# Visual Reverse Engineering of Binary and Data Files

Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster

Department of Electrical Engineering and Computer Science
United States Military Academy
West Point, New York
{gregory.conti,erik.dean,matthew.sinda,
benjamin.sangster}@usma.edu

**Abstract.** The analysis of computer files poses a difficult problem for security researchers seeking to detect and analyze malicious content, software developers stress testing file formats for their products, and for other researchers seeking to understand the behavior and structure of undocumented file formats. Traditional tools, including hex editors, disassemblers and debuggers, while powerful, constrain analysis to primarily text based approaches. In this paper, we present design principles for file analysis which support meaningful investigation when there is little or no knowledge of the underlying file format, but are flexible enough to allow integration of additional semantic information, when available. We also present results from the implementation of a visual reverse engineering system based on our analysis. We validate the efficacy of both our analysis and our system with case studies depicting analysis use cases where a hex editor would be of limited value. Our results indicate that visual approaches help analysts rapidly identify files, analyze unfamiliar file structures, and gain insights that inform and complement the current suite of tools currently in use.

## 1 Introduction

Individual files are a fundamental component of today's computing paradigm as well as one of today's biggest threat vectors. With the advent of effective network security devices based upon firewalls, intrusion detection systems and similar security applications, attackers are moving away from network protocol attacks and toward attacking applications themselves. This transition is problematic because firewalls must pass some traffic in order to provide services to their users, particularly web and email. It is through these services that users send, receive, upload and download files, sometimes as email attachments, web downloads, or more worrisome, surreptitiously through encrypted channels such as HTTPS or SSH. The problem is worsened by the rapid evolution of file-based attacks that exploit vulnerabilities in parsing by applications and common software libraries, as well as by the attacker's use of packers which obfuscate the contents of files.

Legitimate files function as either stand alone executable programs or as data to be used by other applications, such as word processors, text editors or graphics programs.

J.R. Goodall, G. Conti, and K.-L. Ma (Eds.): VizSec 2008, LNCS 5210, pp. 1–17, 2008.

Executable files are executed by the operating system, whereas, data files are loaded by applications. In both cases the operating system and application assume some knowledge of the file's underlying format and structure in order to operate on it successfully. In these legitimate cases, the user of the file only sees the interpreted version of the file as determined by the application or operating system. The underlying structure of the file is hidden. It is through this hidden nature that both attackers and legitimate programmers attempt to place security mechanisms. While file extensions and magic numbers serve the purposes of legitimate use, security analysts are forced to dig much deeper and face the frequent task of exploring the raw structure of files to detect modifications, determine file type, determine what a file does, determine authorship, create virus signatures, and detect code evolution, among other tasks. Further hampering analysis is that the file may be corrupted, obfuscated, or encrypted to slow analysis. In some cases, files contain code designed to crash analytic tools or function differently if used in a virtual machine. Sometimes files contain largely legitimate data with a small fraction containing a malicious aspect and others are dedicated malicious applications.

At their heart, files are just binary objects whose meaning is based on the applications or the operating systems that use them. However, in some cases little will be known about a given file. We therefore view the problem of reverse engineering binary files as having two levels, a context independent level where we assume the analyst has little or no knowledge about the structure and purpose of a given file, and analysis must occur initially in a context-independent manner. At the second level, the analyst knows some information, such as an expected file format, and can make informed assumptions as they analyze the file. The difference between these two levels is evident in the current suite of tools for reverse engineering files. At the context independent level, the analyst employs general purpose tools such as hex editors and byte frequency analysis to gain insight. At the semantic level, analyst tools are crafted specifically to a given file format or family of related formats, such as dissemblers and debuggers, leading to very precise insights.

Unfortunately, the vast majority of best of breed tools for reverse engineering of files are strictly text based and provide only a tiny viewport into the file under analysis. Visualization is underutilized in reverse engineering and bears great promise for assisting analysts in their work and augmenting their existing tool suite. In this paper we present an analysis of user tasks that is useful for the design of visualization systems, and employ this analysis to implement a system which combines proven text based techniques with innovative visualization approaches in order to augment the analyst and complement their tool suite for a number of essential tasks. We validate the efficacy of these contributions through several case studies. It is important to note that we are addressing the problem of reverse engineering of binary file formats across the entire range of possibilities and not focusing specifically on the special case of reverse engineering of executable files.

