

Submission for the 2006 DFRWS Forensics Challenge

James Madison University Computer Forensics Group

Glenn Henderson

David Horvath

Jeff Jones

{hende2ga,horvatdj,jonesjr}@jmu.edu

Adviser: Prof. Florian Buchholz

buchhofp@jmu.edu

The purpose of the 2006 Digital Forensics Research Workshop File Carving Forensic Challenge was to recover files from raw file system data. All file system information had been removed from the data file and all that was left were the file data itself.

To solve the challenge we developed *FragMend*, a GUI tool that classifies sectors of the raw data and allows an investigator to group them together interactively into files. Files can then be tested to a certain degree within the GUI. Using FragMend and a small number of supporting scripts, we were able to recover 32 files from the raw challenge data: 3 text files, 6 HTML files, 3 PK Zip files, 14 JPG files, and 6 Office documents (5 Word and 1 Excel). No sectors with well-known file headers remained in the list of remaining unallocated sectors.

FragMend is open-source, licensed under the BSD Open Source License and is available for download at Sourceforge¹.

FragMend overview

FragMend is a GUI tool written in Java/Swing. It allows an investigator to specify a source file to work on, which contains the raw sectors to be assembled into files. The sectors (*Fragments*) are initially all put in the 'unallocated fragment list' on the right-hand side of the GUI display. FragMend classifies the fragments from the source stream, identifying file headers and footers as well as performing certain frequency counts on the data. The user can then search and filter on those characteristics and select ranges of sectors to be assembled into files. There are a few high-level parsing routines for common file format (JPG, PKZip,

¹<http://projects.sourceforge.net/fragmend>

MS Compound Document Format) that assist the user in determining the correct sector-composition of a file. There are also ways to display files and sector ranges (as hex, ascii, and for JPG files as the image itself), eliminating the need to use external, third-party tools to verify some files. A more detailed description of the characteristics fields can be found in the next section.

Once the fragments are loaded into the unallocated fragments list the user can search that list or filter the display for certain characteristics. The characteristics of a selected fragment are displayed on the far right of the GUI window. The user can also set selection range start and end markers or select fragments directly in the list. Selected fragments can then be assigned to a new file or appended to the end of an existing file.

The files are displayed in the list on the left hand side (*File list*), and details about the currently selected file can be found in the middle of the GUI window (*File View*). The File View includes a list of the fragments that make up the data for the file. In contrast to the unallocated fragment list, where the fragments are sorted by their position in the source file, the fragments in the files are ordered in the order the user assembled them. There are also buttons to move a fragment up or down in the list or to remove the selected fragments from the file. New fragments assigned from the unallocated list are appended at the bottom of the list².

Fragments, files, as well as fragment selections in the File View can be displayed hex editor style or as an ASCII String. JPG files can also be displayed as an image directly without the need to save the file and open up an external viewing program. Files can be saved to a directory the user can specify.

The carving algorithm used by the FragMend application uses a series of stacks, lists, and specially defined classes to store information about each file type header, footer, and all associated data gathered from a source file. When a header is discovered, a new file type class is created and pushed on a stack. Until another header is discovered, or the footer to match the header, all data fragments are added to the class associated with that header. Once the matching footer is discovered, the file type class is closed out and added to a finished list. This action is performed over and over again until all fragments have been checked and either allocated to a specific file or left unallocated.

For removing extra sectors from a sector range that belong to a file, a "Remove sectors" algorithm is there that creates files with all combinations of a sector block removed from a file. Those files are saved externally and need to be verified with third-party programs.

Finally, there is a functionality to save and load the current state of a project as well as the ability to generate a report as required in the challenge description.

FragMend features

When the user specifies a source input stream, FragMend will break up the stream into fragments and classify them. Once this is done the user can perform certain functions on the fragments and run algorithms to reconstruct the files from the fragments.

²As a future feature we plan to offer drag & drop support as well as the ability to insert fragments at specific positions.

Classification fields

When a fragment is created, the data spanning that fragment (equal to the sector size) is analyzed for the following:

