

# A Visual Study of Primitive Binary Fragment Types

Gregory Conti\*, Sergey Bratus, Anna Shubina†,  
Andrew Lichtenberg‡, Roy Ragsdale, Robert Perez-Alemany,  
Benjamin Sangster, and Matthew Supan\*

July 4, 2010

## Abstract

We argue that visual analysis of binary data objects such as data files, process memory, and file systems presented as grayscale graphical depictions helps distinguish structurally different regions of data and thus facilitates a wide range of analytic tasks such as fragment classification, file type identification, location of regions of interest, and other tasks that require an understanding of the “primitive” data types the objects contain. We believe that, due to the high visual value of this data presentation, such visual analysis is an invaluable help in low-level study of binary data objects and in understanding their structure, and that tools for such visual analysis belong in the toolkit of every researcher studying binary data.

In an effort to facilitate development of such tools, this paper presents a visual study of binary fragments created by common kinds of software, and offers a descriptive taxonomy of primitive binary fragments and their graphical depictions. Although significant research has gone into the study of binary fragments, the depth and breadth of this study to date has been limited. Thus the primary contribution of this paper is an extensible and visual taxonomy to assist and inform researchers conducting low-level analysis of binary objects.

## 1 Introduction

Binary data is central to the modern computing paradigm and exists in large, complex objects such as process memory, file systems, network flows, as well as in data and executable files. In many cases, however, these objects are

---

\*West Point

†Dartmouth College

‡Skidmore College

heterogeneous, consisting of many dissimilar elements created by commonly used software libraries and processes or algorithms (such as various forms of compression). We call these dissimilar elements *primitive binary fragments*.

Even though the variety of data types is theoretically limitless, in practice they mostly get created by a limited number of popular tools for an equally limited number of consuming programs. Thus most fragments of complex binary objects we are likely to encounter can be categorized into a number of recursively refined categories, a *taxonomy*.

To use a biological metaphor, even though there is no limit to the kinds of possible living organisms, nor to their ability to imitate each other to confuse the observer, in reality the ones we encounter can in fact be usefully classified into species, families, etc., based on the common visual features they exhibit. *In this paper, we set out on a journey to define and study the prevalent “binary species”*.

Primitive binary fragments are typically homogeneous, irrespective of potential header and footer values. This property is important when analyzing a fragment where the header and footer may be unavailable, untrustworthy, or unnecessary. Examples of fragment types include compressed MPEG-1 Audio Layer 3 audio, AES encrypted data, US-ASCII text, Intel x86 opcodes, Basic Latin Unicode, Base64 encoded binary data, C++ source code, fixed length data structures, and image bitmaps, among many others.

Forensic analysts, security researchers, and reverse engineers routinely deal with tasks that require understanding of the various primitive fragment types contained within large binary objects (examples include malware detection, file fragment reassembly, file type identification, binary fragment classification, and the location of regions of interest such as cryptographic keys). Conversely, ignoring the distinctions between diverse primary types is likely to be unhelpful – for example, researchers attempting to develop statistical signatures for a complex container file format, such as a Microsoft Word 2003 document, or an executable format like Windows PE or Linux ELF are less likely to be successful if they treat the file as a single homogeneous object rather than a complex object composed of diverse primitive fragments. We argue that such primitive fragments are more amenable to signature matching, data mining, machine learning, and statistical techniques, and can be analyzed more accurately than conglomerates of such fragments.

The primary contribution of this paper is the presentation and analysis of a detailed, visual taxonomy of primitive binary fragment types. We can not and do not claim that our taxonomy is complete. There are nearly infinite ways to structure, manipulate, and transform binary data that make a complete taxonomy well beyond the scope of a single paper. For example, a digital photograph could be saved using a variety of image formats and compression algorithms, encrypted using one of dozens of possible encryption schemes, and finally en-

coded using one of several methods for transmission across a text-based network protocol. The end result is an exponentially increasing number of possibilities, even in this simple scenario.

To appropriately scope the problem, we chose primitive data types that are commonly encountered. We feel this approach is reasonable. Rather than attempt to catalog every conceivable sequence of transformations and possible outcomes, we instead include only those that are likely to be encountered in practice. However, our taxonomy is extensible, and can be extended as new data types gain prominence.

Note that our definition of fragment types does not include the notion of data types found in common programming languages, such as *char*, *int*, *long*, *float*, and *double*. Whereas primitive binary fragments may consist of these low-level building blocks, our taxonomy does not address such atomic elements explicitly.