This paper is organized as follows. Section 2 places our research in the field of related work. Section 3 contains a user-level requirements analysis that we used to guide our development. Section 4 presents our system design and implementation. Sections 5 demonstrate the utility of our approach through case studies. Section 6 presents our conclusions and suggested directions for future work.

## 2 Related Work

The most commonly employed tool for reverse engineering of files and file formats is a hex editor, which typically displays files in hexadecimal and ASCII formats and assumes no knowledge of the underlying file structure. Traditional hex editors offer basic functionality, but modern hex editors contain advanced features including the ability to view, search, and convert between hex, ASCII, Unicode, decimal and floating point data types, among others. Such tools also include the ability to encrypt and decrypt, calculate checksums, encode and decode, calculate computer hashes, and compress and decompress blocks of data within a file. Navigation is straightforward, including the use of scroll bars and the ability to jump to a given offset within the file, but also includes the ability to place navigation labels at user specified locations. The Hiew hex editor is noteworthy because it integrates an assembler and disassembler. Other sophisticated editors, such as WinHex, also include the ability to edit memory, create a byte frequency histogram of both the entire file and a user selected region, and easily link to helper programs such as web browsers and media viewers. The 010 editor also includes binary templates which parse popular binary file formats into their associated variables and data structures. Hex editors are the definitive tool for text-based analysis of binary files and include powerful computation capabilities, even scripting, but lack the ability to provide big picture context, a significant problem due to the complexity of even moderately sized files.

Beyond hex editors, there are other general purpose tools for reverse engineering files of both executable and data formats, including the command line *strings* utility which outputs contiguous runs of printable ASCII. Similarly, the command line *diff* command, the *fc* (file compare command) and related variants allow the comparison of text and binary files in order to locate changes. Recently more powerful GUI-based tools have emerged, such as Compare It!, ExamDiff, Guiffy, Merge, Meld, and WindDiff which allow side-by-side comparison of files. Common attributes of this class of tools include side-by-side views in text or hexadecimal representations, scripting, directory comparison, reporting tools, syntax highlighting, the ability to detect changes (diffing) and to merge files. Compare It! and Merge also provide the ability to compare images. As is the case with hex editors, command line and GUI-based file comparison tools are text based and lack the ability to provide big-picture context that visualization can provide. However, there are several notable exceptions. The first is Visual Insights Difference Viewer, which combines the two-pane textual view with a two column graphical view similar to Eick's Seesoft technique to provide context when comparing two files, but this tool is apparently no longer available. The 010 editor also uses a similar combined text and Seesoft-like view technique. Nwdiff is another interesting approach. It plots pixels based on the actual bit values contained within a pair of files and uses four graphical panes. Two of the panes show the raw structure of the files; the other two panes graphically show similarities and differences between the files by using a logical or and xor. BinaryViewer and RUMINT's binary rainfall view [1] use similar bit-level views, to view binary files and network packets, respectively. Another, albeit rudimentary, technique for visualizing binary files is the *raw2tiff* program. raw2tiff converts raw byte streams to the tiff image format. Designed for image file conversion,

raw2tiff produces similar results by converting a binary file to a tiff image. We believe these bit and byte-level approaches bear great promise in helping analyze binary files; we will discuss these techniques later in the paper.

Other interesting approaches for visualizing binary files include Kaminsky's use of dot plot [2] patterns to explore self-similarity in binary files as well as his use of context free grammars to highlight hex dumps [3]. We have incorporated Kaminsky's dot plot technique into our system as we will discuss later in the paper.

As part of considering tools for analyzing individual files, or pairs of files as in the case of diffing, it is also useful to examine existing tools for visually displaying complete file systems. These include Firelight which uses concentric segmented rings, SequoiaView and GdMap which uses squarified treemaps, KDirStat, Baobab and WinDirStat combine a treemap with a textual tabular view in the same display window.

The file fuzzing community also employs tools designed to manipulate binary files in order to identify vulnerabilities in application parsing algorithms. Popular examples include SPIKEFILE for Linux and FILEFUZZ for Windows. Such tools are primarily text based, but would benefit from the interactive visualization techniques we present later in the paper to assist in identifying promising locations in files that users are attempting to fuzz.

