Digital Investigation 10 (2013) S69–S77

Contents lists available at SciVerse ScienceDirect

Digital Investigation

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

File fragment encoding classification—An empirical approach

Vassil Roussev*, Candice Quates

Department of Computer Science, University of New Orleans, New Orleans, LA 70148, USA

Keywords:
File type classification
Data encoding identification
Zsniff
Digital forensics
Sub-file forensics

A B S T R A C T

Over the past decade, a substantial effort has been put into developing methods to classify file fragments. Throughout, it has been an article of faith that data fragments, such as disk blocks, *can* be attributed to different file types. This work is an attempt to critically examine the underlying assumptions and compare them to empirically collected data. Specifically, we focus most of our effort on surveying several common *compressed* data formats, and show that the simplistic conceptual framework of prior work is at odds with the realities of actual data. We introduce a new tool, *zsniff*, which allows us to analyze deflate-encoded data, and we use it to perform an empirical survey of deflate-coded text, images, and executables. The results offer a conceptually new type of classification capabilities that cannot be achieved by other means.

1. Introduction

File fragment classification—*the process of mapping a sample chunk of data, such as a disk block, to a specific type of data encoding*—has been the focus of substantial research efforts for over a decade. Unfortunately, the practical results of all this work have been underwhelming, at best. This purpose of *this* work is to:

a) clearly spell out the research problem;
b) identify explicit tool requirements based on usage scenarios; and
c) present the empirical background work behind a new classification tool called *zsniff*.

1.1. Motivation

Most digital forensic and incident response professional can look (through a hex editor) at a piece of binary data, such as a disk block or a network packet, and readily identify what kind of data it carries. This skill is developed with experience, and can be very helpful in a variety of forensic tasks, such as diagnosing break-ins, making sense of residual data, decoding memory dumps, reverse-engineering malware, data recovery, and so on.

The problem is that manual examination does not scale and we need automated tools to perform this fragment classification. Once we have a tool that can successfully classify fragments, it can be used to: statistically sample target during triage (Garfinkel et al.), improve file carving (Richard and Roussev, 2005) accuracy, screen network traffic, and provide context for other subsequent forensic processing (e.g., indexing).

1.2. Requirements

Based on the scenarios in the previous section, we identify the following main requirements for a fragment classifier:

Accuracy. This is always the primary requirement with any classification problem; however, we posit that digital forensic applications require *near-perfect* classification rates. The main problem are terabyte-scale targets, where even 95% accuracy would produce an unacceptable number of false positive/negative results.

* Corresponding author.
E-mail addresses: vassil@roussev.net, vassil@cs.uno.edu (V. Roussev), candice@egobsd.org (C. Quates).

1742-2876/$ – see front matter
http://dx.doi.org/10.1016/j.diin.2013.06.008

Line-speed performance. Methods should be fast enough to keep up with bulk data transfers from secondary storage and commodity networks: 100 MB/s and up.

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. Classifiers need to recognize their own limitations, and communicate their level of confidence in the results. This will allow users and higher-level analytical tools to better understand what encoding classification can and cannot do.

2. Related work

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

McDaniel and Heydari (2003) considers 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 codes in the file to be classified. This histogram is 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.

Plain *byte frequency distribution* (BFD) analysis yields rather unconvincing results: 27.50% true positive rate for the BFA algorithm and 46% for the BFC algorithm. McDaniel 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 and would not work for classifying arbitrary file fragments.

The next important piece of work belongs to Li et al. who substantially revamped the BFD approach. 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 a puzzling and unexplained result—using more data yields less accurate results.

Karresand and Shahmehri developed a very similar centroid idea to Li's and called it the Oscar method. This was shortly extended (Karresand and Shahmehri, 2006) with the introduction of a new metric called rate-of-change (RoC), which was defined as the difference of the values of consecutive bytes. The latter is not truly a generic feature, rather it is suitable for detecting *jpeg*-encoded data only. The reason this works is due to a quirk of the *jpeg* format, which utilizes byte stuffing. The 0xFF is used as an escape character marking the beginning of all metadata tags; thus, an extra 0x00 is placed after every 0xFF byte in the body of the file. This produces a reliable and highly characteristic pattern—0xFF00—which has a very high RoC. Apart from *jpeg*, 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, (Karresand and Shahmehri) 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%; (Karresand and Shahmehri) Fig. 4.