- **JPG Header.** We recognize a valid JPG JFIF header when a fragment starts with the byte sequence `0xffd8ffe04a46494600`.
- **JPG Footer.** A fragment is marked containing a JPG footer when somewhere in the fragment data the sequence `0xffd7` occurs. This may generate a number of false positives and thus a fragment marked containing a JPG footer need not be part of an actual JPG file. Also, for well known footers, such as the JPG footer, we do not take into account the possibility that the fragment may end with the start of an (incomplete) footer, which would be continued at the start of another fragment.
- **JPG Data Marker count.** This field was initially a count of a select number of JPG markers, but did not prove to be too useful for identifying fragments that contain JPG markers. It now contains a count of what we call "JPG data markers". These are the only type of markers (a two-byte value starting with `0xff`) that may appear in the stream data portion of a JPG. Those markers are `0xff00` and `0xffd1-0xffd7` (reset markers). This count helped identify where the JPG stream of a file could continue after encountering junk fragments.
- **PKZip Header.** A fragment is classified to contain a PKZip header when it starts with the byte sequence `0x504b0304`.
- **PKZip Footer.** A fragment is classified to contain a PKZip footer when it contains the byte sequence `0x504b0506`.
- **HTML Header.** A fragment is classified to contain an HTML header when it starts with either the string "`<html`" or "`<!doctype`". Any whitespace at the beginning of the fragment is ignored, and the string comparison does not take case into account.
- **HTML Footer.** A fragment is classified to contain an HTML footer when it contains the string "`</html>`", again ignoring case for the comparison.
- **MS Compound Document Header (Office Header).** A fragment is classified to contain a Compound Document Header when it starts with the byte sequence `0xd0cf11e0a1b11ae1`.
- **Office Footer.** This field is not truly an Office (or Compound Document) footer. We merely check to see if it contains the string "`Word.Document.8`". This is exploiting the fact that many Word documents seem to contain that string at the end of a file and it worked for four of the files in the challenge.
- **Office Root Directory.** A fragment is classified to contain a Compound Document Root Directory if it starts with the Unicode string "`Root Entry`". Thus we check for the byte sequence `0x52006f006f007400200045006e00740072007900`.

- Printable Character count. This field counts the printable ASCII characters of the fragments, i.e. byte ranges 0x20 - 0x7f as well as 0x09, 0x0a, and 0x0d.
- Bracket count. This field counts the occurrence of the two byte values for the angular brackets ("<" and ">") used in HTML and XML code. It turned out to be not too useful for the challenge but could come in handy when one needs to differentiate large amounts of text data from large amounts of HTML/XML data.
- Consecutive number sequence. This field analyzes a fragment for the maximum number of increasing 32-bit number sequences. MS Compound Documents contains plenty of allocation tables, and in many cases the sectors held in these tables are consecutive. Thus this field can help identify fragments that belong to such allocation tables.
- Zero-padded flag. A fragment is classified to be zero-padded if the last four bytes of a fragment are zero. The number four was chosen arbitrarily. It turned out to be not a very useful field as file slack space contained garbage data as opposed to zeros. For data to be salvaged from an operating system that zeroes out slack space (such as Linux) this can be very helpful in finding file boundaries, though.

Functionality

Once all the fragments are classified, the user can perform certain actions with them that are described in the following:

Moving fragments

A basic function yet a central aspect of FragMend is the moving of fragments. Selected fragments can be assigned into a new file by clicking the "New file" button in the unallocated fragments view. If files already exist and one is selected, selected unallocated fragments can be appended to that file ("⌋"). Within a file, fragments can be moved up and down ("Up" and "Down" buttons). However, only one fragment is moved up/down by one position at a time. Future versions of the tool will likely remedy that. Fragments can also be "tossed back" into the unallocated fragments list ("⌋" button). This is useful when one has determined that a certain fragment or range of fragments does not belong to the file. Entire files can be removed from the file list, at which point the fragments belonging to the files are also put back into the unallocated list.

Displaying data

Fragments can currently be viewed in two ways: as ASCII text and in a Hex-editor style with the byte position on the left, the hex value of each byte in the middle, and the printable ASCII value to the right. In each case a new window containing the data is opened, so it is possible to look at multiple views concurrently. The ASCII view is Java's interpretation of a string instantiated by the fragment's data. When the "Hex" or "Ascii" buttons in the File View are clicked, only the data of selected fragments are displayed. Here, in the Ascii view,

we also insert a filler string containing the fragment name (sector number) between the data of each fragment so that one can see where the sector boundaries are. In the file list, pressing the "Hex" and "Ascii" buttons will display the data for the entire file. We caution against using the Hex view for even slightly large files at this point, however, as the construction of the Hex-editor style string seems to choke Java and the tool needs to be restarted. For files the user can also display the data as a JPG image right from the tool without needing to save the file first and using an external program. For very large images, however, Java will throw an "OutOfMemoryException" which makes it still necessary to verify those images internally (such as the 23MB large image contained in the challenge data). If the image is corrupt, sometimes Java's error console output can be helpful as to where the corruption occurs.

Searching for fragments

The search functionality opens up a dialog in which the user can select certain criteria that the fragment needs to possess. Currently, there is no "or" or "not" functionality, but future versions of the tool will likely to have those. On clicking "Next" or "Prev" the GUI tries to locate the next/previous fragment in the unallocated list starting from the current selection.