So far, we've discussed related work regarding the analysis of binary files in general, however it is worth specifically discussing the special case of reverse code engineering (RCE) which focuses on the analysis of executable files. There are several ways to approach RCE. The first is static analysis, the examination of a file's contents without executing it. The second is dynamic analysis which studies the internal operation of the code as it is executing. Another approach is to execute the program and study how it interacts with the operating system and network. The primary tools are hex editors, dissassemblers with IDA Pro being the best of breed, debuggers including tools like OllyDbG and SoftIce, and decompilers. It is important to note that IDA Pro allows user created plug-ins as well as scripting, and as a result, there is an active developer community surrounding IDA Pro, such as OpenRCE and IDA Palace. The wide range of *binutils* is often employed, including *objdump* which displays information about object files and *readelf* which displays information about ELF format object files. Tools from the Sysinternal's tool suite augment debuggers and decompilers, by allowing fine grained monitoring of an application's interaction with the operating system as it executes.

There has been a limited amount of work in the visualization of executable files. Yoo used self-organizing maps to detect viruses [4]. The most current work is found in the Zynamics' BinDiff, BinNavi, and VxClass tools which utilize directed graphs. Graph-based techniques have demonstrated great utility in analyzing malicious software [5, 6, 7, 8] including diffing of executable files [9, 10].

The novelty of our work springs from our analysis of reverse engineering tasks, novel visualizations, and our system for analyzing binary files. While there has been a great deal of work on text-based analysis of files, there is only limited work of visualization of files themselves, primarily only executable files, at both the context-independent and context-dependent levels.

## 3 Requirements Analysis

Reverse engineering of file formats is both an art and a science. As such, many analysts develop their own personal approaches to reverse engineering. These ill-defined individual approaches are a significant challenge when attempting to design systems that facilitate the work of many different users. To overcome this shortcoming and ground our work in real-world user requirements we analyzed five different approaches found in: *Fuzzing – Brute Force Vulnerability Discovery* by Sutton, Greene and Amini, Wikibooks' *Reverse Engineering*, *Hacker Disassembling Uncovered* by Kaspersky, *Hacking – The Art of Exploitation* by Erickson, and *Hack Proofing Your Network* by Russell et al. While each text provided unique approaches to reverse engineering, we did find significant commonality. In addition, to supplement our

**Table 1.** Scenarios which require low-level analysis of files

| Goal | Examples |
|---|---|
| Analyze undocumented file format | - Classify basic purpose of file<br>- Understand structure of file format<br>- Understand behavior of creating application<br>- Identify algorithms used for compression, encoding and encryption within file |
| Audit files for vulnerabilities | - Locate structures for targeted fuzzing<br>- Identify vulnerable code structures<br>- Locate caves (empty regions within file) |
| Compare files (Diffing) | - Create signature of malware variant<br>- Determine purpose of a patch<br>- Track evolution of code between file versions |
| Cracking[1] | - Break copy protection<br>- Alter player resources in game (e.g. gold pieces) |
| Cryptanalysis | - Validate protocol or algorithm operation<br>- Confirm encryption occurred<br>- Gain insight into encryption algorithm<br>- Find key or password<br>- Analyze files for steganographic content |
| Forensic analysis | - Determine authorship<br>- Locate and extract metadata<br>- Locate and extract hidden content<br>- Identify compiler, language or application used to create file |
| Identify unknown file format | - Precisely determine application which created file<br>- Classify type of application which created file |
| Malware analysis | - Reverse engineer file's function<br>- Create antivirus or IDS signature<br>- Locate malicious code within file |
| Reporting | - Create analyst annotated reports<br>- Share analytic results with analysts, management, and customers. |

[1] While we don't support file cracking, it is nonetheless a scenario which requires low-level analysis of files and was repeatedly mentioned in our literature review.