Similarly, it is not always possible to say whether a binary fragment that is comprised solely of a certain primitive type contains just one item of that type or is comprised of multiple adjacent items of the same type. In some cases, for example, when header or footer information is available, we can be sure of the boundaries between items. In other cases, without information about the structure of the file, we cannot find out if the data fragment is a concatenation of items of the same data type. For example, a text fragment without visible structure might well be a concatenation of text fragments.

In addition, splitting binary data into minimal building blocks does not always help to analyze the data in a meaningful way. For example, minimal valid text data fragments are the size of one character and can naturally occur in non-text data. The UNIX utility `strings`, which finds printable character sequences in files, by default prints out strings that are at least 4 characters long. Even with this length restriction, the output of `strings` on non-text files is typically full of strings that are clearly not text data. The above considerations led us to define a primitive binary fragment as the longest fragment of binary data that may constitute a single instance of a certain primitive type, whether or not this instance is a concatenation of multiple items of that data type.

This paper is organized as follows. Section 2 places our research in the field of related work. Section 3 presents our taxonomy. Section 4 provides additional analysis, conclusions, and promising directions for future work.

## 2 Related Work

Researchers have studied binary fragments in numerous contexts, including forensic analysis, reverse engineering, fuzzing, visualization, and binary object mapping, among many others.

Areas of study within computer forensics include file type identification, file

carving, and file fragment classification and reassembly. File type identification research seeks to determine the type of file without reliance on the file extension and magic number, the fixed byte sequence often embedded at the start of files to indicate file type. For example, Li used complete and “truncated” files (truncated files were created by extracting the file’s first 20, 200, 500, and 1000 bytes) and 1-gram analysis to identify the file’s format [1]. Karresand used an algorithm based on the rate of change of byte values to classify 51 different file formats [2]. McDaniel used byte frequency analysis and byte frequency cross-correlation analysis to identify files without regard for their metadata [3]. Hall used a sliding window to take 100 measurements each of Shannon Entropy and “compressibility” to identify file types [4].

The areas of file carving and file fragment reassembly are also related to our work. Veenman used statistical techniques to classify disk clusters by file type in order to facilitate file recovery [5]. Erbacher used a sliding window algorithm to visually examine the internal structure of seven different file types, .doc, .exe, .jpg, .pdf, .ppt, .xls, and .zip. Erbacher’s work points to the diverse internal structures contained within complex file formats [6]. Moody also used a sliding window algorithm to study the internal structure of complex files and introduced the idea of base data types, i.e. file formats that can reasonably be considered to consist of the same data type, such as .jpg and .txt [7]. Shanmugasundaram used context models to sequence and reassemble document fragments including system logs, source code, executable code, binary data files, unformatted text, and random text [8]. Richard studied the performance of the Scalpel and Foremost file carvers, which are based on databases of header and footer signatures [9]. Finally, Calhoun studied algorithms that classify .jpg, .gif, .pdf, and .bmp files based on longest common subsequences and Fisher’s linear discriminant [10].

Importantly, Roussev and Garfinkel explicitly mention the need to understand the complex internal structures of popular file types including .pdf, .doc, and .zip. They included a study of 131,000 valid pdf files and clearly illustrated that these files regularly contain a wide variety of internal data objects. They do not however attempt to create a generalized taxonomy [11].

Researchers have also explored the identification of fragments of a single type contained within a larger binary object. For example Shamir located cryptographic keys in disk images by using a 64-byte sliding window algorithm and counting the number of unique bytes [12]. Stolfo used n-gram analysis to detect documents containing malicious software [13].

Reverse engineers are concerned with the internal structures found within files, particularly executable files. Typical tasks include understanding the behavior of a given compiled application by examining its disassembly, and modifying or disabling certain behaviors. Many reverse engineers seek to identify major sections within executables such as .text (machine instructions), .data (initialized data), and .bss (uninitialized data) in Windows PE executables. They also seek to identify major programmatic structures within the disassembly, such as

loops, logical decisions, and jumps.

Advanced fuzzers target low-level structures contained within binary objects, particularly those structures that are provided as input to an application or network service. By modifying these sources of input data in a brute force fashion, fuzzers attempt to identify software vulnerabilities. An excellent overview of the field was provided by Sutton [14].

Little recent work has been done in the visualization of low-level binary data, with one exception. Conti studied the use of a byteplot visualization to view binary objects and created an interactive system for exploration [15, 16]. He did not however develop a taxonomy of primitive fragments. Note that we use his open source *binvis* tool, modified to produced grayscale output, to create the fragment images included in this paper [17].