2.2. Statistical approaches

Erbacher and Mulholland argue 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. Unfortunately, the argument is rather thinly supported by plotting the behavior of these measurements over several specific files.

Follow-up work by Moody and Erbacher (2008) attempts to develop a general methodology based on the observations. Unfortunately, the proposed statistics can only distinguish broad classes of data encodings, such as textual, executable code, and compressed, but becomes easily confused in trying to pick out more subtle differences—e.g., *csv* vs. *html* vs. *txt*. Secondary analysis is made to make a finer distinction with mixed results.

Veenman (2007) combined the BFD with Shannon entropy and Kolmogorov complexity measures as the basis for his classification approach. To his credit, he used a non-trivial (450 MB) evaluation corpus, employed 11 different file formats, and his formulation of the problem—identify the container format for a 4096 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 and Coles expanded upon Veenman's work by employing a set of additional measures (16 total) and combinations of them. While the test sets were small—50 fragment per file type—he recognized the need for more subtle testing and performed all-pairs comparison of three compressed formats: *jpeg*, *gif*, and *pdf*. Another positive in this work is that header data is 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 14–40% range.

More recent work has targeted incremental improvements but has not produced any ground breaking ideas. Due to space limitations, we will stop here—well short of a comprehensive survey of the field.

3. Redefining the problem

Overall, prior work in the area of fragment classification exhibits two critical problems:

a) it fails to provide a workable problem formulation that is in line with real-world file formats; and
b) it fails to provide an appropriate evaluation framework and reproducible results.

In many respects the latter is a consequence of the former, so our first task is to properly state the problem we are trying to solve. The basic *conceptual problem* is that researchers have continually *confused* the notions of *file type* and *data encoding*. The purpose of this section is to differentiate between the two, and to derive a correct problem formulation that enables scientific evaluation and interpretation of the classification results.

3.1. Application file types

Let us start with the basic notion of a file. *A file is a sequence of bytes that is stored by a file system under a user-specified name*

Historically, operating systems have avoided interpreting the names and contents of files; however, every operating system needs to be able to at least determine if a file is executable. For that purpose, it uses a combination of file naming conventions and a magic number—file type- specific binary string at the beginning of a file—to perform sanity checking before attempting execution. For example, CP/M and MS-DOS will load any file with extension .com into memory and execute it, if the file's name is typed; MS Windows marks the beginning of executables with the string MZ; Linux marks the beginning of executables with the hexadecimal string 7f 45 4c 46, and so on.

For modern, more user friendly, operating systems it became helpful to associate data files with the corresponding applications so that actions like 'open', 'new', and 'print' could be initiated by the user from the OS interface. This loose association between file naming and applications is what the operating system presents to end-users as a '*file type*'.

It is important to recognize that this is not a particularly useful definition from a forensic perspective. Under the hood, the internal data representation of a file type can change radically between application versions. Moreover, compound file formats can have dramatically different content and, consequently, look very different at the bit stream level. For example, an MS Powerpoint presentation document could be primarily text, or it could include a lot of photos, scanned images, audio/video files, spreadsheets, etc. Since the individual components have different encodings, the layout of the data in the file would be very different in each of these cases.

The latter observation directly contradicts the central belief of prevalent machine-learning-based methods that an application-defined file type has characteristic statistical attributes that are useful for classification purposes. Such an assumption works to a certain extent for simple formats, like ASCII text, but is patently wrong in the general case. Therefore, we must rebuild the definition of a file type from the ground up, starting with the definition of primitive data encodings.

3.2. Data encodings and file types

A data encoding is a set of rules for mapping pieces of data to a sequences of bits. Such an encoding is primitive, if it is not possible to reduce the rule set and still produce meaningful data encodings.

The same piece of information can be represented in different ways using different encodings. For example, a plain text document could be represented in ASCII for editing, and in compressed form for storage/transmission. Once encoded, the resulting bit stream can serve as the source for further (recursive) encodings; e.g., a base64-encoded *jpeg* image.

A file type is a set of rules for utilizing (sets of) primitive data encodings to serialize digital artifacts

Unlike data encodings, file types can have very loose, ambiguous, and extensible set of rules, especially for complex file types. Consider the breakdown of recursively embedded MS Office objects found inside a set of ~20,000 MS Office (Table 3). A Word document may contain a Powerpoint presentation, which in turn may contain Excel spreadsheet, which may contain OLE objects, and so on. To correctly discover the entire stack of embeddings, one needs the *entire* file as a frame of reference. Contrast that to our problem, where we have a (small) fragment to work with.

