Towards an Engineering Approach to File Carver Construction

Leon Aronson* and Jeroen van den Bos*†
*Netherlands Forensic Institute, The Hague, The Netherlands
†Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
leon@holmes.nl, jeroen@infuse.org

***Abstract*—File carving is the process of recovering files without the help of (file system) storage metadata. A host of techniques exist to perform file carving, often used in several tools in varying combinations and implementations. This makes it difficult to determine what tool to use in specific investigations or when recovering files in a specific file format.**

We define recoverability as the set of software requirements for a file carver to recover files in a specified file format. This set can then be used to evaluate what tool to use or which technique to implement, based on external factors such as file format to recover, available time, engineering capacity and data set characteristics.

File carving techniques are divided into two groups, format validation and file reconstruction. These groups refer to different parts of a file carver's implementation. Additionally, some techniques may be emphasized or omitted not only because of file format support for them, but based on performance effects that may result from applying them.

We discuss a simplified variant of the GIF image file format as an example and show how a structured analysis of the format leads to design decisions for a file carver.

***Keywords*-digital forensics; software engineering; file carving; data recovery; file formats; performance;**

I. Introduction

Forensic data recovery tools such as *file carvers* [1] use a large amount of different techniques to recognize (parts of) files, ranging from simple recognition approaches such as magic number matching [2] to more advanced and involved approaches such as file structure validation [3] and statistical fragment classification [4] [5].

Research in data recovery techniques typically provides empirical results of effectiveness on different types of files as well as a discussion of the underlying causes. However, it is difficult to properly evaluate the complexity of recovering a file of a newly introduced file format or the expected effectiveness of a proposed recovery technique. Having this capability would make it easier to decide what techniques to use during digital forensics investigations as well as assist developers creating new file formats to make design decisions that improve a file format's *recoverability*.

We propose a definition of recoverability specific to digital forensics based on the software engineering requirements of implementing a file carver for a file format. These requirements consist of techniques that are a reflection of two factors that influence recoverability. First, the difficulty of automatically validating that any given block of data conforms to the given file format. Second, the impact that outside effects, such as fragmentation, have on the difficulty of recovering a file of the given file format.

These requirements can be used in several ways. First, to determine the complexity of (automatically) constructing a file carver for a given file format. Second, to help decide what file formats to search for or which file carver (or algorithm) to use during a digital forensics investigation. Finally, to guide application developers in creating file formats that are relatively easy to recover.

This paper is organized as follows. Section II discusses the techniques that are available as requirements for a file carver for a given file format. As such it also serves as a discussion of related work in the domain of file carving. Section III discusses the impact these techniques have on the performance of file carvers and the recoverability of the file formats they support. Section IV presents an example discussing all the issues presented before and demonstrating their use. Section V discusses the suitability and applicability of the proposed approach. Section VI concludes.

II. File Carving Techniques

If metadata describing where a file is stored (usually as part of a file system) is missing or inaccessible, a content-based approach can be used to recognize and reassemble available data in an attempt to recover files. *File carving* is the term used for all combined approaches in this area. Two factors influence how difficult it is to recover a file using file carving: the file's own format as well as the state of the (surrounding) data. The following subsections discuss these two factors and split them into separate techniques that can be implemented independently.

A. Format Validation

The easiest way to determine whether a block of data conforms to a given format, is to load it into an application accepting that format and, if it loads, manually inspect the loaded file to see if the content makes sense. However, when recovering data this approach is generally unfeasible: from terabytes of data, millions of files can potentially be recovered, which could take months to inspect manually.

Cutting out user intervention increases feasibility considerably, which is achieved by using an automated *format validator*. This is a program (or function in a system) that

accepts a block of data and determines whether it conforms to the defined structure of the file format it validates.

Although this approach is typically orders of magnitude faster than manual format validation, major scalability issues remain. The more strict a validation is, the more computing power it generally requires. For example, validating compressed files may require decompressing all contents and calculating multiple hashes over large amounts of data.

Whether automated format validation is at all feasible also depends on the file format's defined structure. If the structure is only loosely defined and does not have any internal verification mechanisms (such as using length fields or an embedded hash) it may even be impossible to automate format validation.