The common trend in the above work is that each area of research explores the internal structure of binary objects but is hindered by the lack of a comprehensive binary fragment taxonomy. Thus the novelty of our work springs from the creation of a taxonomy of binary fragment types along with accompanying graphical depictions to complement and support future binary data analysis and research.

### 3 Fragment Taxonomy

There are many ways one could organize a taxonomy of binary fragments. We have chosen to build the taxonomy, see Table 1, based on common types of source media and common ways this media may be encoded, encrypted, or compressed. To determine appropriate base media types we studied RFC 2046, Multipurpose Internet Mail Extensions (MIME) Media Types [18]. RFC 2046 provides five discrete top-level media types: text, image, audio, video, and application (i.e. typically executable binary data or data designed to be processed by an application). We also reviewed the registered basic media content types cataloged by the Internet Assigned Numbers Authority, the binary template archive provided by Sweetscape Software, the FILEExt file extension database, and numerous file format specifications and Object Linking and Embedding documents [19, 20, 21].

These reviews were the foundation of the design of the taxonomy. However, one of the challenges when constructing a taxonomy of binary fragments is how to cope with the possibility of the near infinite number of permutations of encoding, compression, and encryption that are possible, at least theoretically. To account for this issue, we have chosen to include generic categories for each of these three transformations. We have also included categories for random number sequences and repeating values (such as the padding used to align a data structure to a page boundary). Finally we have included an “other” category to capture fragment types not included elsewhere in the taxonomy.

**Table 1. Taxonomy of Primitive Binary Fragment Types**

	Major Category	Minor Category	Subcategory	Notes
Binary Fragment	Text	programming language source code	C	Typically encoded in US-ASCII. May alternatively be grouped by compiled and interpreted languages.
			Python	
			JavaScript	
			...	
		written language	English	Typically encoded in US-ASCII, and more recently, Unicode. Further variants include all upper and all lower case characters.
			Russian	
			...	
		markup language	LaTeX	Some markup languages may encapsulate other binary fragment types, such as JavaScript within HTML and binary images in XML. We recommend treating such embedded objects as distinct types.
			HTML	
	XML			
	...			
	Image	uncompressed	bitmap	Images are commonly encoded using $2^n$ bits per pixel, where $n=1, 2, 4, 8, 16, 24, \text{ or } 32$ . The process of generating an image may generate associated data structures, such as image headers and palettes.
		compressed	LZ77 / Huffman	
			JPEG	
	Audio	uncompressed	PCM	Commonly stored in the .wav file format.
		compressed	MPEG-1 Audio Layer 3	MPEG-1 Audio Layer 3 is commonly referred to as mp3. Vorbis is the lossy audio compression commonly paired with the Ogg container format.
			Vorbis	
			...	
	Video	uncompressed	Full Frame	Infrequently used. Stored in the .avi file format.
		compressed	MPEG-4	The video class may include interleaved image and audio information.
			WMV	
			...	
	Application	packed executable	UPX	UPX and ASPack do not list their specific compression algorithm(s), which might be preferable to include in the taxonomy.
			ASPack	
			...	
		machine code	native	16 bit real mode x86, 32 bit protected mode x86...
			virtual	This category could be extended to include processor type and specific modes. An example is Java bytecode.
data structure		fixed length	Examples include some arrays, databases, log files.	
	variable length	Examples include some stacks, heaps, pointer tables, and database files. pcap format packet captures.		
Random	high quality random	atmospheric noise	Atmospheric noise and patterns in lava lamps are just two of many techniques that generate random numbers, with varying degrees of success.	
		lava lamps		
		...		
	pseudo random	Linear Feedback Shift Register	Pseudo random sequences appear random under light scrutiny, but are actually deterministic.	
Mersenne Twister				
Encrypted	symmetric	AES	Encryption algorithms may be further categorized based on their input parameters, such as key length, and possibly the entropy of the plaintext data. Encryption is commonly applied to virtually every category in the taxonomy.	
		Blowfish		
		...		
	asymmetric	RSA		
		ElGamal		
Other Compressed		RLE	Compression is commonly used on virtually every category in the taxonomy, except those that are already compressed or encrypted.	
		LZW		
		...		
Other Encoded		Base64	Encoding is commonly used to convert binary objects to US-ASCII byte values for use on text based network protocols, but is also used in many other ways.	
		uuencode		
			...	
	Repeating Values			A single value of one or more bits that is repeated regularly.
	Other			An unforeseen or newly created fragment type.



Figure 1: Byteplot visualization from a Windows thumbs.db file depicting significant structural differences between header information and compressed image data (seen as white noise).