**Table 2.** Representative reverse engineering high-level tasks

| Category | Task |
|---|---|
| Analyze | -Identify and analyze non-standard file formats and algorithms<br>-Understand, annotate and document the file's structure, including header/footer and block/record/field formats<br>-Test and evaluate hypothesis as to the meaning of the data and file format |
| Calculate | -Perform decimal, hexadecimal, and binary calculations<br>-Encrypt and decrypt, encode and compress blocks of values within a file, calculate checksums |
| Compare | -Compare two or more files and precisely locate differences. |
| Explore | -Understand the big picture context of a file's structure<br>-Identify major structures within a file |
| Filter | -Remove undesired content |
| Identify | -Identify which algorithms and libraries were used<br>-Identify and analyze regions containing executable code and data<br>-Identify in-file references to data |
| Locate | -Locate regions that have been encoded, compressed or encrypted<br>-Locate free/slack space |
| Modify | -Edit values within files<br>-Fill regions with desired values<br>-Load and save text and binary files |
| Navigate | -Easily navigate to regions within the file |
| Report | -Generate report of analysis |
| Search | -Locate specific values or sequences of values, including those in hex, floating point, binary, decimal, ASCII and Unicode representations. |
| Semantics | -Correctly parse binary file formats<br>-Apply external knowledge of file structure and format to gain additional insight |
| View | -View files and regions in native viewers/formats, including assembly<br>-View/convert values in native format/encodings/datatypes/byte orders (e.g. 2 and 4 byte integers, floats, strings, Unicode, real and string, signed and unsigned) |

literature review of user requirements we conducted semi-structured interviews of 12 intermediate and advanced security analysts from the academic, industry and hacker communities, conducting the interviews primarily at the RSA 2008 and Shmoocon 2008 Conferences. From the results of our literature review and interviews we compiled scenarios which require low-level analysis of files, see Table 1, and categorized specific tasks analysts face when seeking to reverse engineer data and executable files, see Table 2.

When faced with an unfamiliar file, the analyst will also employ common command line utilities such as *strings*, which looks for sequences of ASCII characters contained within binary files, *file*, which attempts to identify a file's format. The next step is often to load the file in a hex editor and scroll the textual display looking for regions of interest. In the case of an executable file, the analyst will likely run the file and observe its interactions with the underlying operating system and network using tools which monitor system calls, network activities, file accesses, and registry changes. The analyst employs debuggers and disassemblers to understand the code in operation.

When the file is untrusted, analysis will almost certainly be conducted on an isolated malware analysis workstation, usually in a virtual machine environment to provide additional isolation. Depending on the analyst's objective, the machine may have network connectivity. Reverse engineering of both executable and data files is, in many ways, an adversarial relationship. For example, there is an increasing trend by malware authors to attempt to detect virtual machine environments and behave in an unexpected manner, such as crashing the debugger, to frustrate analysis. File extensions and other metadata, particularly in forensic analysis, are not fully trusted by the analyst. The designers of the file format will often go to great lengths to obfuscate file contents by using encryption, packing, or obfuscated coding techniques. There are legal issues as well. In some cases attempting to reverse engineer file formats, particularly when encrypted or deliberately obfuscated, can be considered a violation of intellectual property rights.

## 4 System Design and Implementation

There are a number of situations that necessitate low-level analysis of files and file formats, but they fall into two main categories: context independent analysis when little is known about a given file's format and semantic analysis where the analyst knows some information about the structure of the file. For our work we have chosen to design our system to facilitate context independent analysis where a hex editor and command line tools, such as *strings*, would be used. These include analyzing undocumented file formats, auditing files for fuzzing opportunities, and forensic analysis. More specifically, we designed our system to aid rapid analysis, provide big picture context, facilitate navigation, and assist identification of internal structures contained within files as we believe these are promising areas for visual support. We leave semantic analysis and other user tasks for future work. That being said, an understanding of file formats in general, is critically important even in the context independent case.

Visualization of files allows the analyst to see structures within files and it is useful to study file formatting techniques. File formats come in a myriad of different types, from extremely simple to highly complex. While it is impossible to determine the exact number of different file formats, the popular FILExt database of file extensions currently tracks 24,048 different types and Wotsit, a leading file and data format website, provides information on 1,030 different publicly available and closed file formats. The end result is an environment with wide variety of commonly employed techniques as well as the likely possibility of unique file formats written by individual authors.

Common file structuring techniques include embedding metadata (e.g. serial numbers and magic numbers), storing fixed and variable length records, compressing and encrypting regions within a file, embedding images, as well as various approaches to storing and encoding strings, integers and floating point values.

Analysis of files needn't be constrained to data contained within the file itself, but could incorporate external information stored by the operating system, such as file name, file size, date of creation, date of modification. Similarly, a visualization system may employ a wide range of statistical techniques to add meaning to the visualizations, assist filtering, and aid navigation within the file, such as calculating the frequency of

bytes, calculating entropy, and performing n-gram analysis. Such calculations could occur across the entire file, or be constrained to a given window selected by an algorithm or end user.