Several approaches exist to perform automated format validation, all related to aspects of existing file formats. Following is a discussion of those approaches, ordered by increasing complexity.

1) Magic Number Matching: Binary file formats typically use *magic numbers*, identifiers that signal the beginning (or end) of a file or internal data structure. For example, GIF files always start with the ASCII string `"GIF"` and end with the byte `0x3B`. A validator using magic numbers only needs to compare values in order to make decisions. However, except for the bytes containing the magic numbers, not much is known about the data.

Scalpel [2], the successor to ForeMost, is one of the most popular file carvers and nearly exclusively uses this technique.

2) Data Dependency Resolving: The use of *data dependencies* allows file formats to parameterize (parts of) their own layout. For example, BMP files contain length fields that specify both the size of the entire file and of an internal data structure, as well as a flag specifying whether a color table is embedded in the file. Interpreting these values can help validators to locate possible inconsistencies (e.g., when the end of a block described by a length field is not followed by the next expected data structure) but even then, the actual contents of the data blocks are not validated.

3) Internal Verification Checking: As opposed to the previous two approaches, *internal verification* does take the actual contents of (part of) a file into account. For example, PNG files consist of a series of so-called chunks, which are blocks of data that specify their length, type (using magic numbers), contents and a cyclic redundancy code (CRC) over the type and contents. While calculating CRCs takes time, a validated block of data is very likely to be correct.

4) Algorithm Output Analysis: Most file formats that are interesting in a digital forensics context employ some kind of encoding or compression. *Output analysis* examines the encoded or compressed data as stored in the file in relation to the algorithm that was used to create it. For example, JPEG uses Huffman coding to compress data. Given a block of data, it is possible to determine whether it was likely compressed with a given Huffman table using bit sequence matching [6].

5) Compressed/Encrypted Data Decoding: Data decoding as part of validation is basically an automated version of the manual validation process without the inspection. For example, successfully decoding an MPEG file has a high chance of yielding at least a partially viewable movie. Additionally, if the decompression is only partially successful, the location of the error is usually close to where the corrupted or missing data is. However, it does require significant computation, even compared to typical internal verification or even output analysis.

B. File Reconstruction

If all data were stored in single consecutive blocks and never overwritten, data recovery would be nothing more than running all available format validators on a block of data and collecting the resulting files. In practice, operating systems implement a host of performance optimizations that both enable and complicate data recovery.

The biggest performance gain for file systems is typically achieved by not actually removing files upon deletion, but simply marking their location as available for writing. This optimization makes file carving at all possible. File fragmentation is an optimization that causes files to be split into several parts and scattered over the physical contents of a storage device. This complicates data recovery significantly.

Without metadata, it is difficult to determine the original order the fragments were stored in. Attempting all possible combinations of a set of fragments is intractable. *File reconstruction* is concerned with employing heuristics such as knowledge of typical fragmentation patterns or file characteristics in order to reduce the search space. Following is a discussion of the approaches in this area.

1) Fragment Reordering: File reconstruction based on *fragment reordering* attempts a subset of all possible combinations of available fragments and uses a set of format validators to determine matches. There are two general approaches to compute this subset and keep implementations within an acceptable running time.

Bifragment Gap Carving [3] restricts the search space by only carving fragmented files that are split into two fragments that occur consecutively on the physical storage. First, a block of data starting with a header and ending with a footer that is rejected by the format validator is located. All possible combinations of fragments that make up this block are then attempted, under the constraints that no fragments are reordered and that all removed fragments are contiguous. In effect, all embedded "gaps" are attempted.

Advanced Carving [7] uses a format validator not only to accept or reject files, but also to determine the location in rejected files where the fragmentation has occured. As a result, only certain file formats (that have extensively defined internal structure) and format validators (that implement data

dependency resolving or data decoding) can work with this approach. It can recover files where the fragments are out-of-order on the physical storage, but on realistic data sets is only tractable when recovering files split into two fragments.

2) Fragment Classification: An alternative approach to reducing the amount of possible combinations of fragments to consider when reconstructing files is *fragment classification*. Individual fragments are considered and, based on their contents, either included or excluded from further reconstruction. As such this approach combines well with fragment reordering or any other file carving technique, as it simply reduces the amount of fragments to consider.

