available at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/diin

Digital Investigation

Carving contiguous and fragmented files with fast object validation

Simson L. Garfinkel[a,b]

[a]*Naval Postgraduate School, Monterey, CA, USA*
[b]*Center for Research on Computation and Society, School of Engineering and Applied Sciences, Harvard University, Cambridge, MA, USA*

Keywords:
DFRWS Carving Challenge
File carving
Fragmentation
Object validation
Bifragment Gap Carving

ABSTRACT

"File carving" reconstructs files based on their content, rather than using metadata that points to the content. Carving is widely used for forensics and data recovery, but no file carvers can automatically reassemble fragmented files. We survey files from more than 300 hard drives acquired on the secondary market and show that the ability to reassemble fragmented files is an important requirement for forensic work. Next we analyze the file carving problem, arguing that rapid, accurate carving is best performed by a multi-tier decision problem that seeks to quickly validate or discard candidate byte strings – "objects" – from the media to be carved. Validators for the JPEG, Microsoft OLE (MSOLE) and ZIP file formats are discussed. Finally, we show how high speed validators can be used to reassemble fragmented files.

1. Introduction

"File carving" reconstructs files based on their content, rather than using metadata that points to the content. File carving is useful for both data recovery and computer forensics. For data recovery, carving can recover files from a device that has been damaged – for example, a hard disk where the sectors containing the disk's directory or Master File Table are no longer readable. In forensic practice, file carving can recover files that have been deleted and have had their directory entries reallocated to other files, but for which the data sectors themselves have not yet been overwritten.

Because it has application in both data recovery and computer forensics, file carving is an important problem. File carving is also challenging. First, the files to be carved must be recognized in the disk image. Next, some process must establish if the files are intact or not. Finally, the files must be copied out of the disk image and presented to the examiner or analyst in a manner that makes sense. The first two of these activities require specific, in-depth knowledge for each file type. The third requires a good user interface.

Most of today's file carving programs share two important limitations. First and most important, these programs can only carve data files that are contiguous – that is, they can only create new carved files by extracting sequential ranges of bytes from the original image file. Second, carvers do not perform extensive validation on the files that they carve and, as a result, present the examiner with many *false positives* – files that the carver presents as intact data files, but which in fact contain invalid data and cannot be displayed.

Carrier et al. (2006) created the 2006 DFRWS Carving Challenge to spur innovation in carving algorithms. The Challenge consisted of a 49,999,872 byte "Challenge file" containing data blocks from text files, Microsoft Office files, JPEG files, and ZIP archives, but having no file system metadata such as inodes or directory entries. Some of the files in the Challenge were contiguous, while others were split into two or three fragments. The goal was to reconstruct the original files that had been used to create the Challenge.

E-mail address: simsong@acm.org
1742-2876/$ – see front matter © 2007 DFRWS. Published by Elsevier Ltd. All rights reserved.
doi:10.1016/j.diin.2007.06.017

1.1. This paper's contribution

This paper significantly advances our understanding of the carving problem in three ways.

First, we present a detailed survey of file system fragmentation statistics from more than 300 active file systems from drives that were acquired on the secondary market between 1998 and 2006. These results are important for understanding the various kinds of file fragmentation scenarios that appear on computer systems that have sustained actual use, as opposed to simulated use in the laboratory. With this understanding we can more productively guide research efforts into the carving problem.

Second, this paper considers the ranges of options available for carving tools to validate carved data – that is, to distinguish files that are actually valid carved objects from a haphazard collection of data blocks that is a combination of different files. These options are used to develop several proposed carving algorithms.

Third, this paper discusses the results of applying these algorithms to the DFRWS 2006 Carving Challenge. Even though the Challenge was artificially constructed, we feel that any algorithm that can reassemble the fragmented files in the DFRWS 2006 Challenge will also be able to reassemble fragmented files in an FAT or NTFS-formatted file system. The converse is not true: any algorithm that is tuned specifically for those file systems is unlikely to work on the Challenge data set or other carving problems, such as carving memory dumps, because those other media will not have the FAT and NTFS-specific features.

1.2. Outline of paper

Section 2 reports related work. Section 3 presents the results of our file fragmentation survey. Section 4 discusses the need for object validation to improve carving, and discusses algorithms that object validation can make possible. Section 5 presents carving algorithms that use object validation and presents our experience in applying them to the 2006 Challenge.

2. Related work

The Defense Computer Forensics Lab developed a carving program called CarvThis in 1999. That program inspired Special Agent Kris Kendall to develop a proof-of-concept carving program in March 2001 called *snarfit*. Special Agent Jesse Kornblum joined Kendall while both were at the US Air Force Office of Special Investigations and the resulting program, Foremost, was released as an open source carving tool. After several years without development, Foremost was extended by Mikus (2005) while working on his master's thesis at the Naval Postgraduate School. Most notable was his implementation of a module with specific knowledge of the Microsoft OLE (MSOLE) file format and the integration of file system-specific techniques. Version 1.4 of Foremost was released in February 2007.

While Foremost was temporarily abandoned by its original authors, Richard and Roussev (2005) reimplemented the program with the goal of enhancing performance and decreasing memory usage. The resulting tool was called Scalpel. Scalpel version 1.60 was released in December 2006.

Garfinkel (2006a) introduced several techniques for carving fragmented files in his submission to the 2006 Challenge. This paper improves upon that work with a detailed analysis of his approach and a justification of the approach using Garfinkel's corpus of used drive images.

CarvFS and LibCarvPath are virtual file system implementations that provide for ''zero-storage carving'' – that is, the ability to refer to carved data inside the original disk image without the need to copy it into a second file for validation (Sourceforge).

Douceur and Bolosky (1999) conducted a study of 10,568 file systems from 4801 personal computers running Microsoft Windows at Microsoft, but did not consider file fragmentation.