Even the simple (but common) case of including *png*/*jpeg*-encoded images in a compound document illustrates the inadequacy of the status quo. For example, assume that method *A* advertises that it can correctly classify *pdf* fragments of 1024 bytes in size 80% of the time and it misclassifies them as *jpeg* pieces 15% of the time. Is this a good method to adopt, or not? The answer is unknowable:

On the one hand, 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 *deflate*-encoded parts, this would be a rather poor method as it is relatively easy to separate *jpeg*- and *deflate*-coded streams.

Also, how will the method perform if we encounter a group of fax-encoded scanned documents?

Due to the poor problem formulation, we have no real basis to understand the performance of the method. Yet, one can find multiple examples (e.g., Calhoun and Coles) where researchers treat compound file types as primitive data encodings and produce effectively meaningless classification rates and confusion matrices.

3.3. Problem statement

We submit that file fragment classification problems consists of three autonomous sub-problems. Given a sample fragment, a classifier should be able to answer the following questions:

1. *What is the primitive data encoding of the fragment?*
2. *Does the encoding contain recursive encodings?*
3. *Is the fragment part of a compound file structure?*

(The second question is strictly optional but is often interesting for simple encodings. For example, identifying a fragment as being *base64*-encoded is trivial but not very informative; discovering that it encodes an image, or text is substantially more useful.)

Note that we can always make a meaningful effort to answer the first two questions. Our ability to answer the third one depends, to a much larger degree, on luck as the fragment must contain enough characteristic data from the container.

By splitting the problem into three separate questions, we can begin to build a meaningful and informative evaluation framework. The split also means that establishing the ground truth becomes much more complicated than simply collecting a set of files with a given extension—we need to *parse* the contents of our reference files to know *exactly* what is in each evaluation sample.

Evidently, the *size* of a fragment is a crucial evaluation parameter and performance can vary greatly. At the extremes, if the fragment is too small, there will not be enough data to work with; as the fragment gets big the results may get noisier as multiple encodings may be detected.

Another aspect of the evaluation process is defining the required level of specificity. For the same piece of data, we can often provide multiple classification in increasing order of specificity—e.g., text → xml → mathml—and classifiers should aim for the most specific result.

Finally, we submit that the true test of a classification method is to be performed on independent fragment samples that contain *no* header information. This is not to suggest that identifying headers is useless but that a header sample is the outlier case that is readily solvable with some basic parsing code.

4. Design of *zsniff*

The overall goal of our work is to build *zsniff*—an extensible classification framework, populate it with a set of high-quality classifiers for the most common data encodings, and leave it open to further extensions.

Some of the classifiers are inherently more compute intensive. Therefore, our plan is to build a tree of classifiers—from the most generic to more specific ones that (hopefully) filter out most poor candidates based on easy to compute attributes.

For the rest of the section we *briefly* discuss the various classes of known useful classifiers, and focus most of our attention on the *deflate* classifier, which we have developed. We omit all discussion on text classification, as this is a mature subject by now.

4.1. Simple classifiers

The purpose of simple classifiers is to provide a quick and general classification; based on the results, more specific (and computationally expensive) classifiers are chosen for follow-up work. Below are a couple of examples of these:

4.1.1. Entropy

Shannon's classical formula allows us to quickly place the observed (probability distribution of) data into three useful categories: *low*, *medium*, and *high* entropy. All compressed, encrypted, and random data would exhibit high entropy; text and code—medium; and sparse data, such as large tables, logs, etc., are usually of low entropy.

4.1.2. Base16/32/64/85

These are simple classifiers that detect the most common binary-to-text encodings. The detection is straight forward as each encoding uses a well-defined range of ASCII codes. Once identified, the blocks using these encodings can be converted to binary and sent back to the beginning of the pipeline for second-order classification.

4.2. N-gram classifiers

N-gram classifiers rely on characteristic strings that can provide both definitive classification and more subtle clues, such as the use of specific markup. These classifiers are particularly useful in identifying *format keywords* for top-level containers, and are often the only reliable clue in that regard.

