File Fragment Classification—The Case for Specialized Approaches

Vassil Roussev
Department of Computer Science
University of New Orleans
vassil@cs.uno.edu

Simson L. Garfinkel
Department of Computer Science
Naval Postgraduate School
slgarfin@nps.edu

Abstract

Increasingly advances in file carving, memory analysis and network forensics requires the ability to identify the underlying type of a file given only a file fragment. Work to date on this problem has relied on identification of specific byte sequences in file headers and footers, and the use of statistical analysis and machine learning algorithms taken from the middle of the file. We argue that these approaches are fundamentally flawed because they fail to consider the inherent internal structure in widely used file types such as PDF, DOC, and ZIP. We support our argument with a bottom-up examination of some popular formats and an analysis of TK PDF files. Based on our analysis, we argue that specialized methods targeted to each specific file type will be necessary to make progress in this area.

1. Introduction

Most forensic practitioners can look at a piece of data, such as a disk block or a network packet, and readily identify what kind of data it carries. This skill is important in many forensic tasks, from diagnosing break-ins, making sense residual data, decoding memory dumps, reverse-engineering malware, or simply trying to recover data from a crashed drive.

As investigations become increasingly complex, tools need the ability to perform *file fragment classification* automatically—that is, to be able to infer the type of the file from which the fragment was taken. For example, file carvers that perform fragment reassembly need to be able to classify file fragments that they find on the drive, otherwise they suffer from combinatorial explosion.

File fragment classification is complicated because there are many different kinds of file types—from simple *primitive* types like a block of ASCII text or a JPEG file, to complex container files such an Adobe Acrobat File (pdf), to archive files such as TAR and ZIP that can themselves contain many other files (and even other archives). The problem is also complicated by the fact that the phrase "file type" is itself not well defined.

This paper examines the file fragment classification problem. Section 2 presents a survey of prior work. Section 3 restates the problem in a form that we believe to be more productive. Section 4 explores the opportunity for specialized file fragment classification approaches. Section 5 concludes.

2. Generic Approaches to Fragment Classification

Probably the most commonly used file identification program today is the Unix **file** command and the "libmagic" library on which it relies [4]. This system works by comparing specific regions of a file—typically the file's beginning and head—with a database and reporting the first match. The **file** command is reasonable accurate when given an entire file, but it typically reports "ASCII text" or "data" when given file fragments.

Most work on file fragment identification to date has attempted to solve the classification problem using a combination of machine learning techniques and statistical analysis. Researchers typically assemble a corpus of files of different types. The corpus is divided into two groups, a "training set" and a "test set." The files in the training set are processed with some sort of statistical technique and the results are fed into a traditional machine learning algorithm. The results are used to create a classifier. The test set is then fed into the classifier and its ability to classify is measured and finally reported.

2.1. Byte frequency distribution (BFD) approaches

McDaniel [1] appears to have been the first to consider the identification of file fragments based on techniques other than header/footer analysis. His basic approach was to create for each file a histogram of the frequency of ASCII code (0..255) in the file to be classified. This histogram was turned into a 256-element vector; the vectors representing each file type were then clustered. McDaniel's corpus consisted of 120 files from 30 different file types; only whole files were considered.

BFD-style analysis yielded rather unconvincing results: 27.50% true positive rate for the *BFA* algorithm and 46% for the *BFC* algorithm. McDaniel then proposed an alternative approach that created a file type fingerprint based on a correlation of byte positions and the ASCII value at that position. This approach achieved a respectable 96% success rate—but careful analysis shows that it was simply a variation on the traditional header/footer analysis with the exception that the headers and footers were automatically learned. This approach would not work for file fragment identification.

The next important piece of work belongs to Li [7] who substantially revamped the BFD approach (referred to as 1-gram analysis). The basic idea is to use a centroid, or multiple centroids, derived from the byte frequency distribution as the signature of a file type. But Li's published evaluation did not evaluate the approach on fragments drawn from the middle of a file. Instead, the fragments all started at the beginning of the file with evaluation points at 20, 200, 500, 1000 bytes, as well as the whole file. Interestingly enough, the 20-byte fragments were identified with near perfection, yet the accuracy of the same approach applied to entire files drops significantly—down to 77% for whole *jpeg* files. This is obviously a paradoxical result—using more data, Li got less accurate results. Since no confusion matrix was published, the results are difficult to explain.