The carving terminology in this paper was developed jointly with Joachim Metz.

3. Fragmentation in the wild

In this section we present statistics about the incidence of file fragmentation on actual file systems recovered from used hard drives purchased on the secondary market. The source material for this analysis was Garfinkel's (2006b) used hard drive corpus, a copy of which was obtained for this paper.

Garfinkel's corpus contains drive images collected over an eight year period (1998–2006) from the US, Canada, England, France, Germany, Greece, Bosnia, and New Zealand. Many of the drives were purchased on eBay. Although approximately one-third of the drives in the corpus were sanitized before they were sold, a significant number contain the data that were on the drive at the time of their decomissioning. The kinds of fragmentation patterns observed on those drives are representative of fragmentation patterns found in drives of forensic interest.

3.1. Experimental methodology

Garfinkel's corpus was delivered as a series of AFF (Garfinkel et al., 2006) files ranging between 100 K and 20 G bytes in length. Analysis was performed using Carrier's Sleuth Kit (Carrier, 2005a) and a file walking program that was specially written for this project. Results were stored in text files (one for each drive) which were imported into an SQL database, where further analysis was performed.

Sleuth Kit was able to identify active file systems on 449 of the disk images in the Garfinkel corpus. But many drives in the Garfinkel corpus were either completely blank or else had been completely wiped and then formatted with an FAT or NTFS file system. Only 324 drives contained more than five files. On these drives Sleuth Kit was able to identify 2,204,139 files with filenames, of which 2,143,553 files had associated data. This subset of files accounted 892 GB recoverable data.

3.2. Fragmentation distribution

Overall 125,659 (6%) of the files we recovered from the corpus were fragmented.

Table 1 – Distribution of file fragmentation for files on drives with more than five files

Fraction of files on drive that are fragmented	Total drives	Total named files
$f = 0.00\%$	145	17,267
$0 < f \leq 0.01$	42	459,229
$0.01 < f \leq 0.10$	107	1,115,390
$0.1 < f \leq 1.0$	30	412,297
	324	2,004,183

Drives were not equally fragmented: roughly half of the drives had not a single fragmented file! And 30 drives had more than 10% of their files fragmented into two or more pieces (Table 1).

Despite the fact that fragmentation appears to be relatively rare on today's file systems, we nevertheless feel that the ability to carve fragmented files is an important capability that has not been addressed by today's carving tools. This is because *files of interest in forensic investigations are more likely to be fragmented than other kinds of files, for reasons explained below*:

Modern operating systems try to write files without fragmentation because these files are faster to write and to read. But there are three conditions under which an operating system must write a file with two or more fragments:

1. There may be no contiguous region of sectors on the media large enough to hold the file without fragmentation. This is likely if a drive has been in use a long time, is filled near capacity, and has had many files added and deleted in more-or-less random order over time.
2. If data are appended to an existing file, there may not be sufficient unallocated sectors at the end of the file to accommodate the new data. In this case some file systems may relocate the original file, but most will simply write the appended data to another location.
3. The file system itself may not support writing files of a certain size in a contiguous manner. For example, the Unix File System will fragment files that are long or have bytes at the end of the file that will not fit into an even number of sectors (Carrier, 2005b). Not surprisingly, we found that files on UFS volumes were far more likely to be fragmented than those on FAT or NTFS volumes (Table 2).

Table 2 – Fragmentation of files that could be recovered by Sleuth Kit, by file system type, for file systems containing more than five files

	FAT[a]	NTFS	UFS
# File systems	219	51	5
# Fragments	Number of files		
(Contiguous)	1,286,459	521,663	70,222
2	25,154	22,984	10,932
3	4932	6474	1047
4	2473	3653	408
5–10	4340	13,139	658
11–20	1593	7880	94
21–100	1246	11,901	13
101–1000	186	5953	0
1001–	2	590	0
Total files	1,326,385	594,237	83,374

Note: this table omits the eight files found on the single UFS2 file system in the corpus (drive 620) and the 16 files found on the single EXT3 file system (drive 1041). The table also omits empty files 0 bytes in length, since they have zero fragments.

[a] Includes FAT12, FAT16 and FAT32.

3.3. *Fragmentation by file extension*

We hypothesized that different kinds of files would exhibit different kinds of fragmentation patterns. In particular, we thought that files that were installed as part of the operating system would have low fragmentation rates. Conversely, large files created by the user, log files, and files written to as databases (such as DOC, XLS and PST files), would likely have high fragmentation rates.

Table 3 shows a cross-tabulation of fragmentation rate by file extension for the files in the corpus. As suspected, high fragmentation rates were seen for log files and PST files, but we were surprised to find that the most highly fragmented files were TMP files. We suspect that this is because many TMP files were quite large (note the high standard deviation for file size) and that temporary files are created throughout a system's lifetime – so some were created after small files scattered throughout the drive made it impossible to write the TMP file without fragmentation.

For this purpose of this paper, it is highly significant that the file types likely to be of interest by forensic examiners (e.g. AVI, DOC, JPEG and PST) had significantly higher fragmentation rates than those files that are of little interest (BMP, HLP, INF, and INI). Thus is behooves the research community to develop algorithms that work with fragmented files?

3.4. *Files split into two fragments*

We use the term *bifragmented* to describe a file that is split into two fragments. Bifragmented files represent an attractive target for automated carving because these files can be carved using relatively straightforward algorithms discussed in Section 5. Table 7 shows the number of bifragmented files, the average file size, and the maximum file size observed in the corpus for the 20 most popular file extension.

