527-616532	16085	MIME
Partial	633745 – 637497, 633632 – 633740, 561213 – 561231	2882560	MPEG2
partial	102857 – 111837, 111891 – 117394	7418880	MPEG2
partial	210077 – 212753, 213825 – 220391, 226403 - 228787	5957120	MPEG2
partial	291114 – 302254, 555984 – 556002, 557749 – 557767,	5730304	MPEG2
partial	208031 – 210071, 212758 – 213760	1561600	MPEG2

5 There are many mime messages, which form a large corpus of mbox files. This table shows some of the more complex messages (i.e. those that do have fragmentation in the middle of the message). There are obviously many messages with no fragmentation which makes their recovery trivial. This is by no means an exhaustive list just an indicative list.

MD5	Sectors	Size	File Type
partial	478923 – 479643, 476335 - 478919	1694720	MPEG2
partial	598557 – 601513, 542163 – 542207	1538062	MPEG2
partial	645861- 646241, 193128 – 193616, 643787 – 644431, 377713 – 378133, 378140 – 378492, 494400 - 495088, 498547 – 498599, 493604 – 494364, 497244 – 498168, 289871 – 290407, 228791 – 229211, 229217 – 229381, 656555 - 657287, 657361 – 657569, 270570 – 270934, 655296 – 655860, 225974 – 226398, 231514 – 232290, 655876 – 656400, 657646 - 658502, 145683 – 145803, 645735 – 645787, 644436 – 644568, 644740 – 645664, 145810 – 146602, 128939 – 129707, 595225 – 595305, 359677 – 361277, 595370 – 596534, 122506 – 122594, 120654 – 121190, 55780 – 56572, 56577 – 57141, 199676 - 200104, 648237 – 648609, 647698 – 648162, 647054 – 647654, 324127 – 324219, 318278 – 319178, 629372 – 629688, 630652 – 631888, 629695 – 629987, 637502 – 637774, 639205 – 639485, 638647 – 639083, 637780 – 638640, 583779 - 584467, 583269 – 583709, 585214 – 585826, 585831 – 586263, 584479 – 585207, 87687 – 87715, 639088 – 639200,	15570944	MPEG2

MD5	Sectors	Size	File Type
	87582 – 87674, 639491 – 640255, 479649 – 480125, 481081 – 481749, 480370 - 481074		