Karresand [6] developed a very similar centroid idea to Li's and called it the *Oscar* method. This was shortly extended [5] with the introduction of a new metric called *rate-of-change* (RoC), which was defined as the *difference* of the ASCII values of consecutive bytes. This was done squarely to improve the accuracy of *jpeg* recognition, which becomes near perfect. That is not a surprise, as the *jpeg* format uses the 0xFF as an escape character for all metadata tags. To avoid ambiguity and to simplify processing, the encoder stuffs an extra 0x00 after every 0xFF byte in the body of the file. This produces a very regular, unique, and easily exploitable pattern—0xFF00—which has a very high RoC. Unfortunately, RoC does nothing to improve the rather modest classification success of other file formats considered. For Windows executables, the false positive rate actually *exceeded* the detection rate for most points shown ([5] Fig 3), although the peak detection rate of 70% is equal to a false positive rate of 70%. For *zip* files, things look a little better with false positive rate of 70% when the detection rate reaches 100% ([5] Fig 4).

2.2. Metrics-based approaches

Erbacher argued that one *could* differentiate among the formats by taking a purely statistical perspective of the file container by using standard statistical measurement—averages, distributions, and higher momentum statistical measurements [13]. The argument

was made mostly by plotting the behavior of these measurements over several specific files, which makes the differences apparent to a skilled observer. There was no actual method described as to how exactly these observations fit into a workable classification system.

Follow-up work by Moody and Erbacher [10] attempted to flesh out such methodology based on the observations. The researchers discovered that a metrics-based approach can help distinguish some broad classes of file data, such as textual, executables, and compressed, but becomes easily confused in trying to pick out more subtle differences—*csv* vs. *html* vs. *txt*. For those, they used secondary analysis to make a better distinction with mixed results.

Veenman [16] combined the BFD with Shannon entropy and Kolmogorov complexity measures as the basis for his classification approach. To his credit, he used a sizeable (450MB) evaluation corpus, employed 11 different file formats, and his formulation of the problem—identify the container format for a 4,096 byte fragment—was the closest to our view of how the analysis should be performed. The classification success for most was quite modest: between 18% for *zip* and 78% for executables. The only standouts were *html* with 99% and *jpg* with 98% recognition rates.

Calhoun [3] expanded upon Veenman's work by employing a set of additional measures (16 total) and combinations of them. While the test sets were rather small—50 fragment per file type—he recognized the need for more subtle testing and performed all-pairs comparison of three compressed formats: *jpg, gif,* and *pdf*. Another positive in this work is that header data was not considered so the results are not skewed by the presence of metadata.

The improved evaluation methodology (with the notable exception of sample size) provides one of the first realistic evaluations of generic metrics-based approaches. Provided true positive rates for the binary classification are between 60% and 86% for the different metrics, implying false positives in the 14 to 40% range.

2.3. Discussion

Researchers are trained to pursue generalized solutions whenever possible. Researchers are also taught that "reinventing the wheel" is generally a mistake. Thus, it is quite understandable that researchers exploring the file identification problem would seek out a generalized solution that would work on any file type by borrowing classification methodologies that have worked with spectacular success in other areas.

Before applying a technique from one field to another, it is useful to carefully examine the techniques' underlying assumptions to make sure that they will still apply. The data networking community provides us with a cautionary tale. Before Leland's 1993 seminal paper [17] on the self-similarity of network traffic, practically all research in that area assumed that network traffic followed a Poisson distribution. This assumption led to beautiful models that were analytically very tractable, and were a natural extension of prior work on phone networks. The only problem was that the underlying assumptions were not valid; more than a decade of research had to be written off as useless in the real world.

The prevalent research approach to the file fragment classification problem to date has been based on statistical and machine learning methods. Purely statistical approaches rely on experimentally established thresholds to make decisions: for example, if the fraction of printable characters is above a certain value, it is likely that this is a text file. Machine learning techniques try to be a little more generic by incorporating a number of measurements and letting the algorithm determine the weights through training. Both approaches are evaluated by supplying known data that was not part of the training set to calculate basic statistics like true and false positives.

Clearly, both approaches are critically dependent on the composition and size of the sets used for evaluation (and, in the case of ML approach, training). If these sets are not

representative of what an investigator is encountering, then the results are not predictive. Because the authors of each paper we reviewed use different corpora for their evaluation—sometimes with sizes differing by 2 orders of magnitude—it makes no sense to compare these published results in an attempt to determine which approach is "better."

More importantly, for any generic technique to work, the encoded data must exhibit discernable patterns. Unlike other ML application areas such as biology, physics, and chemistry, where physical phenomena leave fingerprints in the censor data, files are synthetic objects: there are no guarantees that *any* reliable patterns will 'naturally' emerge unless the files have been specifically designed that way. In fact, two popular transformations—compression and encryption—result in the removal of any persistent patterns in the coded stream. In both of these cases, any patterns present in the stream indicate some kind of process failure: a compressed file with patterns could be compressed further, while an encrypted file with patterns is subject to cryptanalysis. Thus it is highly unlikely that a naïve ML approach could identify any type of file or distinguish one from the other.