4.3. Format parsers & verifiers

Format parsers look for synchronization markers in the input and attempt to parse the remaining data according to the format rules for the particular encoding. Since this is meant to be a fast process, the parser will not attempt full decoding of the data. Additional sanity checks can be applied to data to reduce errors as illustrated by the following example classifiers:

4.3.1. Mp3

Mp3 encoding is classification-friendly and is representative of a broader class of audio/video containers. An *mp3* file consists of a sequence of independent, self-contained *mp3* frames. The beginning of a frame is marked by 12 consecutive bits set to 1—a synchronization marker— 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 (Roussev and Garfinkel).

There are several criteria that allows the confirmation/rejection of the fragment as mp3-encoded. First, the predictable appearance of the frame header—for a fixed-rate encoding this will be almost periodic. 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.

Jpeg header recognition is relatively easy to accomplish—the header has a variable length record structure in which synchronization markers are followed by the length of the field. Thus, some simple 'header hopping' can reliably identify the header, much like in the *mp3* case.

Jpeg body recognition is also not difficult to accomplish as the encoding uses byte stuffing that results in the 16-bit hexadecimal FF00 occurring every 191 bytes, on average (Roussev and Garfinkel). Placed next to a high-entropy sample with a different encoding, like deflate, this feature sticks out rather prominently and can be incorporated into the classifier in a number of ways. This can be done quickly and efficiently, and we consider this a solved problem.

4.4. Deflate classifier

Our starting assumption is that the methods described are applied to high-entropy data, and the real problem is separating the various types of compressed data, and to distinguish them from encrypted/random data.

The deflate classifier is the main contribution of this work and seeks to: a) identify deflate-coded data; and b) distinguish among different types of underlying data.

The *zlib*/*deflate* encoding (RFC, 1950/1951) is entirely focused on storage efficiency and contains the *absolute minimal* amount of metadata necessary for decoding. It consists of a sequence of compressed blocks, each one comprised 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. In practice, dynamic Huffman is present 99.5% of the time (Roussev and Garfinkel)).

Huffman tables. These describe the Huffman code books used in the particular block.

Compressed data. The table is followed by a stream 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.

As soon as the end-of-block code is read from the stream, the next bit is the beginning of the following block header—there is no break in the bit stream between blocks, and there are no synchronization markers of any kind. The end-of-block code depends on the coding table, so it varies from block to block. The upshot is that, absent sanity checking, a deflate decoder can sometimes "decode" even random data.

Using prior open source *deflate* implementations as a base, we built a robust, stubborn, and heavily instrumented C++ version of the deflate decoder. Robust means that we can throw *any* data at the decoder and it will either successfully decode it, or fail gracefully.

Stubborn means that the decoder will successively attempt to decode the given data starting at *every* bit offset. While this is a brute-force approach, we see no alternatives. One shortcut is that virtually all deflate blocks in the wild use dynamic Huffman, which means that we have a two-bit header pattern we can key off, saving us 75% of the work.

Once we have found a candidate block header, we attempt to decode the Huffman tables that should be present in a correct block. Using a combination of parsing failures and sanity checks on decoded tables, we eliminate most candidates within a few hundred bytes, so the attempt is relatively cheap.

If the tables are deemed correct, we proceed to decode the block until we reach the end-of-block marker and restart the decoding process. One nice side effect of this behavior is that deflate data does not need to be separated from other (differently coded) data *a priori*.

We instrumented the decoder to output the decoded tables, as well as block boundaries, and this data serves as the basis for our empirical analysis of *deflate* data that follows.

We analyze the content of three different types of deflate-encoded types of data. The goal is to establish ground truth with respect to what is feasible to achieve based on block size, and to search for characteristic patterns in the Huffman tables that would allow robust classification.

5. Analysis of MS Office 2007 files

We start with an empirical analysis of MS Office 2007 files—docx, xlsx, and pptx—which we refer to as *msx* files.

5.1. Data sample

Since we are not aware of any public data set that provides an appropriate sample, we created our own, called *MSX-13* (Table 1).

We built it using the results of a popular search engine to identify approximately 10,000 candidate documents of each type. Due to search engine restrictions, it is difficult to obtain more than 1000 results per query. Therefore, we ran 10 instances of the same query with different *site* restrictions. We used queries of the form ext:<file_type> site:<tld>, where

<file_type> = docx|xlsx|pptx, and
<tld> = com|net|org|edu|gov|us|uk|ca|au|nz.