Filtering unallocated fragments

The unallocated fragment list can also be filtered to display only those fragments that possess certain criteria. The same dialog as for the searching is used with different buttons. For example, filtering the list to display only fragments with a printable character count greater than 499 is very useful to quickly identify most text and HTML fragments³. The filters are "stackable," meaning that one could specify a filter for the printable characters and then filter the resulting fragments for a certain bracket count. In practice stacking filters was not very useful, but with different new classification fields this may become useful in the future.

Determine length/high-level parsing

Pressing the "Get length" button in the File View will attempt to determine the file's length in bytes. For some file types (JPG, PKZip, and Office) we also perform a high-level parsing that tries to verify the file's structure with its current composition. Given that the HTML files in the challenge data could be retrieved rather easily by visual inspection, we did not see a need to do high-level parsing for the HTML files, as well. This probably is not difficult to add, as plenty of HTML/XML parser libraries exist for Java.

For JPG files we make sure that each internal marker is followed by another valid marker. We start by looking for the JPG header at the start of the document. We then get the next marker and its size. If it is not a valid internal PKZip marker the function aborts and outputs an error message that includes the current byte position (within the file) to the console. If

³The user needs to keep in mind, however, that the last fragment belonging to a text or HTML file may well have a much lower printable character count, depending on how much of that sector is used by the file.

the marker is valid, we advance to the next marker using the size value obtained earlier. This is done until we reach the SOS (Start-of-scan) marker. Now the only valid markers we may encounter are 0xff00 and the reset markers. Furthermore, the reset markers can only appear in sequence, meaning R1 has to follow R0, R2 R1, and R0 follows R7. Thus for the scan data we look at every 2-byte sequence and if we find an invalid marker or a reset marker that is out of its order, we again abort with an error message and byte position. The "Byte pos" button in the File View can be used to select the sector that contains that data. If we reach the EOI (End-of-image) marker we return the position of its end.

For PKZip files we start by looking for a local file header. We check if it has a descriptor field and if it does not, get the size of the compressed data. If it does have a descriptor field, the size will be there at the end of the file. Next we get the name for the file, the name length, and the size of the "extra" field (these values are printed out on the console). The file data will start after those fields, and we count the number of bytes from there until we encounter either another local file header, the central directory header, or a span header. If there is a descriptor field, we now extract the size of the compressed data. If the reported size and the actual data size we counted do not match, we abort and give an error message. This is useful for identifying the existence of extra sectors within the file data, but it will not give the exact position. Once the central directory is found we determine the length of it and return the overall file length. Future versions of the function could also compare the values of the central directory against the ones encountered in the local header to make sure that the directory actually belongs to the file, but this proved to be not necessary for the challenge data.

For Office documents we parse the entire Compound Document (OLE2) structure of the file. We first analyze the header sector and retrieve the document parameters such as the OLE2 sector size and the sector numbers where the allocation tables and root directory start. We then construct the Master Sector Allocation Table and from that the Sector Allocation Table (SAT). We then use the SAT to get the sectors for the Short-stream allocation table (SSAT) and attempt to build it. We then try to assemble the short sector data stream and the root directory data stream using the SAT. All this time we output debug information to the console and abort with an error message if something goes wrong (such as an index that is beyond the file data). From the console output the user can get a good idea if everything is in order or not. Once we have the root directory we also parse this, outputting stream and storage names, their size and starting sector. We currently do not recursively obtain storage streams and parse them as directories.

As it turns out, the challenge data contained one Excel spreadsheet that contained a number of extra sectors that we needed to identify (see below). Thus we also parse Excel "Worksheets" if we encounter them as a storage item. The parsing of the workbook is similar to our parsing of a JPG file. We get a record header identifier and the record size, print them out and then jump to the next header according to the size we obtained. We map all of the identifiers described by the OpenOffice documentation, but a few "UNKNOWN" values can still be encountered in valid documents. However, their number is small, and

searching the console output for "UNKNOWN" is how we identified the extra sectors in the Excel spreadsheet file.

As mentioned above, the parsing we perform is only high-level in the sense that we do not re-assemble the image data of a JPG file, we do not try to uncompress the PKZip data, and we certainly do not try to re-assemble an Office document. Such a "true" verification would be immensely useful for the carving algorithm as well as giving up all dependence on external tools. The time permitted for the challenge was simply too short to incorporate such functionality into the tool.

A comment about text file lengths: the data sectors of the challenge contain random data as slack space as opposed to zeros. This random data can be printable characters or even white-space. It is thus impossible to determine exactly where a text or HTML file really ends for sure. One can make educated guesses, but maybe a user on the system from where the data came from really appended the characters "nNV" to the end of "The Adventure of the Copper Beeches" text, who knows? As such this is unlikely. Any white-space, however, is likely to belong to the file. We thus hand-checked the length of the text files and our policy (as well as for HTML files) is that we included all the white-space we found at the end of the text.