3. Reconsidering File Fragment Classification

All of the papers presented in Section 2 assumed that each file or file fragment actually had a specific *type*—some kind of ground truth that could be used for training or evaluation. In our reading of these papers the term *file type* was used synonymously with the phrase *file extension*. Before going further, it is worth stepping back and examining what precisely what files are and whether or not they have types.

3.1. What is a 'file type'?

For this work, we consider a *file* to be a sequence of bytes that are stored by a file system under a user-specified name. Generally the file system views files as opaque objects that are created, interpreted and modified by applications on behalf of end users.

Operating systems generally avoid interpreting the names and contents of files, although there are exceptions. Many determine how an executable file will be executed based on the file's name and the first few bytes that the file contains. CP/M and MS-DOS will load any file with extension .COM into memory and execute it if the file's name is typed. Windows marks the beginning of executables with an "MZ". Unix marks the beginning of a.out and ELF files with a different header; if the first two bytes of a file are "#!" then the file is treated as a set of commands to be fed into the program whose name follows the exclamation mark.

Application programs are a different matter. By the 1980s most microcomputers were being purchased by businesses and home users to run specific applications—typically word processors and spread sheets. These programs invariably stored their data in proprietary binary files; sometimes these files could be read by competing programs with varying amounts of fidelity, other times they could not.

As graphical user interfaces became the norm, it became very useful to associate a file types with the corresponding applications so that basic actions like 'open' and 'print' could be initiated by the user from the graphical shell, rather than from within a specific application. DOS and Windows shells created this association through the file's extension. This loose association between file naming conventions and application actions is what the operating system presents to end-users as a 'file type.'

Now, consider the binary application-specific files from the application's point of view. The sole purpose of the byte streams is to store application state across user sessions. To achieve this, the application developer needed some way to serialize and then deserialize the application's state. Frequently, files were simply a dump of the application's memory, although later versions of an application needed the ability to read files created by earlier

versions. To distinguish one version from another, some programmers started data files with a header containing a flag and perhaps a version number.

Logically, the serialized data can be split into two parts—actual data, and structural metadata. The amount of metadata can vary significantly across file formats—text files may have little or none, whereas MS Office documents (such as *doc* and *xls*) are, in fact, small file systems [7]. Over time, we have developed a good number of standardized data representations and those are being shared in significant ways across file formats. This is particularly true of archive and compression formats, as they are rather expensive to code and debug from scratch, and are surrounded by numerous patent and copyright restrictions.

3.2. What is file fragment classification?

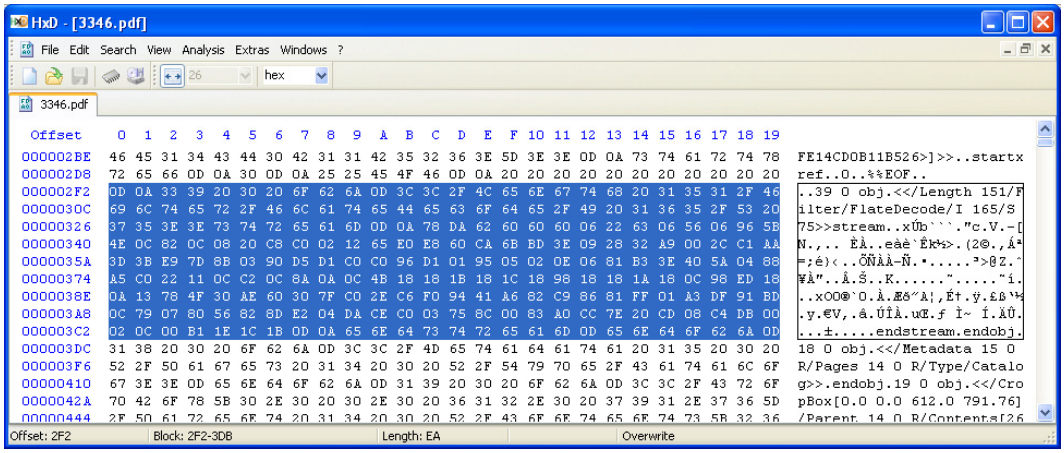

Figure 1 Partial view of a *pdf* file in a hex editor

We are now able to present our definition of *file fragment classification*. Given a contiguous piece of data (a *fragment*), the *classification* is the *encoding format* of the file from which the fragment was taken. This is different from the problem from the *file fragment identification*, in which we try to find the *specific* file from which the fragment was extracted. File fragment *classification* and *identification* are related problems, but solutions to these two problems tend to be quite different.

3.3. What is a pdf file?

To keep the discussion grounded, we will consider one of the prevalent file formats—*pdf*—and try to identify what distinguishes it from other formats. For this purpose we did a survey of over 131,000 valid *pdf* files, obtained from a search engine using random keywords. The overall size of the data set is 85 GB.