In other words, we asked for a collection of files based on file type, with no particular keyword. The split along TLDs ensures that the results do not overlap and the chosen TLDs heavily favor English documents. Once downloaded, the sample files were validated to ensure correctness. A detailed description of the data set is available at http://roussev.net/msx-13.

5.2. Content

Broadly, *msx* files are zip files that consist of deflate-encoded files/components (almost entirely in *xml*), and

Table 1
MSX-13 data set statistics.

	docx	xlsx	pptx
File count	7018	7452	7530
Total size (MB)	2014	1976	20,037
Avg size (KB)	287	265	2661

embedded media content that is stored in its original (compressed) encoding. Table 2 provides a breakdown of the major components based on four statistics: *count*—the total number of components of the given type; *avg size*—average size per component; *total size*—the total volume of all components of the given type; and *percent of total*, which provides the fraction (by volume) of all components relative to the size of the respective data set.

The first category of components—referred to as *deflate*—represent all deflate-coded objects. The remaining categories represent *stored* objects in their native encoding—*jpeg*, *png*, *gif*, and *tiff* represent image data, the *other* category encompasses embedded *msx* documents, fonts, *audio/video*, and a tiny number of unclassified.

Observations

Although the three file formats share a common design (and some components), the actual content differs substantially.

Xlsx data is dominated (84%) by deflate content and has the highest average size of deflate objects, whereas the other two are dominated by embedded images with 66% (*docx*) and 83% (*pptx*) of content.

Generally, small objects (deflate) are good news as the beginning/end have readily identifiable markers and can save us deeper analysis. *Jpeg* content is also good news as it can be identified quite reliably.

Png content, quite extensive in *docx*/*pptx* data, is *not* good news—it consists of *deflate*-coded image data. Consequently, we need deeper analysis to distinguish it from deflate-coded *xml* data.

As already mentioned earlier, *msx* objects can be embedded recursively, which creates additional challenges. Table 3 shows a breakdown of all embedded objects found in our test data; each row represents the number of embedded objects of the given type on each of the host file types. The *bin* group consists almost entirely (98.5%) of OLE objects, which are stored deflate-compressed.

The most important takeaway is that one size is unlikely to fit all and, most likely, we need a whole bag of tricks to correctly classify the *msx* formats.

Table 2
Content breakdown of MSX-13 files.

	docx	xlsx	pptx
deflate			
Count	128,088	199,899	1,002,162
Avg size (KB)	5	8	3
Total size (MB)	651	1652	2693
Percent of total	32	84	13
jpeg			
Count	5644	2838	59,067
Avg size (KB)	142	68	121
Total size (MB)	802	193	7147
Percent of total	40	10	36
png			
Count	6777	1728	65,692
Avg size (KB)	68	46	134
Total size (MB)	462	80	8820
Percent of total	23	4	44
gif/tiff			
Count	574	193	4261
Avg size (KB)	102	32	160
Total size (MB)	59	6	680
Percent of total	2.9	0.3	3
Other			
Count	431	90	3606
Avg size (KB)	30	54	107
Total size (MB)	13	5	384
Percent of total	0.6	0.2	1.9

Table 3
MSX-13: Recursively embedded objects statistics.

	docx	xlsx	pptx	Total
bin	844	105	6842	7791
doc	106	30	221	357
docx	18	30	163	211
ppt			26	26
pptx	2		4	6
xls	71	4	693	768
xlsx	275	1	2951	3227
Other	7		53	60
Total	1323	170	10,953	12,446

5.3. Format keywords

One simple and effective means to reliably classify fragments is to find keywords (*n*-grams) that are highly characteristic of the encoding. In this case, the overall data format is a generic (*zip*) archive; however, the file/directory structure of its content is highly characteristic. We used our data set to empirically derive the most useful keyword roots from the names of the constituent files. The result are as follows: *docx*—23 keywords (of the form word/*); *xlsx*—22 keywords (xl/*); *pptx*—21 keywords (ppt/*); *msx*—five keywords common to all three formats. With one exception, all keywords are between 8 and 13 characters.

For the next study (Table 4), we identified the keyword locations on our test set, and calculated the fraction of blocks that contain a keyword for several block sizes.

The results show that, for small block sizes, it is inherently difficult to unambiguously classify *msx* blocks. (Recall that only blocks containing the keywords can be unambiguously attributed to *msx* documents—the rest is compressed xml and media data.) At 64KiB, almost 1/4 of the average file size for docx/xlsx, the classification rate approaches 50% and starts to become a practical, low-cost approach.