Algorithms

For the challenge, we developed two algorithms that automated certain tasks. One is a simple carving algorithm that populates the file list using sectors from the unallocated list. The second is a 'remove' sectors algorithm that generates lots of external files to be tested by a third-party tool.

Carving algorithm

The CarveAction class inside of the FragMend class file performs all actions required to carve individual files out of the entire source file passed to the FragMend program. When an instance of the class is created, several class variables are defined for use during the carving action. One of the base variables used is the genTypeClass class, which is used to store the beginning and ending sector of each file found as well as the source file fragments that make up that file.

Two sets of lists are used for the storage of the fragment related info. The first is a list, which is used as a stack and keeps track of all files in the process of being carved. There are in fact several stacks which store the relevant information and fragments as they are being carved: jpgStack, htmStack, offStack, zipStack, and txtStack. The next type of list used is a simple list used to store all genTypeClass objects that are representative of a complete file. As a complete file is encountered, the file fragment storage is completed, removed from its associated file type stack, and added to the second list.

The analyzeFragments method inside of the carveAction class performs the actual carving of the elements obtained through the fragmentView.getElements() method. This method uses an iterator that runs through each element of the class allFragments list, which allows it to

examine each fragment and determine whether it contains a header, footer, or data element of some defined file type.

The method also uses a stack type list (`hdrList`) to keep track of the last header that was discovered. As a header is discovered, its associated internal integer flag is pushed on the stack. All data fragments are deemed to belong to the specific type of header last discovered unless another header is encountered or the matching file type footer is found. If another header is discovered, its integer flag is pushed on the stack and all data fragments are then associated with that file until the same conditions are met as mentioned above. Once the same file type footer is discovered, the last header flag is removed from the stack, which signals the algorithm that it has found a complete file. Once the complete file is found, all associated fragments are removed from the `allFrag`s list to ensure that a file fragment is allocated to only one file.

After all the fragments located in the `allFrag`s list are processed, the complete files found are located in the `jpgFinal`, `htmFinal`, `offFinal`, `zipFinal`, and `txtFinal` lists. Once the `analyzeFrag`s method is complete, each of the lists mentioned above are used to create `assembledFile` objects and are thus passed back to the user interface for display on the left hand side of the user interface. Any fragment still in the `allFrag`s list is unallocated to any file discovered in the run and will thus remain in the unallocated list on the right hand side of the user interface.

JPG file fragments incur an additional level of checking because if a fragment contains a JPG file header as well as some type of application data, then we must ensure that the next JPG file header is added to the same file. To perform this check, the `hasMarker` method is used to check for the occurrence of hex values `0xd0` through `0xd7`, which denotes that this needs to be included in the overall JPG file and does not signal a JPG file in and of itself. The `genTypeClass` uses a boolean flag (`r1Complete`) to keep track of the occurrence of this type of situation. If the `hasMarker` method returns true, then the next JPG header will be pushed on the same `jpgStack` file type class and will also ensure that the file will have two footers pushed onto the list associated with that file.

There is limited functionality for office files at the present time. Headers are easy to detect; however, footers are presently only detected by the presence of specific strings associated with Word documents. This area will be improved as time goes on.

Removing fragments from files

Inspired by the analysis of the PKZip file that consists of sectors 45015-45386 and sectors 45390-45545, this algorithm takes a start fragment, an end fragment, and a sector count. It then creates files with the sector count number of fragments removed from the file for every position ranging from start to end. The sectors are removed as consecutive blocks. The files are saved externally into a directory the user can specify and the file names contain the number of the starting sector that was removed. If, for example, a file consisted of sectors 1-5, and the user specified a start sector 2, an end sector 3, and a count 2, then two files are saved, one with sectors 2 and 3 removed, the other one with sectors 3 and 4 removed. For file types for which quick, scriptable tests exist as to the file validity (such as PKZip's "`unzip -t`" option), this functionality is extremely useful. See the description below how the

extra sectors of the challenge PKZip file were identified.

In future versions of the tool, if there is a reliable verification method of a file available, the need to save the files externally will be eliminated. The tool can simply report those combinations that result in valid files. Also, we currently only remove one consecutive block of sectors. This is useless if any extra fragments are in more than one location. Thus a variant that removed all possible permutations of the fragments (k out of n) would be desirable but could result in a very long time to compute.

Using FragMend to solve the challenge

Rather than describing how each of the 32 files were recovered in detail we highlight the major techniques we utilized using FragMend to solve the challenge.