At a high level, the structure of a *pdf* file is relatively simple—the bulk of the data consists of a number of autonomous objects encoding text and images in various formats. Those are held together by font, layout, formatting, and other information. Figure 1 shows a snippet from a *pdf* file in a hex editor as an illustration. The highlighted data encodes a compressed block of text of 151 bytes using the standard *deflate* algorithm and *zlib* storage format (RFC 1950[11] /1951[12]).

Even at first glance, it is clear that *pdf* metadata that encapsulates the compressed data blocks is ASCII-encoded and can be readily parsed and identified. Taking advantage of this feature, we were able to produce a breakdown of various types of data objects and their volume relative to overall file size. Table 1 shows the results.

Table 1 Structural composition of *pdf* files

Encoding	Count	Avg Size(KB)	Total (MB)	%
Deflate	10,406,780	4.11	41,730	49.1%
Image (jpeg/jpeg2000)	853,321	25.88	21,570	25.4%
BW Image (fax/JBIG2)	756,532	12.82	9,470	11.2%
PDF-Characteristic			***8,236***	***9.7%***
Application/XML/Form	520,220	3.18	1,614	1.9%
ASCII85/ASCIIHex	205,421	4.51	905	1.1%
LZW/RunLength	64,911	7.70	488	0.6%
Fonts	10,005	1.14	11	0.0%
Other	412,570	2.23	899	1.1%
Grand Total	**13,229,760**	**6.57**	**84,921**	**100.0%**

The first column identifies the encoding method for the object; the rest identify the total number of embedded objects of the particular category, average size, absolute total size, and the relative fraction of the total amount of data, respectively. The main point here is that ***all*** the encoded objects with the exception of the ones labeled *PDF-identifying* can be encountered as part of non-PDF objects. In other words,

Deflate encoding represents compressed text using RFC 1950/1951; *image* encoding includes JPEG and JPEG2000 objects; *BW image* includes CCITFax-encoded and JBIG2-encoded bi-tonal images; *application/xml/form* represents data included as application tags; ASCII85 is a text transcoding scheme, which is designed to communicate binary data using only human-readable characters and typically used in combination with compression encodings; *LZW* [18] is a dictionary compression scheme, best known for its use by the Unix *compress* utility; *fonts* represents embedded fonts. The *PDF-characteristic* data is simply the leftover after we've accounted for all the possible encodings that are shared with other formats.

Returning to our problem of identifying the encoding format of a fragment, consider the case of classifying an excerpt of 512 to 4,096 bytes (which covers the most interesting cases). We can immediately notice that, most of time, we have *no* way of knowing that the excerpt is part of a *pdf* file. Sometimes the fragment will be indistinguishable from a standalone JPEG file; other times it will be indistinguishable from any number of standalone files using the deflate method—including *png*, *zip*, and *swf*, to name a few; in fact, every single encoding on the list is shared with other file formats and applications. Consequently, we can only classify a fragment as part of a *pdf* if it contains some metadata that is characteristic of PDF files—the character sequence "`obj<</`", for example.

We have focused here on PDF files, but the same is true of Microsoft Office files. As mentioned above, Microsoft's older *doc*, *xls* and *ppt* files resemble small file systems with embedded objects represented as streams. Microsoft's newer *docx, xlsx* and *pptx* files are actually *zip* archives where the embedded components are literally stand-alone primitive files. Typically these files are not compressed [14].

Clearly, characteristic metadata is required to identify a fragment from *any* compound file format. Otherwise technique can only possibly identify the embedded object.

3.4. Asking the right questions

From the discussion so far, it should be quite clear that the general problem of file fragment classification would be better formulated as two separate questions:

- What is the primitive data format of the fragment?
- Is the fragment part of a compound file structure?

Combining the two questions, as others have done, leads to an ill-defined problem for all compound document formats. To illustrate, assume that method *A* can correctly classify *pdf* fragments of 1,024 bytes in size 80% of the time and misclassifies them as *jpeg* pieces 15% of the time. Is this a good method to adopt, or not? The correct answer is that we simply do not know. On the positive side, maybe the method only gets confused when the fragment is entirely *jpeg*-encoded so the 15% confusion is just bad reporting on part of the author and should be added to the 80% true positives. On the other hand, if the sample evaluation set were heavily text-based with many *zlib*-encoded parts, this would be a rather poor method as it is relatively easy to separate *jpeg*- and *zlib*-coded streams. Finally, how will the method perform if we encounter a group of *fax*-encoded scanned documents? We have no basis to expect any particular performance for any of these cases because the experimental methodology was not designed to address them.