We performed a histogram analysis of the most common gap sizes between the first and the second fragment and present the overall findings in Table 4. Tables 5 and 6 show common gap sizes for JPEG and HTML files, respectively. The gaps tended to represent 1, 2, 4 or 8 512-byte sectors. We hypothesize that this gap corresponds to a single FAT or NTFS clusters that had been already allocated to another file when the operating system was writing the file that was fragmented. This hypothesis appears partially confirmed by Table 8: with more files with a gap of eight blocks in Table 8 than a gap of eight sectors in Table 4, it appears that some of the files with gaps of 16 or 32 sectors in Table 4 were actually on file systems with a cluster size of two or four sectors.

Table 3 – Per-file fragmentation seen in the disk corpus for selected file extensions

Ext.	File size in bytes			# Drives in corpus	# Files in corpus	Number of files with			Percent frag. (%)
	Avg.	Std. dev.	Maximum			Two frag.	Three frag.	>Three frags.	
art	2483	4285	171,534	18	10,631	74	8	101	1
avi	10,218,679	51,355,670	734,117,888	94	998	17	6	185	20
bmp	66,053	393,456	12,032,066	160	26,018	367	129	1630	8
chm	120,804	408,393	15,867,327	113	12,033	306	66	933	10
cnt	23,425	1,968,647	201,326,592	141	10,458	13	9	426	4
cur	1714	59,251	5,429,170	89	12,265	0	1	100	0
dat	408,286	31,151,890	4,737,728,512	220	23,193	784	252	3,205	18
dll	165,799	375,338	18,200,064	183	227,415	7507	2211	27,490	16
dl_	65,905	249,511	8,422,595	71	19,537	79	21	161	1
doc	85,358	1,597,635	135,477,136	158	7673	209	65	1100	17
exe	299,249	6,190,411	1,166,868,544	236	78,646	2352	827	8648	15
gif	5328	251,095	145,752,064	139	357,713	2990	795	27,581	8
hlp	95,480	288,297	8,121,820	180	26,374	476	99	1467	7
html	12,761	135,179	18,911,232	146	125,222	4085	929	10,330	12
inf	23,849	65,214	4,044,538	175	73,988	683	217	3185	5
ini	271,512	41,025,655	6,440,357,888	193	24,643	228	57	2221	10
jpeg	31,137	159,456	24,265,736	129	108,539	2999	400	13,973	16
js	12,870	249,835	16,289,792	108	18,508	535	247	1712	13
lnk	1561	52,971	5,373,952	139	29,229	227	112	3962	14
log	109,571	731,137	39,808,746	235	7058	394	98	1725	31
mdb	915,714	2,821,426	32,440,320	93	402	30	14	68	27
mpeg	2,639,141	5,714,052	60,958,724	14	168	4	3	22	17
pnf	37,040	95,387	7,254,942	107	21,385	7583	108	1183	41
png	13,813	56,818	3,436,437	85	9995	175	93	300	5
ppt	137,167	861,927	16,913,920	123	1120	20	6	73	8
pst	8,839,321	50,856,271	421,249,024	31	70	6	6	29	58
sys	687,401	18,313,906	1,610,612,736	286	22,348	513	134	2168	12
tmp	91,460	759,610	52,428,800	157	57,007	452	154	37,376	66
ttf	134,393	651,666	24,131,012	145	16,943	540	122	906	9
txt	6141	98,558	10,499,104	252	64,315	496	125	6726	11
vxd	63,594	140,152	1,464,566	133	11,910	174	57	1547	14
wav	145,406	1,479,044	65,658,924	157	24,550	584	143	1721	9
wmf	15,649	28,085	1,884,160	106	77,694	418	86	1430	2
xls	149,851	368,855	3,787,776	136	2159	67	28	148	11
xml	30,366	353,767	6,966,403	81	13,404	241	86	1219	11

3.5. *Highly fragmented files*

A small number of drives in the corpus had files that were highly fragmented. A total of 6731 files on 63 drives had more than 100 fragments, while 592 files on 12 drives had more than 1000. Surprisingly, most of these files were large system DLLs and CAB files. It appears that these files resulted from system patches and upgrades being applied to drives that were already highly fragmented. Although we lack algorithms to reassemble highly fragmented files, the sectors

Table 4 – Gap size distribution for all bifragmented files

# Files	Fragments gap	
	Bytes	Sectors
4272	4096	8
1535	8192	16
1344	16,384	32
921	2	0
817	32,768	64
441	12,288	24
354	40,960	80
328	24,576	48
305	49,152	96
284	20,480	40

Table 5 – Gap distribution for JPEG bifragmented files

# Files	Fragments gap	
	Bytes	Sectors
99	2	0
88	4096	8
40	16,384	32
38	8192	16
38	651,264	1272
23	59	0
16	32,768	64
14	24,576	48
14	20,480	40
11	12,288	24
11	122,880	240
11	131,072	256
11	28,672	56

Table 6 – Gap distribution for HTML bifragmented files

# Files	Fragments gap	
	Bytes	Sectors
165	4096	8
126	77,824	152
90	79,872	156
77	8192	16
74	16,384	32
52	75,776	148
48	32,768	64
39	81,920	160
31	12,288	24
25	28,672	56

Table 8 – The most common gap sizes for all bifragmented files, expressed in terms of file system allocation block size

# Files	Gap blocks
4327	8
1519	32
1431	16
697	0
649	64
470	24
398	40
328	48
296	96
277	80

Block sizes ranged from 512 to 4096 bytes.

belonging to well-known DLLs and CAB files could be eliminated from a file being carved if one had a database of hash codes for every sector of well-known files.

3.6. *Fragmentation and volume size*

An anonymous reviewer of an earlier version of this paper suggested that large hard drives are less likely to have fragmented files than the smaller hard drives that are typically sold on eBay, which was the source of drives in the Garfinkel corpus. In this sample, 303 drives were smaller than 20 GB, while only 21 were larger.