To illustrate primitive binary fragment examples, we have included images created using the *binvis* tool’s *byteplot* visualization. The byteplot visualization displays byte sequences one byte per pixel. Pixel colors are rendered as a grayscale, where a byte value of zero is black, a byte value of 255 is pure white, and other values are intermediate shades of gray. The top left pixel of the byteplot equates to the first byte of the fragment being displayed. The second byte in the fragment is plotted on the top row in the second column. Subsequent values are plotted in sequence, left to right, until the end of the row is reached, and plotting continues at the leftmost column of the row immediately below. We believe the byteplot technique is a useful means for intuitively interpreting many disparate types of binary structure.

We made other design decisions when constructing the taxonomy, including separating data and metadata. For example, many image file formats include a header structure, which describes the image it contains as well as the values for the image. Figure 1 depicts a byteplot of a portion of a Windows thumbs.db file. Two compressed images (seen as white noise) are visibly distinct from the noticeably structured header information preceding each. Intuitively, we believe that separating data from metadata makes sense, as the structure of each of these regions is often quite different.

Another design decision we faced was whether to assume a theoretical minimum fragment size. We chose not to do so. Fragments should all belong to a fragment class, even if a given fragment’s size might preclude accurate classification using existing techniques. We believe this is an important concept; our lack of ability to accurately classify an unknown binary sample, for example confusing a high entropy encrypted fragment with a random number sequence, does not negate the fact that the fragment is actually encrypted data. Ultimately,

we believe it is useful to seek to categorize all types of binary fragments. In doing so, we may highlight areas meriting future research, such as the automated classification of high entropy fragments. The sections following Table 1 describe categories and subcategories in the taxonomy.

### 3.1 Text

The *text* category exhibits tremendous diversity. Ethnologue, an encyclopedic reference which catalogs the world’s languages, currently reports 6,909 languages still in use [22]. Google offers their search interface in over 120 different languages (including some artificial languages, e.g. Klingon). The latest Unicode standard supports more than 90 different alphabets [23]. We acknowledge this diversity and structured our taxonomy to account for it. The primary subcategories of the text type include: *written language*, *programming language source code*, and *markup language*. Each of these types is commonly encoded using ASCII or Unicode and may be compressed or encrypted as well.

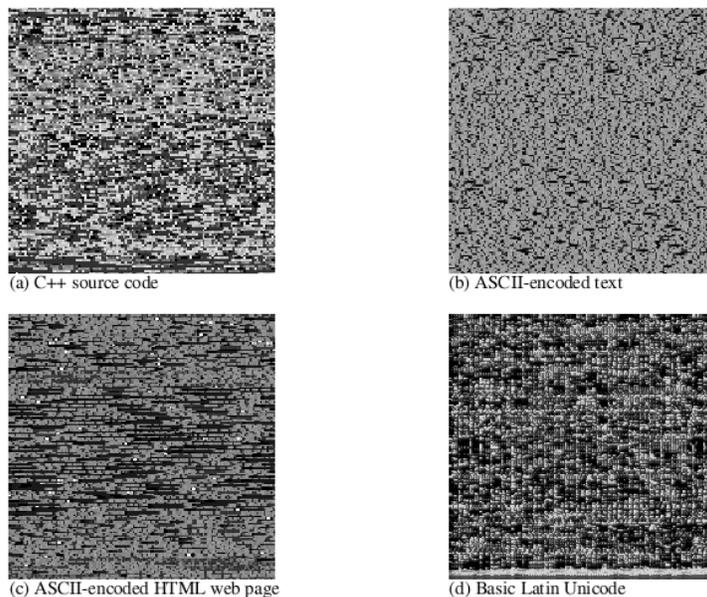


Figure 2: Example Text Primitive Binary Fragment Types

*Programming Language Source Code* - Programming languages, in our definition, are the source code of programs written in common programming languages, such as C++, Java, Perl, JavaScript, and Python. Although similar to written languages, they may exhibit different structural characteristics, such as an increased frequency of certain punctuation characters. See Figure 2(a) for an example of ASCII encoded C++.

*Written Languages* - We define written languages as the written forms of communication used by humans, such as English, French, or Russian, and possibly including artificial languages as well. See Figure 2(b) for an example of ASCII-encoded English text. Note the high density of midrange gray pixels due to printable ASCII byte values in the range of 32 to 126.