Restating the fragment classification problem as two separate questions points to both the correct way to approach it and to a much more rigorous and informative evaluation framework. Given a fragment, a classification method should be able to detect evidence of known primitive encoding formats, such as *jpeg*, as well as evidence of compound format metadata (e.g. *pdf*). If it is determined that the fragment contains only bytes associated with a primitive type, then the method should classify the fragment accordingly and state that it has found no evidence that the primitive type is inside a container. Conversely, if the fragment contains evidence of a file container or a compound document, that should be stated.

It is also useful to score methods on two scales—the ability to identify primitive types, and ability to identify compound formats. For primitive types we can measure the size of a fragment vs. correct classification. Separately, we can measure the effectiveness of classification with and without header information. For compound formats, any method would have to rely on structural metadata outside the primitive objects: otherwise the method becomes less accurate as the embedded objects grow in size! Instead, it would be better to compare the amount of metadata contained in the fragment vs. its ability to identify the compound format—a method requiring less metadata would obviously be superior, as it will classify more fragments correctly.

Classification methods should also be able to state 'I don't know' when the data exceeds their abilities. From a practical perspective, having a method that has 60% true positive rate, 0% false positive, and 40% unclassified is much more useful than a method that has 80% true positive and 20% false positive rate. This is especially true for large cases—with the former we can conclusively deal with 60% of the objects and try alternative approaches on the remainder, whereas with the latter method, we are running a gamble so we cannot reliably eliminate anything from consideration.

4. Specialized Approaches to Fragment Classification

We define a specialized method for fragment classification as one designed to correctly identify whether or not the fragment belongs to a *particular* encoding format as opposed to all other formats. Such classifiers rely on the most specific information available about the format and typically employ minimal raw data parsing and focused statistics to achieve their goal. The more general problem of classifying the fragment is solved by cross-correlating the results from the binary classifiers. Such results may be in conflict, in which case the method should flag the discrepancy or refuse to classify the fragment.

4.1. Primary goals

We advocate a practical, minimalistic, and conservative approach that strives to be quick and efficient and openly informs the user of its own limitations:

- *Accuracy*. As with any classification scheme, we are interested in the best possible detection rate. It is important to recognize that with terabyte-scale targets, we need very high classification rates—upwards of 99%—to make a difference.
- *Reliable error rate estimates*. Every classifier should have well-studied error rates based on large and representative studies, preferably on a standardized, public data set.
- *Clear results*. Methods should strive to have *no* false positive/false negative rates. It is preferable for a method to return 'unable to classify' then to risk false results. This is a practical consideration as it allows an investigation to quickly eliminate data from consideration—uncertainty is not helpful in this context.
- *Line-speed performance*. Methods should be fast enough to keep up with bulk I/O transfer from secondary storage. This simple requirement allows us to easily scale up the classification in response to continuous growth of parallelism on commodity hardware.

4.2. Compressed data case study: zlib

To illustrate the difficulties in dealing with compressed data objects, we present a brief study of the *zlib* data format, which is employed by popular formats such as *pdf, zip, cab, gz, bz2, zip, png, jar,* and *swf*. This is an example of a format that entirely focuses on storage efficiency and contains the minimal of metadata necessary for decoding. The *zlib/deflate* bitstream consist of a sequence of compressed blocks. Each of the blocks consists of:

- 3-bit header—the first bit indicates whether this is the last block in the sequence; the following two bits define how the data is coded: raw (uncompressed), static Huffman, or dynamic Huffman.
- If the data is compressed with static Huffman, the decoder uses a default coding table; if dynamic is chosen, the coding table immediately follows (in compressed form).
- The table is followed by a bitstream of variable-length Huffman codes that represent the content of the block. One of the codes is reserved for marking the end of the block.

Note that there is no break in the bitstream between blocks—as soon as the end-of-block code is read from the stream, the next *bit* is the beginning of the following block header. Note also that the actual end-of-block code depends on the coding table for the block and is not fixed. In other words there is no *synchronization* information of any kind—if a decoder were to miss a single bit, there is no opportunity to recover from that error. Worse yet, due to the properties of variable-length codes, the decoder will likely continue to work, producing garbage output potentially through the end of the bitstream.

We already know that statistical variation of the coded stream is quite uniformly random [2]. The lack of synchronization information is very bad news from a forensic perspective. To illustrate the point, we conducted a study of 1,317 *gz* files (414 MB) downloaded from a search engine using random keywords. We instrumented a version of *pyflate* [15] to empirically answer some basic questions of forensic interest about *zlib* streams.

- *How likely is that a compressed block will use the default Huffman table?* If the static Huffman option is the norm, then we could try to decode streams using the default table and that could give us an opening for further analysis.

Based on our sample, that probability is in the order of 0.1% as only 14 out of 13,711 blocks were statically coded. 99.5% of all the blocks (13,638 out of 13,711) in the sample were coded using dynamic Huffman tables, and 0.4% of the blocks were uncompressed. The

coding tables also turn out to be very different from each other with 99.8% of them being unique. In short, we have virtually no hope of guessing the correct Huffman decoder.