To test this hypothesis, we computed the percentage of JPEGs that had two or more fragments on all of our drives. Overall we found that smaller drives did tend to have more fragmentation, but that some of the most highly fragmented drives were drives in 10–20 GB range. For example, the drive with the highest percentage of fragmented JPEGs was #1028, a 14 GB drive; 43% of this drive's 2517 JPEGs were fragmented. A 4.3 GB drive had 34% fragmentation, followed by 33% of a 9 GB drive. Our conclusion is that fragmentation does appear to go down as drive size increases, but that many large drives have significant amounts of fragmentation, and this fragmentation may affect files of critical interest to forensic investigators.

Table 7 – Most common files in corpus consisting of two fragments, by file extension

Ext.	File count	Size of files with two fragments		
		Avg.	Std. dev.	Max.
pnf	7583	41,583	81,108	1,317,368
dll	7507	220,640	384,246	9,857,608
html	4085	25,961	61,267	2,505,490
jpeg	2999	29,477	177,511	6,601,153
gif	2990	19,826	92,231	3,973,951
exe	2352	398,867	4,350,378	206,199,144
1	1125	57,475	130,630	1,998,576
dat	784	290,892	672,600	7,793,936
z	716	74,353	340,808	6,248,869
h	693	16,454	12,206	110,592
inf	683	79,578	101,448	522,916
swf	591	59,967	117,133	1,155,989
wav	584	1,921,482	6,300,175	39,203,180
ttf	540	163,854	649,919	10,499,104
js	535	18,595	28,393	466,944
sys	513	1,276,323	12,446,966	150,994,944
txt	496	32,724	271,185	5,978,896
hlp	476	184,897	375,150	3,580,078
tmp	452	206,037	770,690	8,388,608
so	440	103,939	205,617	1,501,148
…	…	…	…	…

4. Object validation

In order to carve bytes from a disk image into a new disk file, it is necessary to have some sort of process for selecting and validating the carved bytes. Foremost and Scalpel use sequences of bytes at the beginning and end of certain file formats (file *headers* and *footers*); Mikus enhanced Foremost with a validator for the Microsoft Office internal file structure. When the carving program finds a sequence of bytes that matches the desired requirements, the bytes are stored in a file which is then manually opened and examined.

In this paper we use the term *object validation* to describe the process of determining which sequences of bytes represent valid Microsoft Office files, JPEGs, or other kinds of data object sought by the forensic investigator. Object validation is a superset of *file validation*, because in many cases it is possible to extract, validate and ultimately use meaningful components from within a file – for example, extracting a JPEG image embedded within a Word file, or even extracting a JPEG icon from within a larger JPEG file.

4.1. *Fast object validation*

Object validation is a *decision problem* in which the validator attempts to determine if a sequence of bytes is a valid file, by which we mean that a target program (e.g. Microsoft Word) can open the file and display sensible information without generating an error.

If one had a fast computer and a fast object validation algorithm, a simple way to find all contiguous objects that could be carved from a disk image would be to pass all possible substrings of bytes from the disk image through the validator and keep the sequences that validate. A disk with n bytes has $(n)(n+1)/2$ possible strings; thus, a 200 GB hard drive would require 2.0×10^{22} different validations.

A carefully designed carver can eliminate the vast majority of byte sequences without even trying them. For example, if it is known that sequences can only start on sector boundaries, then $511/512 = 99.8\%$ of the strings need never be tried. If the validator does not generate an error if additional data are appended to the end of a valid data object, then the carver can simply try the set of all byte sequences that start on a sector boundary and extend to the end of the disk image; for each valid sequence found, the carver can perform a binary search to rapidly find the minimum number of bytes necessary for validation. These two assumptions hold when carving contiguous JPEG images from FAT and NTFS file systems, since both will only allocate JPEG at the start of sectors (512-byte boundaries) and the JPEG decompressor can recognize the end of a file. Together these two shortcuts would reduce the number of validation operations for a 200 GB drive from 1.9×10^{22} to 4×10^{8}, plus roughly 40 validations for each object that is identified. As discussed in the following section, all JPEG files begin with a distinctive 4-byte sequence. Checking for these sequences is extremely fast. Only the object candidates with these headers need be subjected to more time consuming validations. As a result, all of the contiguous JPEGs in a disk image file can frequently be found as quickly as the file can be loaded into the memory system of a modern computer – typically an hour for every 50 GB or so.

4.1.1. *Validating headers and footers*

Byte-for-byte comparisons are among the fastest operations that modern computers can perform. Thus, verifying static headers and footers (if they are present) is an excellent first pass of any validation algorithm.

For example, all JPEG files begin with the hexadecimal sequence `FF DE FF` followed by an `E0` or `E1`; all JPEG files end with the hexadecimal sequence `FF D9`. The chance of these patterns occurring randomly in an arbitrary object is 2 in 2^{48}. A JPEG object validator that checks for these static sequences can quickly discard most candidate objects.

Header/footer validation is not sufficient, however, since by definition it ignores the most of the file's contents. Header/footer validation would not discover sectors that are inserted, deleted or modified between the header and the footer because these sectors are never examined. Thus, header/footer validation should only be used to reject a data object. Objects that pass must be processed with slower, more exhaustive algorithms.

4.1.2. *Validating container structures*

Many files of forensic interest are in fact *container files* that can have several internal sections. For example, JPEG files contain metadata, color tables, and finally the Huffman-encoded image (Hamilton, 1992). ZIP files contain a directory and multiple compressed files (Katz, 2006). Microsoft Word files contain a Master Sector Allocation Table (MSAT), a Sector Allocation Table (SAT), a Short Sector Allocation Table (SSAT), a directory, and one or more data streams (Rentz, 2006).

As with validating headers and footers, validating container structures can be exceedingly fast. Many container structures have integers and pointers; validating these requires little more than checking to see if an integer is within a predefined range or if a pointer points to another valid structure.