*Markup Languages* - Markup languages are used to annotate written languages in order to embed additional information. Examples include HTML, XML, and LaTeX. Figure 2(c) depicts an ASCII-encoded HTML web page containing English text, and Figure 2(d) illustrates Basic Latin Unicode. The vertical banding is caused by the use of 16 bit values, where the most significant 8 bits are zero. Some markup languages allow embedded programming languages, such as JavaScript in an HTML web page. For purposes of classification, we consider these to be two distinct categories, *programming language source code* and *markup language*. Another example is that of a Microsoft Word 2007 document, which may contain a binary encoded image; both the XML document and the image would be distinct primitive binary data types.

### 3.2 Audio, Image, and Video Media

Multimedia fragments are divided into three major categories: *audio*, *image*, and *video*. Whereas there are common techniques for storing multimedia data without compression, compression using a variety of algorithms is common.

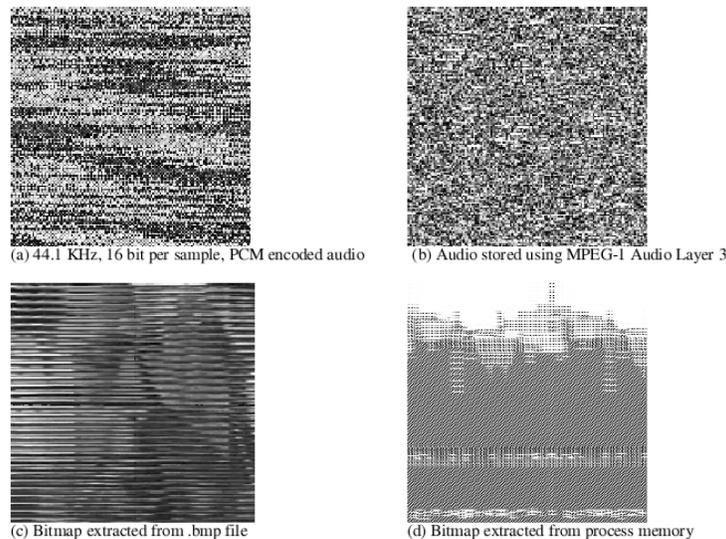


Figure 3: Example Audio and Image Primitive Binary Fragment Types

*Audio* - Digitized audio is typically based on the sampling of a source signal at a fixed number of bits per sample using an encoding algorithm. For

example, the Red Book audio CD standard includes two channels of 16 bit samples taken at 44,100 samples per second using Pulse Coded Modulation (PCM) encoding (see Figure 3(a)). Because the data is not compressed, there is significant structure present. Contrast this with Figure 3(b), which shows the same song stored as MPEG-1 Audio Layer 3 audio. Due to the use of compression, visible structure is noticeably absent.

*Image* - Image data is traditionally stored as a bitmap. Besides width and height, a key feature is color depth, the number of bits assigned to each pixel. Common color depths include 1, 4, 8, 16, 24, and 32 bits per pixel. When visualized using the byteplot display, the fragment often can be recognized as an image (see Figure 3(c)), but may appear distorted due to differences in color depth and image dimensions between the byteplot and the bitmap.<sup>1</sup> Bitmap images are frequently stored using compression algorithms, but may be decompressed at run time for display by an application. Figure 3(d) shows a byteplot of an image first compressed with the Lempel-Ziv-Welch (LZW) algorithm for storage on disk, then decompressed and placed in process memory for display to the user.

*Video* - We debated including video as an explicit category in the taxonomy, as video can be considered just a sequence of still images. In practice, however, there are important differences between such sequences and actual video formats. Video formats interleave audio and video content at the binary level, and, in some cases, fragments from these regions are distinguishable as *audio* and *image* primitive types. However, video formats do include novel characteristics, such as key frames (frames where the complete image is stored) and frames that contain only incremental differences from the key frame.

### 3.3 Application

Application fragments are divided into three main categories: *machine code*, *data structure*, and *packed*.

#### 3.3.1 Machine Code

Machine language instructions are commonly found within executable files, but may be of particular interest when they exist within data files, such as a Microsoft Word document containing executable code. Machine code may vary depending on the target processor as well as the compiler or assembler that generated the code. We further categorize machine code into *native* (i.e. code designed to run directly on a processor) and *virtual* (i.e. code designed as an intermediate, and sometimes cross-platform, representation that is not run directly by a processor). Figure 4(a) shows a machine code fragment extracted

---

<sup>1</sup>Bitmaps are commonly stored “upside down,” i.e., bottom to top in memory. To increase legibility we have flipped Figures 3(c) and 3(d) upright.