- *How likely is that the incorrect decoder will decode a zlib fragment?* If incorrect decoders were to fail quickly, that could help us classify the source of a fragment.

To answer this question, from each file we picked one decoder and tried to decode 4,096 bytes from each of the other files, starting at a random position (almost all Huffman codes are self-synchronizing [2] so the actual starting point makes little difference). We were successful on all but one occasion. In fact, we fed the decoders random data blocks and none of them crashed. We also examined how quickly different decoders synchronize and saw no statistical deviations that would allows us to pick the correct one.

In summary, it appears to be very difficult to distinguish a compressed data object, which contains no synchronization information, from random data, or from encrypted data. By extension, machine learning or statistical methods that have been applied to file fragment classification to date simply have no discernable patterns to exploit.

4.3. Compressed data case study: jpeg

As mentioned, the *jpeg* format has some nice features as far as fragment classification is concerned. We split the concerns of detecting the jpeg header from the detection of the encoded image. The header has a simple record structure where the beginning of each record is announced by the presence of a marker—a 16-bit number in the 0xFFC0 to 0xFFFE range, which is followed by a 16-bit number describing the length of the record. The exact meaning of the records is irrelevant here—all we need to do is skip from record to record until we find the 0xFFDA tag ('start of scan'), which marks the beginning of the encoded image.

The minimum valid beginning (magic number) of a valid jpeg is thus 0xFFD8FF. What we wanted to test is—what is the probability that such a header-scanning approach would wrongfully identify a piece of data as a jpeg image header. We ran the test against several randomly obtained sets: 4,000 *pdf* files (2,150MB), 4,000 *doc* files (955MB), 1,300 *xls* files (307MB), and 1,300 gz files (414MB). In each case, after a valid header was discovered, we simply carved out all the data from the beginning until the 0xFFD9 marker and asked the operating system to show thumbnail images as proof that valid images were obtained. We found *no* false positives at all.

In our analysis we found that the average distance between two occurrences of 0xFF00 in a jpeg file was 191 bytes. This is considerably more frequent than it would be under uniformly random distribution but so frequent that it's presence would be exploited by typical machine learning techniques.

Although we have termed jpeg a "primitive" file type in this paper, it is actually a container file format. Certain products, such as Adobe Photoshop, have a propensity of using jpeg's allowance for 'application segments' rather heavily. Using this feature, one could embed *any* kind of data, which could create problems during data recovery. We found, for example, in a small 20KB image another version of the image in the header, which appeared to be of *higher* quality. There are usually non-trivial amounts of metadata as well.

The next problem is to identify the jpeg body of the image. It is fairly straightforward to identify compressed/encrypted data using some basic entropy measurements. The true task is to differentiate among different compressed streams. Apart from *zlib*, most compressed formats *do* have some synchronization information—this is almost universally true for multimedia formats as they need to support fast seek operations.

For this experiment, we focus on differentiating between *jpeg* and *zlib*. We mentioned earlier that, in the body of the image, jpeg encoders stuff a 0x00 byte after every 0xFF. In addition to that, there are a few more legal markers that may appear—mostly in the 0xD0 to

0xDB range. In this experiment, we ask the question: given a fragment of size 512/1,500/4,096 bytes, what is the probability that a non-jpeg fragment will be indistinguishable from a jpeg one based on the above rule? In other words, we assume that the fragment is jpeg and we are looking for evidence to disprove the hypothesis. To do so, all we need is one instance of an illegal two-byte sequence that starts with 0xFF.

Figure 2 shows the empirical cumulative distribution function of the distance between two consecutive instances as determined by our *gz* test set. For our specific points of interest, we find out that the probability that the distance between two instances will *not* exceed 512/1,500/4,096 bytes and, therefore, will provide us with definitive evidence is 83.41%, 98.5%, and 99.97%, respectively.