Selecting start and end markers

The most basic approach is to search for a fragment that contains a file header and then look for the corresponding footer further down the fragment list. In order for this technique to work the file's sectors need to be consecutive and in order in the unallocated fragments list. Initially, this technique will only those files that are a single block of sectors and are complete and in order from header to footer. This was the case for 14 of the files of the challenge. However, as the unallocated fragment list basically keeps state of the available sectors, in subsequent applications of this technique we can also recover files that "contain" other complete files in their sector range from header to footer. This is the case for example for the PKZip file spanning sectors 28729-29528 and 29896-31368. The sectors 29528-29895 lie right in the "middle" of the zip files sectors. All of these sectors belong to a single HTML file, and once the HTML file was extracted, the start-end marker technique could also recover the zip file. To further make sure that the file footer found at then end matches the initial start sector it is also good practice to search "backwards" from the footer for a header.

The carving algorithm works much in this fashion and thus automates this technique, even though file verification is not automated at this point in time.

Visual inspection: JPG files/text indentation

Visual inspection of the files or parts of files can be a very powerful technique in using FragMend when dealing with text/HTML as well as JPG files. This is because text and JPG files can be easily viewed by the user at the touch of a button.

For JPG files the visual inspection can show corrupt JPG data or indicate that an image lacks data. Furthermore, the console JAVA output can further indicate what is wrong with the image.

Example 1: If we use the technique above to carve out a JPG image that starts at Sector 31475, we find the next JPG footer at Sector 31887. When trying to view the image, JAVA gives an error message: "Corrupt JPEG data: premature end of data segment." When

searching from the position of the JPG footer for a JPG header, we find another header at Sector 31533. Carving out just sectors 31533-31887 produces an image of what looks to be a picture from planet Mars, but the lower 40% of the image is corrupt. It seems likely that two images are interlinked, i.e. they are laid out on the disk as follows:

```
Start of Image1
Image1 data
Start of Image2
Image2 data
Image1 data
End of Image1
Image2 data
End of Image2
```

The Mars image in this case is Image2. To find where data for Image2 ends and data for Image1 starts again, we can successively remove sectors from the sector range 31533-31887 and visually inspect the file until the corruption disappears and Java will switch error messages from "Corrupt JPEG data: bad Huffman code" to "Premature end of JPEG file" this is the case when sectors 31753-31887 are removed. We can now append sectors 31888 (after the JPG footer for image 1) through 32036 (next JPG footer) to the file and the Mars image displays flawlessly. Now selecting all sectors between Sector 31475 and 31887 (which now does not include sectors 31533-31752) and creating a new file for them lets us view an image of a porcupine.

Example 2: Using the Header/footer technique we can identify a JPG file that seems to span sectors 94846-95629. The high-level parsing of the "Get length" function reports no errors, but visual inspection shows that the image ends prematurely. We thus search for the next JPG footer (Sector 96653) and also append the following sectors up to the footer to the image. The sector with the initial JPG footer needs to be removed (Sector 95629). In this case this immediately leads to the proper file, but if there had been any garbage data in sectors 95630 and following, we could have successively removed those sectors and inspected the file.

When visually inspecting text and HTML files, one method obviously is to read the text and make sure that everything is all right. However, this can be cumbersome and time consuming. Another technique we have found useful is to look at average line lengths. Sometimes the text files were created with a specific editor that inserts line breaks after a certain amount of characters for each text line (line wrapping). Or the user who created the file has certain line-wrapping preferences. It can be possible to detect anomalies within text simply by scrolling through the ASCII text display.

Example: An HTML file starts at Sector 4436. If we use the HTML footer found at Sector 4501 to construct a file and visually inspect the ASCII data (using the selection of