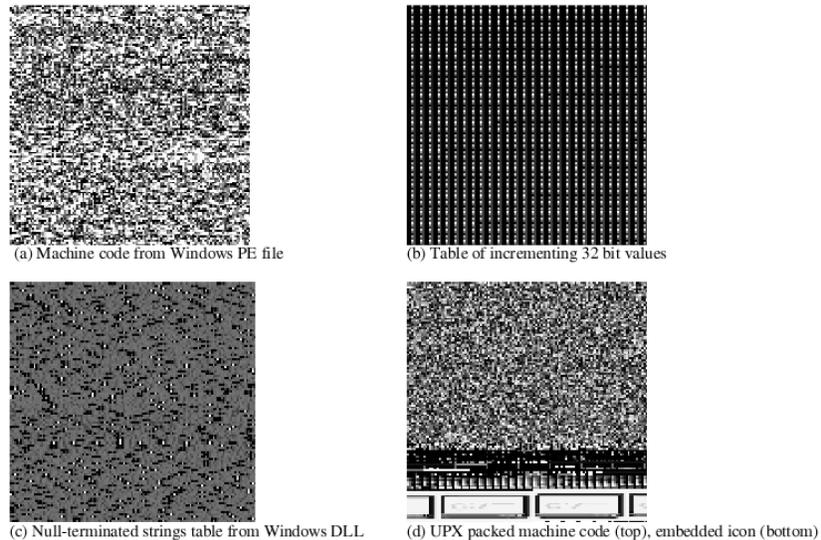


Figure 4: Example Application Primitive Binary Fragment Types

from the `.text` section of a Windows PE file. Due to space constraints, we have not further characterized machine code by its function, such as a logical block or subroutine, nor by processor or compiler, but it may be appropriate to extend the taxonomy to include these types and facilitate increased precision.

### 3.3.2 Data Structures

This category is extremely diverse. We define *data structures* as regions that are used to store data in a binary format. Examples include strings tables, segment tables, symbol tables, pointer tables, arrays, linked lists, stacks, etc. Members of the data structure category may reside in memory, in files, or in network flows. Because of this diversity, we have defined two broad subcategories, *fixed length* and *variable length*, to distinguish between structures composed of fields (or records) of varying or constant size. Figure 4(b) shows a table of incrementing 32 bit values extracted from a Word 2003 document; note the regular structure of the fixed length fields. Figure 4(c) depicts a table of variable length strings extracted from a Windows DLL.

### 3.3.3 Packed

Executable applications, especially malicious software, may also be converted to a packed format, where the file is encrypted, compressed, or otherwise obfuscated. Typically, the packing process is reversed at runtime in order to allow the program to execute. Figure 4(d) shows a UPX packed Windows PE file. Most

of the file (both code and data) now appears as a compressed region, however an embedded icon and other structures are visible at the bottom.

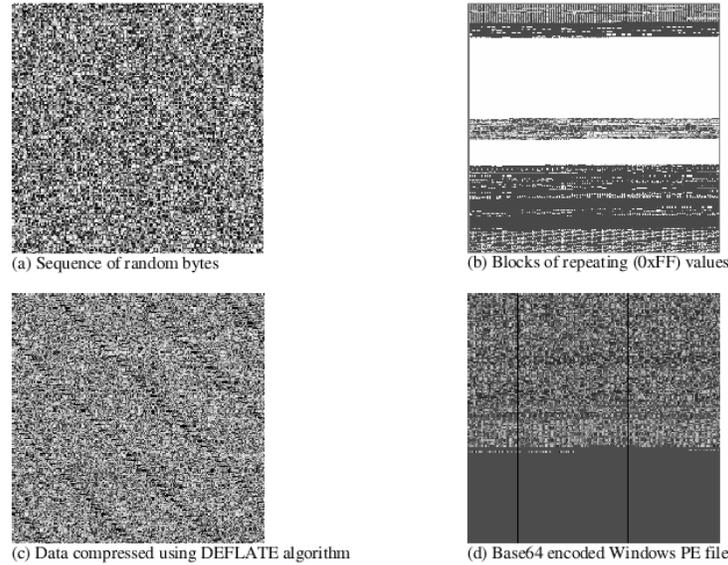


Figure 5: Other Primitive Binary Fragment Types

### 3.4 Random

The *random* primitive fragment type is a sequence of random values and includes two subcategories, *high quality* and *pseudorandom*. High quality random sequences, such as those derived from atmospheric noise, exhibit no discernible pattern. Pseudorandom numbers may seem random, but are actually deterministic in nature. Both subcategories appear visually as white noise (see Figure 5(a)), but given a long enough sequence may be distinguished using statistical techniques. Shorter random sequences, such as cryptographic keys, may also appear in binary objects, but can be difficult to identify, particularly if they are adjacent to other high entropy data.