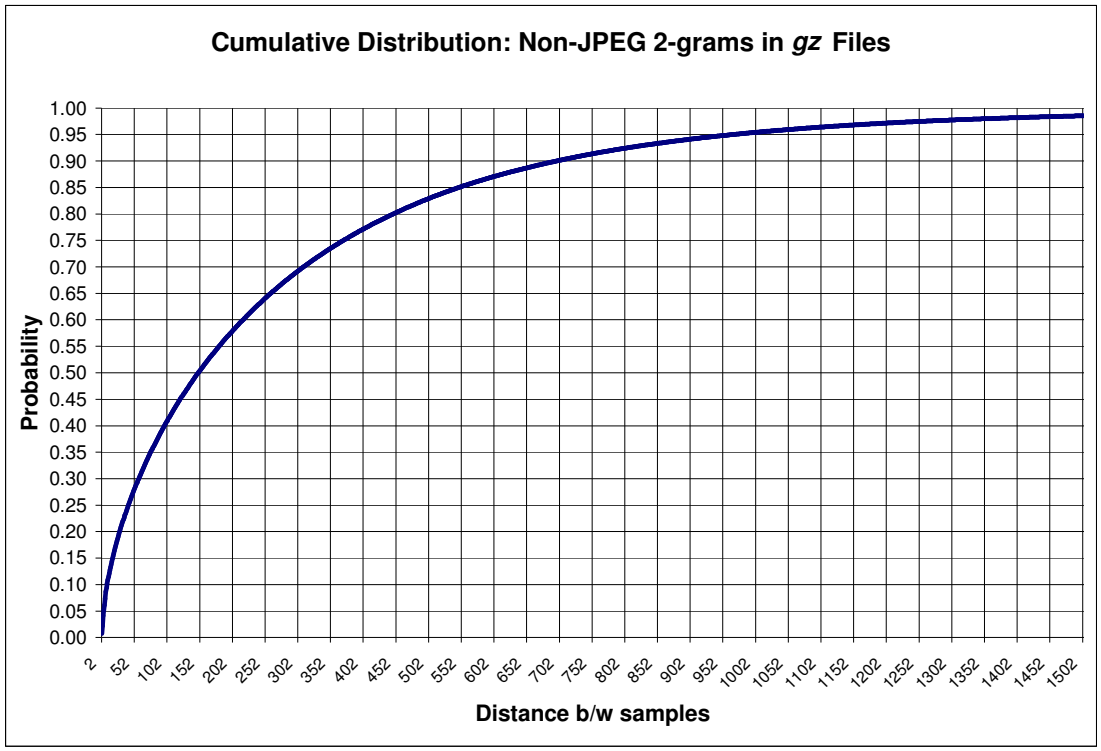

Figure 2 Empirical CDF of the distance between two non-JPEG instances: *gz* files

4.4. Compressed data case study: mp3

mp3 is another format that is classification-friendly and is representative of a broader class of multimedia containers. The main insights come, again, from the ultimate source—the format specification. An *mp3* file consists of a sequence of *mp3 frames* that are completely self-contained and do not depend in any way on the ones before, or after. The beginning of the frame is marked by 12 consecutive 1 bits that are certain not to appear anywhere else in the encoded stream. Following the synchronization bits is information about the version, bit rate, sample rate, and padding. Based on these parameters we can calculate the length of the frame and predict the appearance of the next header.

There are several criteria that allows us affirm, or reject the fragment as *mp3-encoded*. First, the predictable appearance of the frame header—for a fixed-rate encoding this will be almost periodic (for the typical 44.1KHz sampling rate a padding byte is necessary every once in a while). Next, not all combinations of parameters in the frame header are legal, and even fewer are actually seen in the wild. Consecutive frames may be encoded at different bit rate but little else changes from frame to frame. Finally, if we find the synchronization bits

out of place, we can easily disqualify the fragment. It would not be difficult to perform a quantitative study similar to the one from the previous section to see what is the probability that a non-*mp3* fragment will not deliver rejection proof.

We performed an exhaustive study using the 1,500 and 4,096 fragment sizes and found that the above criteria did not select a single false positive among our *doc, xls, pdf, jpg,* and *gz* sets. We did not study the 512 byte case because we needed to ensure that at least two frame headers are present in the fragment.

As further proof, we created a tool called *mp3scalpel* and used it to extract all the *mp3* fragments from the DFRWS 2006 Challenge [1], which contained numerous non-trivial fragmentation scenarios among several different data formats. With some additional work we were able to automatically reconstruct all the available audio from the challenge.

4.5. Compound document detection

It is a highly desirable to be able to detect not only that a fragment contains part of a compressed object but is also part of a compound file container. We should again emphasize that this determination is not always feasible—we need some data outside the embedded object to appreciate the fact that it is embedded. How much do we *really* need?

As it turns out, for the *pdf* format the answer seems to be very little as *pdf* rather clearly marks all embedded objects. The beginning of an embedding is preceeded by the string '`>>stream`' followed by an end-of-line marker, where as the end is marked by '`endstream`' and another end-of-line marker. In our test set we gradually reduced the number of characters to five for both the beginning and the end and encountered no false positives when tested against the rest of the compound files—*doc* and *xls*.

In contrast, we found no reliable patterns that precede embedded jpegs in *doc* containers. This is not surprising since these are not organized as records but are similar to the FAT file system [9]. Therefore, there is no reason to expect that such information will be in close proximity to the actual embedded object.

4.6. Summary

In this section, we presented a number of specialized classification techniques that are targeted at specific data formats. Our goal was not to claim groundbreaking discoveries but to illustrate that it is possible to produce very reliable fragment classification results if one is willing to read the standards, examine files by hand, and create hand-crafted recognizers.