It is useful to understand what is the main contributor to the relatively low efficiency of keyword searches for small block sizes. Since this is effectively a function of component file sizes, one possible explanation is that (presumably bigger) embedded media files are the main reason. To test, we split each of the file sets into two subsets—those that contain media, and those that do not—and compared the detection rates. We found that the

Table 4
Fraction of blocks identifiable by keyword.

block size	docx	xlsx	pptx
1024	0.061	0.078	0.044
4096	0.115	0.143	0.077
16,384	0.228	0.259	0.152
65,536	0.476	0.474	0.343

hypothesized effect is real but small, and does not account for our observations.

Instead, we studied the effects of large components by comparing the ratio of the size of the largest component file to the size of the entire document. Fig. 1 shows the empirical distribution for the entire *docx* set; note that both axes are logarithmic.

With respect to absolute size of the maximum sized component we see an approximately linear relationship with overall file size. This confirms our hypothesis that larger files are more difficult to classify.

In summary, components for *msx* files routinely exceed the size of most fragment sizes of interest, which places inherent restrictions on our ability to perform keyword-based classification and, by extension, unambiguous file type identification.

5.4. Compressed block distribution

The next step is to dig deeper and start looking at our chances of identifying data encodings. We consider the distribution of deflate compressed blocks sizes. This data can help us relate fragment size and classifier performance.

Fig. 2 plots the cumulative distribution of compressed block sizes for four different deflated-coded data samples. The *docx* and *xlsx* statistics are based on the 3191 *docx* and 1291 *xlsx* files from *MSX-13* that do *not* contain embedded media (the *png* and *exe/dll* data are discussed in later sections).

For *docx*, a 1/4/16KiB fragment has a respective 65.6/90.2/99.6% chance of containing a whole block. For *xlsx*, the numbers are 35.9/47.4/94.9%. The practical significance of these percentages that they place an effective upper limit on the performance of deflate classifiers for *docx*/*xlsx* classifiers.

Alternatively, we can say that we need a minimum of 18KiB fragment to afford near-perfect chance of detecting any present *msx* deflate block.

6. Analysis of *png* files

The *png* image format is a primitive data encoding that utilizes *deflate* as its last step of the compression process. In other words, it contains no overt (statistical) features that can be used to distinguish it from compressed *xml* data, for example. The rest of this section is an empirical search for classification criteria to separate the two.

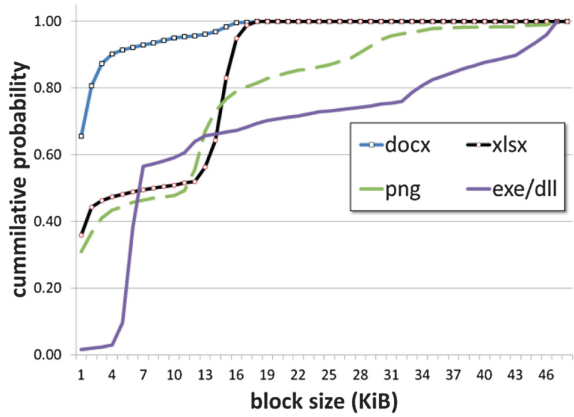

Fig. 2. C.d.f. of compressed block sizes.

First, we find the distribution of compressed block sizes (Fig. 2). Clearly, the distribution is more spread out, with only 81.5% of compressed blocks below the 18KiB threshold established in the prior section. However, block sizes do top out around 48KiB, so a fragment of that size should be classifiable, if appropriate criteria are found.

In search of such criteria, we plot (Fig. 3) the cumulative distribution of the number of valid (Huffman) code table values for the 256 ASCII characters vs. the corresponding distribution for the *msx* files discussed in the previous section (*exe/dll* is discussed later).

We can see very clear differences that have classification value—any code table of size 60, and less, almost certainly indicates a *png* block, whereas any code table of size 140, and higher, indicates that it is *not png*. Together, these two ranges cover about 30% of all observed code tables. By looking at the *p.d.f.* (omitted here) we can cover another 30% of the range where the difference in probabilities is substantial and will yield a reliable distinction criterion. Yet, for the rest, code table size is not enough.