### 3.5 Repeating Values

Repeating values are short sequences of byte values that are repeated a large number of times. For example, some file formats align data regions to fixed offsets by adding a sequence of null byte values until the desired offset is reached. Such values are easily discerned in the byteplot as solid or highly structured regions; see Figure 5(b).

### 3.6 Encryption, Compression, and Encoding

A useful way to think about binary fragments is in terms of generation and transformation. Most of the previous categories included base content, which was encoded in some fashion (US-ASCII, Unicode, PCM, opcodes mapped to byte values, tables of binary coded decimal, and so forth). These binary representations may be further transformed, sometimes repeatedly, in a variety of ways including encryption, compression, and encoding. We have included some common transformations directly in the taxonomy (e.g. compressed audio and packed executables) but due to the near limitless possibilities we have also included *encrypted*, *other compressed*, and *other encoded* categories. However, some combinations are more likely than others. It is unlikely that one would compress an already compressed file, but compressing data before encryption is often recommended to reduce the possibility of unintended information disclosure.

Encrypted fragments are the result of taking input data and transforming it in some fashion so that only someone with special information may reproduce the input. There are many grades of encryption, but most generate random-looking byte sequences with no apparent structure, as seen in Figure 5(a). Compression algorithms effectively remove redundant information in order to make binary objects smaller. Although they generate fragments that appear visually, and statistically, very similar to encrypted and random fragments, compression algorithms were not designed to “resist” decompression or to serve as a random number generator. For example, Figure 5(c) shows a binary fragment containing compressed data created by the DEFLATE compression algorithm; note the visible structure it contains. Encoding transforms data in one format to another, creating a data fragment of a different type in the process. Encoding is commonly employed when digitizing analog data or converting binary data to facilitate transfer over text-based protocols, and has a wide variety of other uses. Figure 5(d) shows a fragment extracted from a Windows PE file that was Base64 encoded. Base64 operates by converting every three bytes into four printable ASCII characters, thus the underlying structure of the executable is still visible.

The transformation process may significantly alter the nature of the source data. As an example, a compression or encryption program may operate on a single source data file, or it may take hundreds of disparate files and produce a single output object, such as a ZIP archive or encrypted drive image. For future extensions to the taxonomy, we believe it is important to consider, as precisely as possible, how such transformations commonly take place, by taking into account the primitive type (or types) of the source data, the specific algorithm (including its exact input parameters, such as key length) as well as the specific implementation of that algorithm.

### 3.7 Other

Finally, for completeness, we explicitly include the *other* top-level category to allow for unforeseen primitive binary fragment types that do not otherwise fit in an existing category. However, additional *other* categories may also be added to subcategories, as needed.

## 4 Conclusions and Future Work

There is a tremendous amount of variety in binary fragments, and it is natural to want to classify it in order to support research and analysis. Visualization of binary objects further illustrates this diversity, highlights the fact that many of the simplest appearing objects (such as the .bmp file format) are actually composed of differing primitive types, and provides insight into structure that is difficult to infer from purely numeric or textual analysis.

The primary contribution of this paper, a taxonomy of primitive binary fragments, categorizes instances using a descriptive approach. Whereas there is a near infinite number of possible fragment types, we have chosen to focus on fragment types that are commonly used in practice. However, the taxonomy is extensible and can be updated as needed to include new fragment types, both found in the wild, and those that are purely theoretical. We believe it is important to think more in terms of specific algorithms used for encryption, compression, and encoding (along with the possible variants of their input parameters) and less about file formats. Many encryption, compression, and encoding implementations (and file formats) provide numerous usage options. Each of these permutations combined with varying classes of input data generates differing resultant binary fragments types. Importantly, there is a distinction between existence of a taxonomy and our ability to classify a given fragment without a priori knowledge. For example, there are many ways to encrypt or compress virtually any type of binary data, resulting in a high entropy object. Accurately classifying fragments, particularly small fragments of these high entropy objects, is an open problem. However, our ability to classify these fragments notwithstanding, such fragments are very commonly encountered and merit representation in the taxonomy, even if just to serve as a guide for the future work of researchers.

For future work, we intend to study the classification of high entropy fragments. We also suggest further study into generalized binary fragment classification techniques, focusing on classifying as broad a range as possible of primitive fragment types when conducting experiments. We also recommend considering large binary objects to be heterogeneous structures composed of primitive binary fragments of different types, rather than assuming, by default, that such objects are homogenous. We believe that the technique of breaking down large

binary objects into simpler parts, will be useful for fuzzing, file carving, reverse engineering, malware analysis, file type identification, and other forensic and security analysis tasks, such as the low-level analysis of process memory and file system images. Finally, we believe the taxonomy provides opportunity for several important extensions, including in-depth exploration of encryption, encoding, and compression algorithm variations.