All of the presented techniques fit our requirement in terms of accuracy, error estimates, clarity, and performance. Our bottom up analysis shows also that many of the artifacts that we use in the analysis a very particular to individual formats so it is extremely unlikely that some generic framework will cover all of them.

Much of the information we used is part of standards specifications and has be been used in many applications, such as file carving. However, no prior work has provided any error estimates on the reliability of such methods.

5. Conclusions and Future Work

The goal of this work was to help advance and focus the research effort in the area of file fragment classification. Specifically, we make the following main contributions:

- We critically examined the state of research efforts in the area over the last nine years and evaluated them both in terms of end results and research methodology.
- We performed a bottom up analysis by starting with the details of how some popular formats are structured. We reached the conclusion that the methods currently applied are highly unlikely to be successful because they rely on easily detectable patterns.

- We found that the current evaluation efforts are inadequate in at least two respects—they do not employ enough sample data and fail to make the distinction between primitive types and compound data formats. The latter problem has major implications regarding the execution of evaluation studies and the presentation of their results in proper context.
- We argued that efforts need to be focused on developing specialized classification methods as the only way to produce practical tools. This may not have the same research elegance but yields results that are much closer to what is needed. In retrospect, most successful methods to date were specialized and worked only on certain data.
- We demonstrated that using specialized methods does not relieve us from the need to be critical and to quantify the effectiveness of each one based on representative studies.

In the near future, we plan to present to the community a set of corpora that researchers can adopt and use for standardized testing and evaluation. This will finally permit both head-to-head evaluation and absolute performance evaluation based on the ground truth.

5. References

[1] B. Carrier, E.Casey, W.Venema, DFRWS Forensic Challenge. http://dfrws.org/2006/challenge/. 2006.

[2] C. Freiling, D. Jungreis, F. Théberge, K. Zeger. "Almost all complete binary prefix codes have a self-synchronizing string." IEEE Transactions on Information Theory. Vol 49(9), Sep 2003, pp. 2219-2225.

[3] Calhoun WC, Coles D. "Predicting the types of file fragments." Proceedings of the 2008 DFRWS Conference, Baltimore, MD. Aug 2008. pp.146-157.

[4] I. F. Darwin, "Libmagic,". ftp://ftp.astron.com/pub/file. 2008

[5] Karresand M, Shahmehri N. "File Type Identification of Data Fragments by Their Binary Structure", Proceedings of the 7th Annual IEEE Information Assurance Workshop, "The West Point Workshop", pp 140-147, United States Military Academy, West Point, 21-23, New York, June 2006.

[6] Karresand M, Shahmehri N. "Oscar—file type identification of binary data in disk clusters and RAM pages," in Proceedings of IFIP International Information Security Conference: Security and Privacy in Dynamic Environments (SEC2006), LNCS, p413-424. 2006.

[7] Li WJ, Wang K, Stolfo SJ, Herzog B. "Fileprints: identifying file types by n-gram analysis". 6th IEEE Information Assurance Workshop, West Point, NY, June, 2005.

[8] McDaniel M, Heydari MH. "Content Based File Type Detection Algorithms." HICSS '03: Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03). 2003.

[9] Microsoft, "Windows Compound Binary File Format Specification," 2007.

[10] Moody SJ, Erbacher RF. SÁDI - Statistical Analysis for Data Type Identification. May 2008. Proceedings of the 3rd IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering, Oakland, CA, May 2008, pp. 41-54.

[11] P. Deutsch, J-L. Gailly. RFC 1950: "ZLIB Compressed Data Format Specification version 3.3". http://www.ietf.org/rfc/rfc1950.txt. 1996.

[12] P. Deutsch. RFC 1951: "DEFLATE Compressed Data Format Specification version 1.3". http://www.ietf.org/rfc/rfc1951.txt. 1996.

[13] R. Erbacher, J. Mulholland, "Identification and Localization of Data Types within Large-Scale File Systems," Proceedings of the 2nd International Workshop on Systematic Approaches to Digital Forensic Engineering, Seattle, WA, April 2007, pp. 55-70.

[14] S. Garfinkel, J. Migletz, "The new XML Office Document Files," *IEEE Security & Privacy Magazine*, March/April 2009.

[15] Sladen, P., *pyflate*. http://www.paul.sladen.org/projects/pyflate/

[16] Veenman CJ. Statistical Disk Cluster Classification for File Carving. Proceedings of the First International Workshop on Computational Forensics, Manchester, UK, August 31, 2007.

[17] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. "On the self-similar nature of ethernet traffic." In Proc. ACM SIGCOMM '93, pages 183—193, 1993.

[18] Welch, T. A. "A technique for high-performance data compression." Computer. Vol. 17, June 1984. pp. 8-19.