For example the first sector of an Office file contains a CDH header. The CDH must contain a hex `FE` as the 29th character and a `FF` as the 30th character; these bytes are ideal candidates for header validation. Once a candidate CDH is found, the pointers can be interpreted. If any of these numbers are negative or larger than the length of the object divided by 512, the CDH is not valid, and a Microsoft Office file validator can reject the object. Checking these and other structures inside an object can be very fast if the entire object is resident in memory.

Information in the container structures can also provide guidance to the carver. For example, when a candidate CDH is found in the drive image, the values of the MSAT and SSAT pointers can be used to place a lower bound on the size of the file – if the MSAT points to sector 1000, then the file must be at least 512,000 bytes long. Being able to set a lower bound is not important when performing header/maximum file size carving (Section 5.1.2), but it is important when performing Fragment Recovery Carving (Section 5.2).

Container structure validation is more likely than header/footer validation to detect incorrect byte sequences or sectors inside the object being validated because more bytes are examined. But we have seen many examples of carving candidates that have valid container structures but which nevertheless cannot be opened by Microsoft Word – or which open in Microsoft Word but then display text that is obviously wrong.

4.1.3. *Validating with decompression*

Once the container structures are validated, the next step is to validate the actual data that are contained. This is more computationally intensive, but in many cases it will discover internal inconsistencies that allow the validator to reject the candidate object.

For example, the last section of a JPEG-formatted file consists of a Huffman-coded representation of the picture. If this section cannot be decompressed, the picture cannot be displayed and the object can be deemed invalid. A computationally intensive way to do this is by decompressing the picture; a faster way is by examining all of the Huffman symbols and checking to see if they are valid or not.

The text sections of a Microsoft Office file can likewise be extracted and used for validation. If the text is not valid – for example, if it contains invalid characters – then the object validator rejects.

Our original plan for carving fragmented JPEGs was to decompress a run of sectors until we encountered an error. This, we thought, would tell us that the previous sector was the last valid sector in the run. We could then search the disk image for a sector that, appended to the current run, allowed the decompressor to continue. But we were wrong. We discovered that the JPEG decompressor will frequently decompress corrupt data for many sectors before detecting an error. For example, the 2006 Challenge included a photo from Mars that was present in two fragments, from sectors

31,533–31,752 and 31,888–32,773. The JPEG decompressor supplied with the contiguous stream of sectors starting at sector 31,533 does not generate an error until it reaches sector 31,761. The 9 sectors in the range 31,733–31,760 decompress as valid data, even though they are obviously wrong, a fact readily apparent by examining the left hand image of Fig. 1.

Despite the fact that the JPEG decompressor will decompress many invalid sectors before realizing the problem, we have never seen a case of corrupted data for which the decompressor concluded that the entire JPEG had been properly decompressed and returned without error. Thus, we have been quite successful in using the decompressor as a validator.

Using this JPEG validator, we were able to build a carving tool that can automatically carve both the contiguous and the fragmented JPEG files on the DFRWS 2006 with no false positives. The six contiguous JPEGs starting at sectors of 8285, 12,222, 27,607, 36,292, 43,434 and 46,910 are identified and carved in 6 s on our reference hardware with no false positives. Solving the split files takes longer, but the time required is minutes, not hours, using the Bifragment Gap Carving algorithm presented in Section 5.2.1.

4.1.4. *Semantic validation*

We believe that it should be possible to use aspects of English and other human languages to automatically validate data objects. For example, if the letters `hospi` appear as the last five characters in a sector and the letters `tals` appear as the first three characters of the next sector, then it is reasonable to assume that what has happened is that the word `hospitals` has been split across a sector boundary. This assumption is especially likely if the document being recovered is about health care policy. The two sectors are likely to be consecutive in the final carved file even if they are separated by 16 sectors containing French poetry; in that case, the French is probably from another file.

Garfinkel solved part of the 2006 Challenge using a manually tuned corpus recognizer that based its decisions on vocabulary unique to each text in question. Although this is an interesting approach, automating it is currently beyond our abilities.

4.1.5. *Manual validation*

One might think that the most accurate way to validate an object is to attempt opening the file using the target program itself. This is still not definitive, however, as Word and Excel will open files that contain substituted sectors (although in our experience they will not open files with omitted or inserted sectors). Not only must the file be opened, it must be examined with human eyes. Since this is not possible in an automated framework, even our best object validators will have the occasional false positive.

4.2. *A pluggable validator framework*

We have developed a pluggable object validator framework that implements each object validator as a C++ class. The framework allows the validators to perform fast operations first, and slow operations only if the fast ones succeed, and provides for feedback from the validator to the carvers.

4.2.1. *Validator return values*

Classic decision problems return either an ACCEPT or a REJECT; our validator framework supports a richer set of returns to allow for more efficient carvers.

Every validator must implement a method that returns on the following two values:

`V_OK` The supplied string validates.
`V_ERR` The supplied string does not validate.