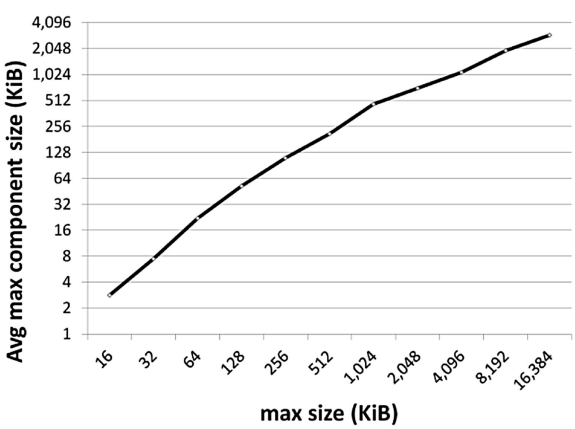

Fig. 1. Largest component file statistics (docx).

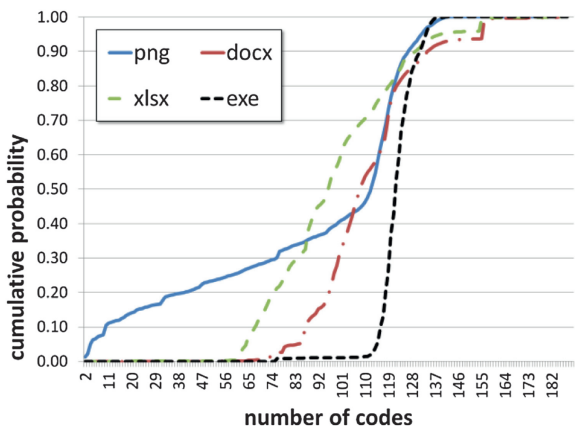

Fig. 3. C.d.f. of code table sizes.

It is time to consider code table *content*. To bootstrap the decoding process, the Huffman table must define a bit string code for every byte value (0–255) *that will be needed*. For *msx* blocks, this means defining a code for every ASCII character *in use*. Since this is a limited range, we can hope that it would be different than the one for *png*, which is encoding pixels.

Fig. 4 shows, for each ASCII code, what is the empirical probability that it will be defined (*docx*/*xlsx* shown with bars to accommodate large jumps between neighbors). For example, for *png*, code 0 appears with probability 0.9778, whereas for *docx*/*xlsx*, that number is 0.0005. At the same time, for code 115 ('s'), we have 0.9991 for *docx*, 0.6900 for *xlsx*, and 0.2559 for *png*.

The chart illustrates the opportunity for near-perfect distinction between *png* deflate-encoded data and deflate-encoded text/markup. Follow up work needs to establish the shortest and most reliable combinations of codes to place in the classifier.

7. Analysis of deflate-coded PE files

Let us briefly examine what happens to *deflate*-encoded Windows portable executables (PE). This is relevant as installation packages are often deflate-encoded. Following the analysis template from the previous section, we placed 1000 exe/dll files from C:\Windows\system32 into a zip archive and looked at the resulting Huffman code table statistics.

First, the distribution of code table sizes (Fig. 3). It is very clear that compressed PE files exhibit the *cdf* of a narrow, bell-shaped distribution with 98.7% of the mass concentrated in the 110–140 range.

Next, we consider the distribution of valid codes in the table (Fig. 4), which also exhibit a very regular and recognizable signature. Specifically, the codes in the 0–114 range show a uniform distribution, which is quite sufficient to formulate strong classification criteria.

8. Discussion

To summarize, our empirical analysis of real-world data (using the *zsniff* tool) has demonstrated the feasibility of reliably distinguishing among various types of compressed data encodings. Specifically, we can distinguish among deflate-coded text/markup, deflate-coded image data (*png*), and deflate-code PE. Combined with prior capabilities to detect *jpeg* and *mp3* encodings, we have the solid beginnings of an advanced fragment classification framework that can cover almost all the data encodings observed in our test corpus.

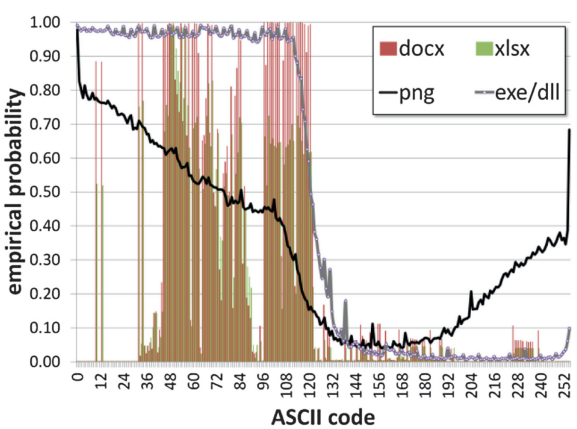

Fig. 4. P.d.f. of deflate codes 0–255 by file type.