all fragments in the File View, not the Ascii view from the file list), we can quickly observe an anomaly for sectors 4456-4485:

```
+++++++ Sector 4455 ++++++
ll side-door, which opened
into a wing of the great hospital. It was familiar ground to me, and I needed
no guiding as we ascended the bleak stone staircase and made our way down the
long corridor with its vista of whitewashed wall and dun-colored doors. Near
the farther end a low arched passage branched away from it and led to the
chemical laboratory.</P>
<P>This was a lofty chamber, lined and littered with countless bottles.</P>
<P>Broad, low tables were scattered about, which bristled with reto
+++++++ Sector 4456 ++++++
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN">
<HTML>
<HEAD>

  <TITLE>1 : Stave 1: Marley's Ghost</TITLE>
  <META NAME="GENERATOR" CONTENT="Microsoft FrontPage 4.0">
<meta name="Microsoft Theme" content="learnlibrary 000, default">
<meta name="Microsoft Border" content="b, default">
</HEAD>
<BODY bgcolor="#D9ECFF" text="#000080" link="#0000FF" vlink="#800080" alink="#FF0000"><!--msnavigation--><table border="0" cellp
+++++++ Sector 4457 ++++++
"top"><!--mstheme--><font face="Verdana">
<!-- Start of Navigation Bar -->

<P><A HREF="index.htm">Prev</A>
| <A HREF="xmascarol_2.htm">Next</A>
| <A HREF="index.htm">Contents</A> </P>

<!--msthemeseparator--><p align="center"></p>
<!-- End of Navigation Bar -->
<A NAME="Section_1"> </A>
<H2><!--mstheme--><font face="Arial, Arial, Helvetica" color="#000000">Stave 1: Marley's Ghost<!--mstheme--></font></H2>

<P>Marley was dead: to beg
+++++++ Sector 4458 ++++++
in with. There is no doubt
whatever about that. The register of his burial was
signed by the clergyman, the clerk, the undertaker,
and the chief mourner. Scrooge signed it. And
Scrooge's name was good upon 'Change, for anything he
chose to put his hand to.</P>

...

+++++++ Sector 4485 ++++++
ys peeping slily down
at Scrooge out of a Gothic window in the wall, became
invisible, and struck the hours and quarters in the
clouds, with tremulous vibrations afterwards as if
its teeth were chattering in its frozen head up there.
The cold became intense. In the main street at the
corner of the court, some labourers were repairing
the gas-pipes, and had lighted a great fire in a brazier,
round which a party of ragged men and boys were
gathered: warming their hands and winking their
eyes before the blaze
+++++++ Sector 4486 ++++++
rts,
test-tubes, and little Bunsen lamps, with their blue flickering flames. There
was only one student in the room, who was bending over a distant table absorbed
in his work. At the sound of our steps he glanced round and sprang to his feet
with a cry of pleasure. &quot;I've found it! I've found it,&quot; he shouted
```

to my companion, running towards us with a test-tube in his hand. "I have found a re-agent which is precipitated by haemoglobin, and by nothing else."
Had he discovered a

Not only is there another HTML header embedded in the file, we can also observe that the line-wrap is noticeably different from sectors 4456-4486 compare to the rest of the file's line-wrap. Two HTML files were similarly interlinked as the two JPG files in the JPG Example1 above. Removing those sectors resulted in a valid HTML file, and the second file could be recovered with the header/footer technique easily.

Google searches

If there is partial data available from a file that contains searchable text and the file is publicly available, performing a Google search on the text can lead to the source of the file and a copy may be retrieved. One can then compare where files differ if the one we try to extract contains extra sectors. One could also search for sectors that match the original file, although currently FragMend has no such functionality.

We used a Google search to identify the Word document that is composed of sectors 36998-37649, 37727-39427, and 39477-40380. The Document, "PREVENTING CRIME: WHAT WORKS, WHAT DOESN'T, WHAT'S PROMISING" can be downloaded at <http://www.ncjrs.gov/works/download.htm>. We were able to identify the extra sectors by comparing hexdump outputs of the original file with the saved file from FragMend.

Originally, we performed a Google search to identify the Excel spreadsheet file after we could recover part of the text. We found a possible match for "FCC Form 477" at <http://www.fcc.gov/Forms/Form477/>, but unfortunately we did this rather late and the FCC had already updated the file to contain data for June 30th, 2006. Thus this technique could not be applied to the Excel file.

Using the parsing debug output

When all the above techniques fail, the debug output generated by the "Get length" functionality is a rich source of information. The function aborts when inconsistencies are found, or the user can scan for them if they cannot be detected automatically.

PKZip file There is a PKZip file whose header starts at Sector 45015 and whose footer is located at Sector 45545. This sector range contains three extra sectors that do not belong to the file. If we applied the "Remove sectors" algorithm to the entire data length, over 528 files would be created to be tested. We can narrow that down first using the "Get length" function. The console output complains that the counted size for the file "2002-29-a-large_web.jpg" (44114) differs from its reported size in the descriptor field (42578) and that the file starts at byte 171477 of the zip file. The difference is exactly three sectors long, so all extra sectors are contained between bytes 171477 and 215591 of the zip file. Using the "Byte pos" function, we can determine that byte 171477 is contained in Sector 45349 and byte 215591 in Sector 45436. If we run the remove sectors algorithm with a start value of

45349, an end value of 45433 and a count of 3, 84 zip files to be tested are created in the specified directory. We can test those files with the following shell script (on a *NIX OS):

```
for name in $( ls ); do unzip -t $name 2>&1 | grep "No errors"; done
```

and this yields the following output:

```
No errors detected in compressed data of zipfile_45387.zip.
```

That means that sectors 45387-45389 need to be removed from the zip file.