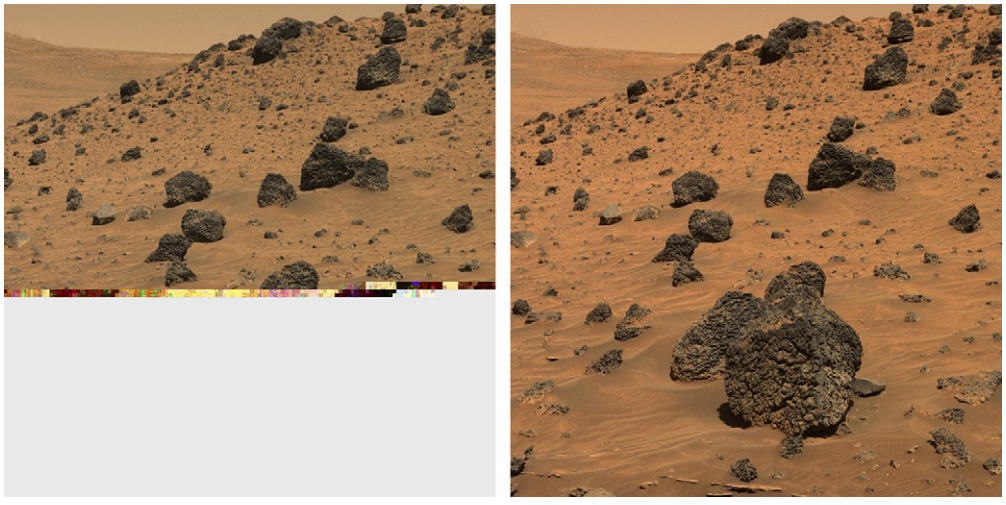

Fig. 1 – These figures show two attempts to carve an image from Mars that was included in the DFRWS 2006 Challenge. The image on the left was formed by supplying a stream of sectors starting at sector 31,533 to a standard JPEG decompressor; the decompressor generated an error when it attempted to decompress sector 31,761. The image on the right was generated by concatenating sectors 31,533–31,752 and 31,888–32,773 into a single file. This example shows that the JPEG decompressor can be used as an object validator, but that it does not necessarily generate an error when it first encounters invalid information. It is thus necessary to augment decompression errors with additional error conditions – for example, the premature end of a file.

Validators may optionally return:

`V_EOF` The validator reached the end of the input string without encountering an error, but no end-of-object flag or other kind of termination symbol was reached. This might be the case with a JPEG image for which the object ends while the JPEG decoder is still decoding the Huffman-coded region.

`object_length` A 64-bit integer which is the number of bytes that object's internal structure implies the file must be. For example, the bytes in a Microsoft Office file can be precisely determined by examining the file's Sector Allocation Table.

4.2.2. *Validator methods*

Validators must implement one method:

- `validation_function()` which takes as an argument a sequence of bytes and returns `V_OK` if the sequence validates, `V_ERR` if it does not, and optionally `V_EOF` if the validator runs out of data before a determination can be made.

Validators may implement additional methods for:

- Sequence(s) of bytes in the file header.
- Sequence(s) of bytes in the file footer.
- A variable that indicates the allocation increment used by file creators. (JPEG files can be allocated in 1-byte increments, while Office files are only allocated in 512-byte increments.)
- An `err_is_prefix` flag that indicates there is no way to turn an invalid object into a valid object by appending additional data. This property is generally true for validators that read an object sequentially from the beginning to the end: these validators can differentiate between the object suddenly containing invalid data (the `V_ERR` condition) and the end of the data stream (the `V_EOF` condition). The JPEG file format has this property, while the MSOLE file format does not.
- An `appended_data_ignored` flag that indicates if data appended to the end of a valid object are ignored. If this flag is not present, then programs that implement the file format ignore additional data that are appended to the end of a file. Most file formats have this property, but formats which place directories or other data at a fixed location from the *end* of the file do not.
- A `no_zblocks` flag that indicates files in this format do not contain sectors filled with ASCII NULs (*zblocks*). JPEG files do not have zblocks, whereas Microsoft Office files frequently do.
- A `plaintext_container` flag that indicates if the file can contain verbatim copies of other files. Microsoft Office files can contain embedded image files, while JPEG files can contain embedded JPEG files as icons.
- A `length_function` which takes as an argument a sequence of bytes and returns a file length if the length of the file can be determined by the byte sequence, or `V_ERR` if the length cannot. Some file formats, such as Office and ZIP, contain characteristic internal structures that can be easily recognized and contain the length of the file.
- An `offset_function` which takes as an argument a sequence of bytes and returns distance that those bytes appear from the beginning of the file, or `V_ERR` if the offset cannot be determined. Some file formats, such as Microsoft Office and ZIP, contain characteristic internal structures that include self-referential structures. From these structures the offset in the file that the structure appears can be readily determined.

We largely implemented three validators with this architecture:

- `v_jpeg`, which checks JPEG segments, then attempts to decompress the JPEG image using a modified libjpeg version 6b.
- `v_msole`, which checks the CDH, MSAT, SAT, and SSAT regions of the Microsoft Object Linking and Embedding format used by Microsoft Office, then attempts to extract the text of the file using the wvWare (Lachowicz and McNamara, 2006) library.
- `v_zip`, which validates the ZIP ECDR and CDR structures, then uses the `unzip -t` command to validate the compressed data.

5. Carving with validation

As discussed in Section 1, *carving* is the general term that we employ for extracting data (files) out of undifferentiated blocks (raw data), like carving a sculpture out of stone.

We have developed a carving framework that allows us to create carvers that implement different algorithms using a common set of primitives. The framework starts with a byte in a given sector and attempts to grow the byte into a contiguous run of bytes, periodically validating the resulting string. Several optimizations are provided:

- The carver maintains a map of sectors that are available for carving and sectors that have been successfully carved or that are allocated to files. As soon as a candidate run extends into a sector that is not available, the run is abandoned and the carver can proceed to the next run.
- If the validator has the `zblock` flag set, the run is abandoned if the carver encounters a block filled with NULs.
- If the validator has the `err_is_prefix` flag set, the run is abandoned when the validator stops returning `V_EOF` and starts returning `V_ERR`.
- If the validator has the `appended_data_ignored` flag set, the run's length can be found by performing a binary search on run lengths, rather than starting with a run that is one block long and gradually extending it.

In this section we present a number of carving algorithms that are enabled by our object validator architecture. The algorithms are divided into two categories: algorithms which will carve a contiguous file from an image, and algorithms that will carve files that are fragmented.