Classifiers are generally implemented in the form of supervised learning applications using some metric that helps recognize different types of file fragments. Experiments in this area have been conducted using a diverse set of metrics, including byte frequency analysis and byte frequency correlation analysis [8], Shannon entropy, chi-square distribution and Hamming weight [9] and Normalised Compression Distance [10].

Although classification techniques all have their own characteristics, a general observation is that compressed and encrypted data is easy to recognize, but hard to classify. Data that has not been encoded, such as plain text or bitmap files are generally easy to classify, because they have easily identifiable characteristics (e.g., plain text only uses a subset of all byte values and bitmaps often have distinct patterns such as having a zero every four bytes as alpha channel value).

III. File Carving Performance

The file carving techniques discussed in the previous section all enable automated file carving and have some performance benefit or cost for a file carver that employs them. In general, without a limitation on the amount of combinations of fragments considered, no matter how fast a format validator is, the resulting running time on an average hard drive can be months or years. At the same time, format validators may require significant computation to come to a conclusion. These two factors are discussed in the following subsections.

A. Format Validator Invocation Reduction

Recovering fragmented files is a combinatorial problem: all combinations of fragments in the set of the smallest unit of data on a data storage device are to be attempted to discover files that originally resided on the device. An attempt in this context is an invocation of the format validator to determine whether a match was found. This solution is intractable even when the large amounts of data involved in practice are ignored. Two approaches are used to reduce the amount of times the format validator is invoked.

The first is to only consider a subset of all fragment combinations, which is what all practical fragment reordering techniques such as bifragment gapcarving do. The algorithm simply only looks for files that have been fragmented in a certain manner that is common in practice, for example, fragmentation into two parts [3].

The second is to use the results of each format validator invocation or some other program or function to reduce the data set either by eliminating or grouping fragments. Elimination is achieved using techniques such as magic numbers, by excluding all fragments that do not start with some fixed value, and fragment classification, by excluding all fragments that do not match the statistical properties of the file format that is being recovered. Grouping is achieved by all format validation techniques that support partial validation by grouping fragments together once they partially validate (typically the start of a block up to a certain point) in some attempted order. The grouped fragments can then be considered a single (larger) fragment, reducing the size of the data set.

B. Format Validator Computation Reduction

The amount of data on current data storage devices is growing to such a size that reducing the amount of fragment combinations to consider from the original intractable solution to a polynomial-time solution may still require days or weeks of processing time dependent on the performance of the format validator.

For example, when considering a relatively small block of 100MB of data consisting of typical 512-byte sectors, a quadratic function to determine all possible candidates for validation requires billions of format validator invocations. This makes it extremely important for format validators to only perform computations that are crucial to validate files in a given file format.

One possible approach is to not implement computationally expensive validation techniques such as output analysis and data decoding and accept a small amount of false positives, especially given that eventual evidence will have to be manually inspected. If the percentage of false positives is manageable in this manner, it may pay off to accept them and handle them in the manual stage.

IV. Recoverability Example: GIF

Based on the techniques enumerated in Section II and the performance considerations discussed in Section III it is possible to assess what combination of techniques can either be discarded or included in the software engineering requirements for a file carver for a given file format, as well as how to use those techniques effectively. As a practical illustration, we discuss a simplified structure of the GIF image file format.