We implemented our system using C# in Microsoft Visual Studio .NET 2005. We chose this environment because C# is a robust and comprehensive language and because of Visual Studio's strength in rapid GUI development. All testing was done on a commodity PC (Dual Core AMD 2500, 1GB RAM, Windows XP). For future work, we plan to explore implementing the system, including all interactive GUI elements, in a platform independent language such as Java, Perl or Python. As malware analysis often occurs inside virtual machines, it is also important that future versions perform well in such an environment.

### 4.1 System Design Goals

Given our analysis of user requirements and the environment in which users operate, we created the following design goals to guide our development. These goals, are just that, goals. Later sections in the paper will demonstrate which we accomplished in our current system implementation [11].

- *Useful* – Allow user to gain useful insight about the file, including big picture structure, embedded objects, obfuscated or hidden data, malicious content, and embedded metadata.
- *Ease of use* – The application should be easy to install, understand, and operate.
- *Extensible* – A small group of developers cannot compete with the ingenuity of an entire user base. An extensible design allows the open source community to develop plug-ins.
- *Incorporate best practices* – Don't rediscover fire. Create a design that can incorporate best practices from existing tools.
- *Open source* – In order to gain trust of our security conscious user base, releasing the source code helps increase adoption.
- *Context independent analysis* – Provide valuable insight into binary files, even if the underlying file format is unknown.
- *Semantic file analysis* – Incorporate relevant semantic information into the visualizations when file format is known or suspected.
- *Multiple coordinated views* – Provide useful windows into the file that complement existing textual tools.
- *Attack resistant* – Design the tool with the understanding that it may be attacked by a malicious file under analysis.
- *Platform independence* – The ideal system should function when used on all major operating systems employed by users.

### 4.2 Visualization Design

Our system incorporated both textual and graphical visualization techniques in order to combine the functionality of command line tools and best practices from hex editors with insightful visualizations. In its current implementation, the system incorporates two

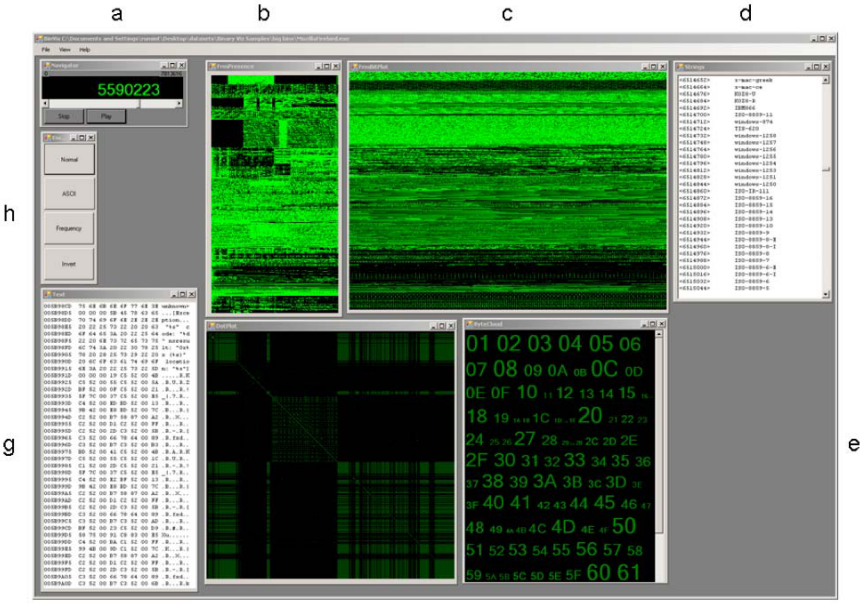



**Fig. 1.** System screenshot depicting each of the visualization techniques

textual views. The first view is the canonical hex/ASCII view commonly employed by hex editors and hex dump command line utilities, see Figure 1(g). While we only included a hex viewer window, a key idea is that a hex editor can be incorporated *in its entirety* into the design we propose. The second textual view displays ASCII strings contained within the file, see Figure 1(d). Both displays include the offset of the data displayed. The system includes a number of graphical displays which are described in the following sections. It is important to note that we view our textual and graphical displays as a starting point. Our ultimate aim is to create an extensible architecture that would inspire end users to create and share additional visualizations.