Excel spreadsheet The Office Compound Document Header at Sector 2051 indicated that an office document might be present in the subsequent sectors. One quick way to verify this is to try and find the "Directory Stream" which is where the compound document format lists the starting headers of the various "streams" that make up document. This header points to 00000698 (Hex). Which indicates that the Directory stream starts at sector 1688 in the office file. $2051+1688+1$ (since counting starts at 0)=3740 which, unfortunately, was not a document directory stream. This information is easy to determine in FragMend by simply creating a new "Office file" and running pressing the "Get length" button. Information gleaned from the office header will be displayed in standard out.

According to the compound document format the Directory stream begins with the "Root Entry" which is represented by the byte stream 52 00 6F 00 6F 00 74 00 20 00 45 00 6E 00 74 00 72 00 79 00. Since Sector 3740 did not contain this stream, it is not a valid directory stream. A quick search of subsequent sectors indicates that Sector 3760 does begin with this stream and therefore maybe the directory stream.

This directory stream contains four entries: Root, Workbook (indicating that the file is an Excel file), Summary Information, and Document Summary information. Checking the alignment of these streams with their relative position in the file could help indicate where the "extra sectors" are located. The directory entries indicate that the Workbook starts at sector 0000 (Hex). The summary information starts at 679 (Hex) and the Document Summary information starts at 681 (Hex). Sector 0000 (Hex) would be file sector 2052 since Sector 0000 immediately precedes the header. Sector 679 (Hex) and Sector 681 (Hex) are fairly close to the directory stream entry of 681 (Hex), so if the structure is to be consistent the Document Summary should start 1F (Hex) sectors prior to 3760 and the Document summary information should start 1D (Hex) sectors prior. These would be sectors $3760-31=3729$ and $3760-29=3731$ respectively. This was not the case since 3729 contains non-string data, and 3731 is all zeros. This could indicate that this directory stream is not valid.

Going back to the header, it also indicates that stream sector allocation table entries (SSAT) should be located at sectors 689 (Hex) 695 (Hex) and sector 699 (Hex). Allocation table entries are often easy to identify, since in large files, sectors are usually filled sequentially. This means that the allocation table entries usually include long streams of sequential numbers. FragMend has a built in filter to identify long streams of sequential 32-bit numbers in sectors. Using this feature, and setting the threshold at 3 or greater indicates that sectors 3746-3759 are potential allocation tables. However there are 14 and the header only indicates

13 contiguous headers. Further examination indicates that sectors 3746-3757 all contain 128 sequential numbers (full allocation tables) and 3758 contains a list of 120 sequential numbers. This is a strong indication that these sectors are the allocation tables that outline the main part of the file. If this is correct, 3758 is 695 (Hex) and sector 3762 would be file sector 699 (Hex) the final allocation table entry. That sector happens to have a list of 7 sequential numbers making it a good allocation table candidate.

If Sector 3762 is file Sector 699 (Hex), then Sector 698 (Hex) would be Sector 3761. According to the header, this sector should be where the Directory streams starts. A quick examination of that sector indicates that sector 3761 does in fact contain directory entries. Its entries are the same as in 3760 except that the "Document Summary information" is located in file sector 69A (Hex).

Using Sector 3761 as Sector 698 (Hex) the "Summary Information" should start in sector 3730 and indeed that sector contains the string "Microsoft Excel" which is normally contained in the summary information of Excel files. The "Document Summary Information" would now be located in file Sector 69A (Hex) which is real sector 3763. Indeed, this sector appears to begin the "Document Summary information". Using the "zero padded" filter in FragMend it was easy to determine that following 3763 was a series of zero padded sectors ending with sector 3770 which is likely the end of the Excel file.

All of this correlating information indicates that the sector information from 3730 through 3770 is self consistent and consistent with the head information if Sector 3730 is located at file sector 679 (Hex). However, currently sector 3730 is at file sector 68E (Hex). This requires the removal of 21 "extra sectors". As a quick check, FragMend was used to remove the 21 sectors 3709-2729 to see if a valid excel file could be extracted. When this was done, Excel would complain about an invalid file, but would eventually open much of the file.

Since the removing the 21 sectors created a file that Excel was able to partially open, this indicates that we are on the right track and that removing the correct 21 sectors would result in a valid file. The next step is to use the "Get Length" function in FragMend to try and determine where the bad sectors are found. Taking sectors 2051-3770, creating a "new file" and then removing sectors 3709-2729 will create a corrupted Excel file. However, by identifying the file as an Office file and running "Get Length", FragMend will attempt to validate the records in the workbook stream. The output from this function is sent to standard out and examining that output reveals

```
511498(999): Record type: UNKNOWN (54111) Size: 18054
529556(1034): Record type: DELTA Size: 41468
```

Indicating and the end of current file sector 999 there is an UNKNOWN record with a very large size, which is not typical for Excel Workbook records. Going back to the FragMend GUI it can be determined that file sector 1000 is real sector 3051. This now becomes the candidate lead sector for removal.