A description of the structure we discuss is shown in Figure 1. The structure is expressed in DERRIC [11], a digital forensics-specific data description language that we have developed to precisely express the structure of data formats in order to allow extensive analysis.

```
1 format GIF
2
3 strings ascii
4 sign false
5 unit byte
6 size 1
7 type integer
8
9 sequence
10 Header ([Image CompressedBlock* ZeroBlock]
11         [AppExtension DataBlock* ZeroBlock]
12        )*
13 Trailer
14
15 structures
16 Header {
17     Signature: "GIF";
18     Version: "87a" | "89a";
19     LSWidth: size 2;
20     LSHeight: size 2;
21     GCTFlag: unit bit;
22     ColorResolution: unit bit size 3;
23     SortFlag: unit bit;
24     GCTSize: unit bit size 3;
25     BGColorIndex;
26     PixelAspectRatio;
27     GCT: size GCTFlag*(3*(2^(GCTSize+1)));
28 }
29
30 Image {
31     Separator: 0x2c;
32     Left: size 2;
33     Top: size 2;
34     Width: size 2;
35     Height: size 2;
36     LCTFlag: unit bit;
37     InterlaceFlag: unit bit;
38     SortFlag: unit bit;
39     Reserved: unit bit size 2;
40     LCTSize: unit bit size 3;
41     LCT: size LCTFlag*(3*(2^(LCTSize+1)));
42     LZWMinCodeSize;
43 }
44
45 AppExtension {
46     ExtensionIntroducer: 0x21;
47     ExtensionLabel: 0xff;
48     BlockSize: 11;
49     AppIdentifier: type string size 8;
50     AppAuthCode: size 3;
51 }
52
53 DataBlock {
54     Length: 1..255;
55     Data: size Length;
56 }
57
58 CompressedBlock = DataBlock {
59     Data: lzw(packing=lsbfirst,
60               codesize=variable,
61               startsize=Image.LZWMinCodeSize)
62           size Length;
63 }
64
65 ZeroBlock { Length: 0; }
66
67 Trailer { Marker: 0x3b; }
```

Figure 1. Simplified structure of the GIF image file format

A. Simplified GIF Format

A simplified structure of the GIF image file format discussed is shown in Figure 1. The only simplification that has been applied is the exclusion of some extension structures due to size constraints in this paper. As a consequence, a file that adheres to this specification is a well-formed GIF image file, making this example realistic.

The specification identifies the name of the format (line 1) along with a set of defaults: strings use the ASCII character set (line 3) and whenever the specification of binary values is omitted, they are unsigned, single-byte integers (lines 4-7). The rest of the specification is divided between the specification of the file format's sequence (lines 9-13) and structures (lines 15-67).

The terms used in the sequence section refer to defined structures of the same name in the structures section. Additional characters are used to define grammatical aspects of the file format, such as optionality (question mark), repetition (asterisk), alternatives (parentheses) and fixed order subsequences (square brackets).

As a result, the defined sequence prescribes that every GIF file starts with a *Header* and ends with a *Trailer*. In between is an arbitrary amount of any combination of two subsequences: starting with an *Image*, followed by any number of *CompressedBlock*s and terminated by a *ZeroBlock* or starting with an *AppExtension*, followed by any number of *DataBlock*s and terminated by a *ZeroBlock*.

The structures referenced in the sequence are defined in the structures section. Every structure has a name along with a list of its fields. Each field has a name and a specification of its contents. Every part of the specification that is not defined is based on the defaults specified at the top of the description (lines 3-7). For example, the *LCTSize* field in the *Image* structure (line 40) has an unknown value (not specified), but its type is an unsigned integer (not specified, based on defaults) with a size of 3 bits (specified).

B. GIF File Carving

Developing a custom file carver for the simplified GIF image file format requires an analysis of its specification and a definition of which techniques in Section II can be used to maximize the amount of recovered data, without using intractable approaches that will often run for months in practice, as discussed in Section III.

1) GIF Format Validation: Every GIF file starts with a fixed header (lines 17-18) and terminates with a fixed trailer (line 67), enabling the use of *Magic Number Matching* to find complete files.

Data Dependency Resolving presents an interesting addition as the format contains several flags to signal the existence of other data structures (lines 21 and 36) and mandates length fields on all *DataBlock*(-based) structures (lines 53-63), including a prescribed *ZeroBlock* terminator (line 65).

GIF files do not contain mechanisms for *Internal Verification Checking*, but *Algorithm Output Analysis* can be performed, especially given the well-known compression algorithm (LZW) and variable starting size for code tokens (lines 59-61). Apart from analyzing the tokens, *Compressed Data Decoding* can be used to fully validate the compressed data stream.

2) GIF File Reconstruction: Extensive structure in the GIF format based around small length-specified *DataBlock*s and a well-known compression algorithm make it relatively easy to develop a format validator that is capable of fairly precisely pinpointing fragment boundaries when attempting to reconstruct a fragmented file, so *Fragment Reordering* approaches such as bifragment gapcarving can be applied.