5.1. Contiguous carving algorithms

Our contiguous carver supports *block-based carving*, which only looks for files beginning and ending at sector boundaries, as well as *character-based carving*, which attempts carving on character boundaries. Block-based carving is fast, but character-based carving will find objects that are embedded in various kinds of container files. Character-based carving is also necessary when carving objects that are stored on file system such as ReiserFS (Mason, 2001) that do not restrict new objects to sector boundaries.

In Section 4.1 we described a general strategy for carving contiguous objects from a disk image using object validation. The carver we have implemented can perform a variety of optimizations, depending on the individual properties of the object validators.

5.1.1. Header/footer carving

Header/footer carving is a method for carving files out of raw data using a distinct header (start of file marker) and footer (end of file marker). This algorithm works by finding all strings contained within the disk image with a set of headers and footers and submitting them to the validator.

5.1.2. Header/maximum size carving

This approach submits strings to the validator that begin with each discernible header and continue to the end of the disk image. A binary search is then performed on the strings that validate to find the longest string sequence that still validates. This approach works because many file formats (e.g. JPEG, MP3) do not care if additional data are appended to the end of a valid file.

5.1.3. Header/embedded length carving

Some file formats (MSOLE, ZIP) have distinctive headers that indicate the start of the file, but have no such distinctive flag for the end.

This carver starts by scanning the image file for sectors that can be identified as the start of the file. These sectors are taken as the *seeds* of objects. The seeds are then grown one sector at a time, with each object being passed to the validator, until the validator returns the length of the object or a `V_ERR`, indicating that a complete file does not exist. If an embedded length is found, this information is used to create a test object for validation. Once an object is found with a given start sector, the carver moves to the next sector.

5.1.4. File trimming

"Trimming" is the process of removing content from the end of an object that was not part of the original file. We have found two ways for automating trimming. If there is a well-defined file footer, as is the case with JPEG and ZIP files, the file can be trimmed to the footer. For byte-at-a-time formats that do not have obvious footers, the files can simply be trimmed a character at time until the file no longer validates; the last trimmed byte is the re-appended to the file.

5.2. Fragment Recovery Carving

We use the phrase *Fragment Recovery Carving* to describe any carving method in which two or more fragments are reassembled to form the original file or object. Garfinkel called this approach "split carving" (Garfinkel, 2006a).

5.2.1. Bifragment Gap Carving

If a region of sectors in a disk image begins with a valid header and ends with a valid footer but does not validate, one possibility is that the file was in fact fragmented into two or more pieces and that the header and footer reside in different fragments. In the Garfinkel corpus there are many cases of bifragmented files where the gap between the first fragment and the second is a relatively small number of disk sectors.

The 2006 Challenge contained several instances of JPEG files that were in two fragments, with one or more sectors of junk inserted in the middle. Aside from the large number of fragmented files in the Challenge and the fact that the gap size was rarely an integral power of two, the scenario was quite realistic.

To carve this kind of scenario Garfinkel developed an approach which involves assembling repeated trial objects from two or more sector runs to form candidate objects which are then validated. Here we present an improved algorithm for split carving, which was called *Bifragment Gap Carving* (Fig. 2):

- Let f_1 be the first fragment that extends from sectors s_1 to e_1 and f_2 be the second fragment that extends from sectors s_2 to e_2.
- Let g be the size of the gap between the two fragments, that is, $g = s_2 - (e_1 + 1)$.
- Starting with $g = 1$, try all gap sizes until $g = e_2 - s_1$.
- For every g, try all consistent values of e_1 and s_2.

Essentially, this algorithm places a gap between the start and the end flags, concatenating the sector runs on either side of the gap, and growing the gap until a validating sequence is found. This algorithm is $O(n^2)$ for carving a single object for file formats that have recognizable header and footer; it is $O(n^4)$ for finding all bifragmented objects of a particular type in a target, since every sector must be examined to determine if it is a header or not, and since any header might be paired with any footer.

5.3. Bifragment Carving with constant size and known offset

Bifragmented MSOLE documents cannot be carved with gap carving because there is no recognizable footer. However,

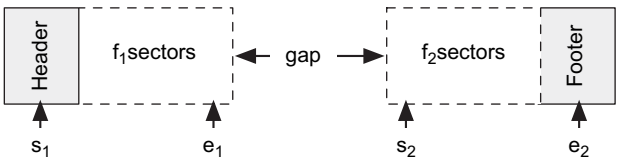

Fig. 2 – In Bifragment Gap Carving the sectors s_1 and e_2 are known; the carver must find e_1 and s_2.

the CDH can be recognized. Recall that the CDH has a pointer that points to the MSAT and the SAT. Because the CDH is the first sector of the file it always appears in the first fragment. If the MSAT occurs in the second fragment (and it frequently does, because the MSAT tends to be written near the end of the MSOLE file), then it is actually possible to find this self-referential sector by examining every sector in the disk image. (There is a small probability that a sector will match by chance, but the probability is quite small.)

We have developed a carver that makes use of this information to find and recover MSOLE files that are fragmented in this fashion. The carver starts with s_1, the address of a CDH, and uses the information in the header to find m_1, the first block of the MSAT. From m_1 the carver can determine L, the length of the final file, as well as the sector offset within the file where m_1 must necessarily appear (Fig. 3).

The carver now employs an algorithm similar to gap carving except that the two independent variables are the number of sectors in the first fragment and the starting sector of the second fragment. The length of the two fragments must sum to L and the second fragment must include sector m_1. This carving algorithm is $O(n^3)$ if the CDH location is known and the MSAT appears in the second fragment, and $O(n^4)$ if the forensic analyst desires to find all bifragmented MSOLE files in the disk image. A variant of this algorithm can be used if the MSAT is in the first fragment and portions of the SAT (which is not contiguous) are in the second fragment. We saw both of these cases in the 2006 Challenge.