We should note that simply decoding any compressed block and analyzing it in source is not a practical approach. The reason is that the compressed blocks are often not self-contained and reference data from previous blocks. Brown (2011) discusses this problem in the context of data recovery.

Due to space limitations, a number of lesser statistical studies have been omitted. For example, one potentially useful classification feature is the length of the code words for each code. Shorter codes indicate higher frequency and vice versa. Thus, we can treat code word lengths as a crude histogram sketch of the symbol distribution in the source.

Performance is an obvious concern with any brute-force method, such as our decoding attempts. The key to making it work in a reasonable amount of time is to fail as quickly as possible on bad input. On actual deflate data, the current quick-and-dirty implementation of *zsniff* can run at about 5–6 MB/s on a single core.

On random data, our control and worst-case performance scenario, it works considerably slower. As one would expect, the tool occasionally decodes plausible Huffman code tables (false positives). The good news is that we never encountered two apparent compressed blocks that are back-to-back (as they appear in true deflate data). This observation is something to be explored further as it can help reduce false positives.

9. Conclusions

This work makes several contributions to the field

Improved methodology. We analyzed the shortcomings of prior work and showed that it has a fundamental flaws in its assumptions. We corrected these foundational problems by reformulating fragment classification as a set of three autonomous problems that are to be pursued and evaluated independently.

New reference data set. We built and validated a set of some 20,000 MS Office 2007 files that can be used for research purposes by the community.

New tool for parsing deflate-encoded data. We built a robust tool that can automatically discover *deflate*-coded data, generate a number of useful statistics, and (optionally) can decode the data.

Empirical analysis of deflate-coded data. We developed a new methodology for characterizing the encoded data by using information from the Huffman compression tables. This analysis shows that coding tables for text/markup, image data, and Windows executables have distinct signatures and can be readily distinguished. Further, we quantified the relationship between fragment size and expected optimal classification performance on deflate-coded data.

The end result of this effort is that we have qualitatively new capabilities that can disambiguate various deflate-compressed data formats; and, ultimately, deflate data from encrypted data.

The current implementation of the *zsniff* tool can be found at http://roussev.net/zsniff.

References

Brown R. Reconstructing corrupted DEFLATEed files. Proceedings of the 11th annual DFRWS conference. New Orleans, LA; Aug 2011.

Calhoun WC, Coles D. Predicting the types of file fragments proceedings of the 2008 DFRWS conference. Baltimore, MD; Aug 2008. p. 146–57.

Erbacher R, Mulholland J. 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. p. 55–70.

Garfinkel S, Nelson A, White D, Roussev V. Using purpose-built functions and block hashes to enable small block and sub-file forensics. In: Proceedings of the tenth annual DFRWS conference (DFRWS'10). Portland, OR; Aug 2010.

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 2006. p. 413–24.

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", United States Military Academy, West point, 21–23. New York; June 2006. p. 140–47.

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.

McDaniel M, Heydari MH. Content based file type detection algorithms. In: HICSS '03: Proceedings of the 36th annual Hawaii international conference on system sciences (HICSS'03) 2003.

Moody SJ, Erbacher RF. Sádi - statistical analysis for data type identification. In: Proceedings of the 3rd IEEE international workshop on systematic approaches to digital forensic engineering, Oakland, CA, May 2008 May 2008. p. 41–54.

Roussev V, Garfinkel G. File fragment Classification—The case for Specialized approaches. Fourth international IEEE workshop on systematic approaches to digital forensic engineering, Apr 2009, Oakland, CA. DOI: 10.1109/SADFE.2009.21.

Richard G, Roussev V. Scalpel: a frugal, high performance file carver. Digital Forensic Research Conference, http://dfrws.org/2005/proceedings/richard_scalpel.pdf; 2005.

Veenman CJ. Statistical disk cluster classification for file carving. Proceedings of the first international workshop on computational forensics, Manchester, UK, August 31, 2007.