Applying *Fragment Classification* however is more difficult. While the compressed data will be relatively easy to recognize, the GIF image file format also allows *AppExtension* structures that may contain data that is not compressed, such as plain text comments or even embedded text that is part of the image. So while classification can be useful to identify possible fragments that may be part of a fragmented GIF file, it is not recommended to discard fragments based on not being classified as compressed.

3) GIF File Carving Performance: GIF files tend to be of limited size because larger (photographic) images are often stored as JPEG or PNG, since they support more colors and better compression. The smaller files typically are, the less they tend to be fragmented. Combined with the opportunities in the file format to create a format validator that can fairly precisely pinpoint fragment boundaries, the amount of format validator invocations will be small compared to other media formats.

However, to maximize the amount of files that can be recovered, the *Algorithm Output Analysis* and *Compressed Data Decoding* may be omitted completely. Instead, format validation can rely on *Data Dependency Resolving* fully. This is possible because each *DataBlock*(-derived) structure must specify its single byte length up front, allowing easy detection of errors at a very high granularity of 256 bytes, half the size of typical 512 byte sectors on storage media. Only checking length fields will significantly reduce the amount of computation a format validator has to perform.

V. Discussion

Given the diversity in file formats, file carving techniques, performance considerations and data storage systems, there are several cases where our approach to documenting the recoverability of files based on their format's enabled file carving techniques raises questions about applicability and suitability. Following is a discussion of the questions that we have currently identified.

A. Outside Factors

There are several factors that impact what the actual contents of files in a given file format is made up of. The first is related to the applications that generate the files. All kinds of design decisions were made by the developers of these applications that impact the recoverability of the files the applications create. For example, whether or not to use the optional restart markers in JPEG files, or whether to split the compressed data of a PNG file into separate IDAT structures. Both would make it much easier for file carvers to recover those files, but since they both rely on implementation aspects related to optional features in the file format, they are difficult to integrate into an objective model.

The second factor deals with actual contents of the files. Bitmap files that contain uncompressed data and that don't use the alpha channel are easy to classify, but it is unclear what that means for the entire format. Another example is file size growth. With photo and video cameras producing larger and larger files, the ratio between metadata and (compressed) contents is constantly changing, which may have an impact on how difficult it is to carve files of that type.

B. Technique Selection

There is no fixed process describing how to proceed after enumerating the types of file carving techniques that can be used to recover files of a given file format. In general, it is important to have a validator that maximizes precision, because more than 90% of files are not fragmented [3] and can be recovered without requiring any type of file reconstruction. Beyond that it is difficult to decide on what technique to implement first, especially since the specific encoding or compression algorithms of the different file formats greatly impact the difficulty of implementing the more advanced format validation techniques, such as *Algorithm Output Analysis*.

Still, a structured assessment of the file carving techniques enabled by features of specific file formats does lead to insights about how or if to apply them. An example is in our discussion of the GIF image file format. Without careful analysis, it may appear obvious that such a compressed format will require *Compressed Data Decoding* or *Fragment Classification*, while in practice, the extensive use of a small length field makes faster approaches also feasible.

C. Engineering Effort

Digital forensics investigations are often performed under high time pressure due to deadlines related to legal proceedings such as pre-charge detainment. As a result, apart from precision and performance, another factor is present when dealing with files of a file format that was previously unsupported: engineering effort. When deciding what technique to implement, these three factors must be taken into account. For example, if a file format's embedded data supports high-speed decompression and high precision with regard to locating corrupted data, implementing *Compressed*

Data Decoding for this file format will probably result in fast file carving. However, if implementing support for this feature takes up a large amount of time, it might be more efficient to implement slower and less precise techniques in the validator so that the file carving may start earlier.

D. Format Engineering Implications

Using file carving techniques to develop easily recoverable file formats may result in some unusual design decisions. For example, while it is generally considered good practice to implement existing standards instead of inventing or modifying a new compression algorithm, custom data formats tend to make file carving easier. For example, the escaping in JPEG makes *Algorithm Output Analysis* possible. However, we believe it is a useful tool to assess the recoverability of files during file format development, since most of the practices it promotes are in line with general engineering guidelines.