Now a "New File" is created by taking sectors 2051-3770 and removing 3051-3071. This works and the extracted file is a valid Excel file.

Using only the carving algorithm

When the program is first run with the challenge file as input, there are 97,655 sectors. When the Carve button is pressed the first time, it will carve out files from all of those sectors and was able to allocate all but 6,536 sectors to individual files. The following types and number of files were found during the first run:

```
-- 14 x JPG files
-- 6 x HTML files
-- 6 x Office files
-- 3 x ZIP files
```

The following break-down lists the file type, starting sector, and last sector in the file that was successful during the first carving iteration:

```
HTML: 9; 44
HTML: 4436; 4556
HTML: 4556; 4501
OFF: 7964; 9976
JPG: 8285; 9473
JPG: 11619; 12017
JPG: 12222; 26116
HTML: 27496; 28196
JPG: 27607; 27977
HTML: 28244; 28344
HTML: 29529; 29895
JPG: 36292; 36640
JPG: 41611; 44200
JPG: 43434; 44028
JPG: 45566; 46826
OFF: 45964; 46013
JPG: 46910; 94836
```

The following list identifies the file type, starting sector, and last sector of those files carved that did not work as desired when attempts were made to view them:

```
OFF: 2051; 33397 (openoffice read error)
JPG: 3868; 3908
JPG: 31475; 32036 (partial image of animal in grass)
JPG: 31533; 31887 (partial image of mars)
OFF: 32837; 33390 (openoffice read error)
OFF: 34228; 36236 (openoffice read error)
OFF: 36998; 40371 (openoffice read error)
JPG: 40638; 41609 (partial image of man speaking at conference)
JPG: 94846; 95629 (partial image of Saturn)
```

The above list of files that did not work were removed from the left hand side of the GUI, which in turn added the fragments that made up those files back to the unallocated fragments. Our list of unallocated fragments now stood at 24,804. We then ran the carving algorithm to determine if we could obtain more files that worked with successive runs.

After the second run, the same files that did not work or appear as they should during the first run, were once again added to the list of files. Therefore, we did not have better results after adding the fragments back to the original list and running the program again. We then ran the carve algorithm on all the 6,536 sectors that were left after the first two runs of the program. The third run did not produce any further files found and, in fact, did not allocate any of the unallocated sectors to new files.

After the three runs, there appear to be several sectors that are detected to be JPG footers; however, without JPG headers with which to match them, they must stay on the unallocated list.

Limitations and future work

In addition to the limitations already discussed in the other sections there are a few other things that should be addressed in future versions of the tool. The tool was designed to generally handle arbitrary data, but solving the challenge was a major influence regarding its current functionality and the time for the challenge was rather short. The following items should be addressed as future work:

- User friendliness. There is currently very little error checking going on. We currently trust that the user has enough knowledge of the tool (or will have to learn it) not to crash the GUI. As such, we do not yet disable buttons when pressing them would crash the program. Also, we currently trust that the user inputs values in the proper formats and do not verify any input. Fixing these shortcomings should be one of the first things to be done in the future.
- Lack of real verification methods. Our "Get length" function only performs high-level parsing concentrating on internal headers and markers, often ignoring much of the actual application data. Functionality that parses and tries to reconstruct the entire file would be of immense help for automating the carving algorithm to the point where successive iterations will no longer yield any more correct files. It would also help narrowing down the point of failure if there is missing/extra data in the file sectors.
- Lack of connection between console output and GUI. Currently the user has to inspect the console output and then map that information to fragment positions. For future versions of FragMend we plan to get rid of console output and point out milestones or anomalies of a file within the GUI.
- Efficiency. Moving a large number of sectors takes quite a bit of time. The same is true for classifying the fragments and displaying the data. Thus we will need better data structures and algorithms to increase efficiency.

This is merely a short list of things we plan to implement in the near future. We believe there are many more features that can be added to the tool. The tool is available at <http://sourceforge.net/projects/fragmend>, and we will welcome helping hands for development.

References

- [1] John Miano. *Compressed Image File Formats*. Addison Wesley, 1999.
- [2] Daniel Rentz. Microsoft Compound Document File Format. <http://sc.openoffice.org/compdocfileformat.pdf>, 2005.
- [3] Daniel Rentz. Microsoft Excel File Format. <http://sc.openoffice.org/excelfileformat.pdf>, 2005.
- [4] .zip file format specification. http://www.pkware.com/business_and_developers/developer/appnote/.