Finally, in our taxonomy we did not assume the presence of an adversary who deliberately attempts to make one binary fragment appear to be another. This is entirely possible. For example, Erickson developed the disassembler program, which converts executable machine code to a printable ASCII string [24]. This year, Mason et al demonstrated the feasibility of creating executable machine code using only opcodes that appear as English-like sentences when viewed as printable ASCII [25]. Our focus in this paper was on establishing categories that researchers are likely to encounter. As new dissemblance techniques are developed, we believe they should be considered for inclusion in a future taxonomy of binary fragment types.

## Disclaimers

*The views expressed in this article are those of the authors and do not reflect the official policy or position of the United States Military Academy, the Department of the Army, the Department of Defense or the U.S. Government.*

## References

- [1] W. Li, K. Wang, S. Stolfo, and B. Herzog. Fileprints: Identifying File Types by n-gram Analysis. IEEE Information Assurance Workshop, 2005.
- [2] M. Karresand and N. Shahmehri. File Type Identification of Data Fragments by Their Binary Structure. IEEE Information Assurance Workshop, 2006.
- [3] M. McDaniel and M. Heydari. Content Based File Type Detection Algorithms. Hawaii International Conference on System Sciences, 2003.
- [4] G. Hall and W. Davis. Sliding Window Measurement for File Type Identification. ManTech Cyber Solutions International White Paper.
- [5] C. Veenman. Statistical Disk Cluster Classification for File Carving. Symposium on Information Assurance and Security, 2007.
- [6] R. Erbacher and J. Mulholland. Identification and Localization of Data Types within Large-Scale File Systems. Systematic Approaches to Digital Forensic Engineering, 2007.

- [7] S. Moody and R. Erbacher. SÁDI - Statistical Analysis for Data Type Identification. Systematic Approaches to Digital Forensic Engineering, 2008.
- [8] K. Shanmugasundaram and N. Memon. Automatic Reassembly of Document Fragments via Context Based Statistical Models. Annual Computer Security Applications Conference, 2003.
- [9] G. Richard and V. Roussev. Scalpel: A Frugal and High-Performance File Carver. Digital Forensics Research Workshop, 2005.
- [10] W. Calhoun and D. Coles. Predicting the Types of File Fragments. Digital Forensics Research Conference, 2008.
- [11] V. Roussev and S. Garfinkel. File Fragment Classification – The Case for Specialized Approaches. Systematic Approaches to Digital Forensics Engineering, 2009.
- [12] A. Shamir and N. van Someren. Playing Hide and Seek With Stored Keys. International Conference on Financial Cryptography, 1999.
- [13] S. Stolfo, K. Wang, and W. Li. Fileprint Analysis for Malware Detection. WORM, 2005.
- [14] M. Sutton, A. Greene, and P. Amini. Fuzzing: Brute Force Vulnerability Discovery. Addison Wesley, 2007.
- [15] G. Conti, E. Dean, M. Sinda, and B. Sangster. Visual Reverse Engineering of Binary and Data Files. Workshop on Visualization for Computer Security, 2008.
- [16] G. Conti and E. Dean. Visual Forensic Analysis and Reverse Engineering of Binary Data. Black Hat USA, 2008.
- [17] BinVis tool download. <http://www.rumint.org/software/danglybytes/db.zip>.
- [18] N. Freed and N. Borenstein. RFC2046 – Multipurpose Internet Mail Extensions (MIME) Part Two: Media Types. Internet Engineering Task Force, 1996.
- [19] Internet Assigned Numbers Authority. MIME Media Types. [www.iana.org](http://www.iana.org), 6 March 2007.
- [20] “010 Editor – Binary Template Archive”. SweetScape Software, [www.sweetscape.com](http://www.sweetscape.com).
- [21] FILEExt. [filext.com](http://filext.com), 8 December 2009.
- [22] Ethnologue: Languages of the World, 16th Edition. Sil International, 2009.
- [23] The Unicode Consortium. The Unicode Standard, Version 5.2.0, 2009.

- [24] J. Erickson. Hacking: the Art of Exploitation, 2nd Edition. No Starch Press, 2008.
- [25] J. Maxon, S. Small, F. Monrose, and G. MacManus. English Shellcode. ACM Conference on Computer and Communications Security, 2009.