If the entire SAT is within the first fragment, the second fragment must be found by validating individual data objects within the Microsoft compound document. This case did not appear in the 2006 Challenge.

Applying this carver to the 2006 Challenge we were able to recover all of the Microsoft Word and Excel files that were split into two pieces. However, in one case we had numerous false positives – files that would open in Microsoft Word but which obviously contained incorrect data. The files opened in Word because our MSOLE validator was able to produce file objects that contained valid CDH, MSAT, SAT and SSAT, but which still had substituted internal sectors. Some of the files opened instantly in Word while others took many tens of seconds to open. However, the number of file positives was low, and we were able to manually eliminate the incorrect ones.

One of the Office files in the Challenge was in three pieces. Using two of these fragments our carver produced a file that could be opened in Word and that contained most but not all of the text. Using this text we were able to locate a source file on the Internet that was similar but not identical to the file in the Carving Challenge. However, enough of the 512-byte sectors were the same that we were able to determine the outlines of the three fragments. We then manually carved these three fragments into a single file, opened it, and verified that it was correct.

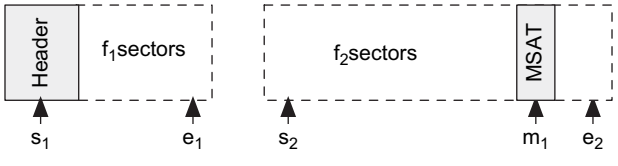

Fig. 3 – In Bifragment Carving with constant size and known offset the sectors s_1 and m_1 and $f_1 + f_2$ are known; the carver must find e_1, s_2 and e_2.

6. Conclusions

Files that are forensically interesting contain significant internal structure that can be used to improve today's file carvers as well as to carve files that are fragmented into more than one piece. Carvers should attempt to handle the carving of fragmented files because these files occur with regularity on file systems recovered from the wild.

6.1. *Future work*

The Sleuth Kit can extract *orphan* files from a file system image. In our survey of the Garfinkel corpus, we have seen many orphans that are separated by number of blocks that is an integral power of two. We suspect that some of these ''orphans'' might in fact be two fragments of a single bifragmented file. We plan to write modify our carver to take into account the output of SleuthKit and see how many of these files can actually be validated.

Elements of the DFRWS 2006 Challenge could only be solved with software that could distinguish English from French text, or which could examine two pieces of English text and determine that they were from different documents. In the future, we hope to integrate semantic carving into our carving system.

Finally, we are developing an intelligent carver that can automatically suppress the sectors that belong to allocated files as well as sectors that match sectors of known good files from the National Software Reference Library (2005).

Acknowledgments

The author would like to thank Brian Carrier, George Dinolt, Chris Eagle, Bas Kloet, Robert-Jan Mora, Joachim Metz, Paula Thomas, and the anonymous reviewers for their useful comments on previous drafts of this paper.

Primary analysis of the Garfinkel corpus was performed at the Center for Research on Computation and Society using equipment provided by Basis Technology. Subsequent analysis of metadata that was derived from the corpus was performed at the Naval Postgraduate School in Monterey, California.

REFERENCES

Carrier Brian. The Sleuth Kit & Autopsy: forensics tools for Linux and other Unixes, <http://www.sleuthkit.org>; 2005a.

Carrier Brian. File system forensic analysis. Pearson Education; March 2005b.

Carrier Brian, Casey Eoghan, Venema Wietse. File carving challenge, <http://www.dfrws.org/2006/challenge>; 2006.

Douceur John R, Bolosky William J. A large-scale study of file-system contents. In: SIGMETRICS '99: Proceedings of the 1999 ACM SIGMETRICS international conference on measurement and modeling of computer systems. New York, NY, USA: ACM Press, ISBN 1-58113-083-X; 1999. p. 59–70.

Garfinkel Simson. DFRWS 2006 challenge report;, <http://www.dfrws.org/2006/challenge/submissions/>; 2006.

Garfinkel Simson. Forensic feature extraction and cross-drive analysis. Digit Investig, <http://www.dfrws.org/2006/proceedings/10-Garfinkel.pdf>; August 2006.

Garfinkel Simson L, Malan David J, Dubec Karl-Alexander, Stevens Christopher C, Pham Cecile. Disk imaging with the advanced forensic format, library and tools. In: Research advances in digital forensics (second annual IFIP WG 11.9 international conference on digital forensics). Springer; January 2006.

Hamilton Eric. JPEG file interchange format v.1.02. Technical report. C-Cube Microsystems; September 1992. <http://www.w3.org/Graphics/JPEG/jfif3.pdf>.

Katz Phil. APPNOTE.TXT – .ZIP file format specification. Technical report. PKWare, Inc.; September 29, 2006. <http://www.pkware.com/documents/casestudies/APPNOTE.TXT>.

Lachowicz Dom, McNamara Caolán. wvWare, library for converting word documents, <http://wvware.sourceforge.net>; 2006.

Mason Chris. Journaling with Reisersfs. Linux J, 2001, <http://portal.acm.org/citation.cfm?id=364719>; 2001. ISSN: 1075-3583 [Article no. 3].

Mikus Nicholas. An analysis of disc carving techniques. Master's thesis. Naval Postgraduate School; March 2005.

National Institute of Standards and Technology. National software reference library reference data set, <http://www.nsrl.nist.gov/>; 2005.

Daniel Rentz. Microsoft compound document file format v1.3, <http://sc.openoffice.org/compdocfileformat.pdf>; December 2006.

Richard III Golden G, Roussev V. Scalpel: a frugal, high performance file carver. In: Proceedings of the 2005 digital forensics research workshop, DFRWS, August 2005. <http://www.digitalforensicssolutions.com/ Scalpel/>.

The carve path zero-storage library and filesystem, <http://ocfa.sourceforge.net/libcarvpath/>.