Following the techniques described in this paper, a file format developer would be advised to use:

- Magic numbers for at least header and trailer.
- Length fields for all data structures.
- Flags to indicate the existence of all optional data structures.
- Checksums to protect the integrity of all data structures.
- Encoded data only if the application requires it.
- Small data structures that will require minimal fragment reordering to recover.

VI. Conclusion

File carving has been the subject of active research for the past decade and has resulted in two techniques that we have discussed in this paper: *Format Validation* and *File Reconstruction*. The first focuses on validating that a block of data adheres to a given file format and file reconstruction focuses on reassembling fragmented files. In practice both techniques are combined so that file carving can be almost entirely automated.

However, automation can easily result in a solution that will still be unfeasible due to the large amount of time required to carve a single file. An important conclusion is that a file carver that simply implements all file carving techniques that a file format supports may not be an optimal solution in practice.

In this paper we have proposed to take all aspects of a file's format into account and consider each technique within the context of accuracy and performance. This will lead to design decisions that are both precise with regard to reducing false positives as well as scalable with regard to recovering data in an acceptable running time.

To illustrate our approach, we have presented an analysis of a simplified GIF image file format and show that considering each technique and combination can lead to different design decisions.

Future work

We are currently developing an automated file carving framework based on model-driven engineering [11], including methods to automatically infer the file carving techniques supported by a given file format in order to generate components for the framework. This would for example enable a user to enter a time limit, which is then used by the application to select a suitable set of carving techniques, thus optimizing the results in a given time frame.

References

[1] A. Pal and N. Memon, "The Evolution of File Carving," *Signal Processing Magazine, IEEE*, vol. 26, no. 2, pp. 59–71, 2009.

[2] G. G. Richard, III and V. Roussev, "Scalpel: A Frugal, High Performance File Carver," in *Refereed Proceedings of the 5th Annual Digital Forensic Research Workshop (DFRWS'05)*, 2005.

[3] S. L. Garfinkel, "Carving Contiguous and Fragmented Files with Fast Object Validation," *Digital Investigation*, vol. 4, no. S1, pp. 2–12, 2007, Proceedings of the Seventh Annual DFRWS Conference.

[4] M. Karresand and N. Shahmehri, "File Type Identification of Data Fragments by Their Binary Structure," in *IEEE Information Assurance Workshop*, 2006, pp. 140–147.

[5] C. J. Veenman, "Statistical Disk Cluster Classification for File Carving," in *Proceedings of the Third International Symposium on Information Assurance and Security (IAS'07)*, 2007, pp. 393–398.

[6] H. T. Sencar and N. Memon, "Identification and recovery of JPEG files with missing fragments," *Digital Investigation*, vol. 6, no. S1, pp. 88–98, 2009, Proceedings of the Ninth Annual DFRWS Conference.

[7] M. I. Cohen, "Advanced Carving Techniques," *Digital Investigation*, vol. 4, no. 3-4, pp. 119–128, 2007.

[8] M. McDaniel and M. H. Heydari, "Content Based File Type Detection Algorithms," in *Hawaii International Conference on System Sciences*, 2003, p. 332a.

[9] G. Conti, S. Bratus, A. Shubina, B. Sangster, R. Ragsdale, M. Supan, A. Lichtenberg, and R. Perez-Alemany, "Automated mapping of large binary objects using primitive fragment type classification," *Digital Investigation*, vol. 7, no. S1, pp. 3–12, 2010, Proceedings of the Tenth Annual DFRWS Conference.

[10] S. Axelsson, "The Normalised Compression Distance as a file fragment classifier," *Digital Investigation*, vol. 7, no. S1, pp. 24–31, 2010, Proceedings of the Tenth Annual DFRWS Conference.

[11] J. van den Bos and T. van der Storm, "Bringing Domain-Specific Languages to Digital Forensics," in *Proceedings of the 33rd ACM/IEEE International Conference on Software Engineering (ICSE'11)*, vol. 2. ACM, 2011.