### 4.3 Byteview Visualization

The system includes four graphical displays[2], the first is a byte plot visualization, see Figure 1(c), which maps each byte in the file to a pixel on the display. The first byte in the file is located in the top left corner, coordinate (1,1), the next byte is displayed at position (2,1). The byteview is 640x480 resolution, so each row can display 640 bytes. When the end of the line is reached, plotting begins at the next line below. At 640x480 resolution, the byteview visualization can display 307,200 bytes. Thus byte 307,200 will be displayed at coordinate (640,480). The color of each pixel maps to the value of the byte displayed, where a byte value of 00 would be black and FF would be bright green. We chose 640x480 resolution because its relatively small size would facilitate rapid drawing. In addition we believe choosing resolutions in multiples of 32

[2] The following sections describe three displays, the fourth, the Byte Map display (Figure 1(f)) alters font size based on byte frequency, but is still under development.

is important when analyzing files written for 32-bit PCs as many structures contained within files are multiples of 32. When testing performance we found that the display could be updated in 0.03 seconds, leaving open the possibility of creating byteview visualizations at greater resolutions while still providing a responsive interface.[3]

### 4.4 Byte Presence Visualization

The byte presence visualization, see Figure 1(b), consists of 256 columns. Each row displays the presence and absence of byte values within a given window in the file being examined. This visualization is designed to act in concert with the byteview display and each of the 480 rows from the byteview visualization is displayed as a corresponding row in the byte presence display. For example, if the eighth row of the byteview contains only byte values in the printable ASCII range (i.e. 32-127) the eighth row of the byte presence visualization will have pixels in the $32^{nd}$ through $127^{th}$ columns illuminated. By designing these two visualizations to act in concert, an analyst is able to perform side-by-side comparison of a given region of interest. The byte presence visualization is particularly useful for identifying regions of text contained within a file (seen as vertical bars in columns 32-127), for detecting regularly changing byte values in the file (seen as diagonal lines, where the slope equals the direction and rate of change), for identifying regions of compression or encryption (seen as a nearly complete horizontal line), as well as for detecting the set of characters used by an encoding scheme, such as *uuencoding* which uses a subset of printable ASCII characters. Our current implementation indicates the presence or absence of a given byte value, a possible future enhancement is to use color to highlight bytes based on frequency or entropy.

### 4.5 Dot Plot Visualization

The dot plot visualization, see Figure 1(e) is a powerful visualization technique used by bioinformatics researchers to measure self-similarity. Kaminsky demonstrated that the technique is also useful for the analysis of binary data, particularly for visually detecting repeated sequences of bytes contained within a file [12]. Due to the promise of Kaminsky's results[4], we included a dotplot visualization in our implementation. Kaminsky's dot plot works by creating a matrix out of a sequence of bytes from the file. Similarly, in our system we used the file under analysis for labeling both the horizontal and vertical axes. Pixels in the display are illuminated at all locations where the horizontal and vertical axes values are identical. Note that the algorithm may also be used to compare two different byte sequences, such as two different files, and visually indicate each difference. In this case, one axis is labeled with the first file and the other axis is labeled with the second file. The dot plot algorithm is $O(n^2)$, thus plotting a full 1MB file would create a 1TB image, beyond the power of desktop workstations. To overcome this shortcoming, we implemented a 500x500 dot plot as

[3] We were able to achieve this level of performance by avoiding C#'s GetPixel and SetPixel methods and directly accessing image memory, see http://www.bobpowell.net/lockingbits.htm for more information.

[4] Note that Kaminsky's approach was not interactive. He generated extremely large dot plots of entire files. Our approach is interactive.

a tradeoff between functionality and processing requirements. As the user navigates the file, the dot plot is redrawn using a 500 byte window from the current offset onward. A full description of the dot plot is beyond the scope of this paper, for more information see Helfman as an introduction [15].

### 4.6 Navigation and Interaction Design

Navigation in our system is designed to be simple and intuitive, applying multiple coordinated visualization windows, both graphical and textual. It is accomplished via a small VCR-like display, Figure 1(a). The analyst may navigate to a new location by adjusting a horizontal scroll bar or by clicking the play/stop buttons. The play button causes each of the graphical displays to scroll automatically, allowing the user to rapidly scan large files. The numeric display on the VCR depicts the current offset in the file. The user may bring up specific textual detail by clicking the byteplot visualization. As a future enhancement, we plan to add similar functionality to all graphical visualizations. Similar navigation could be added to the *strings* and other textual displays by allowing the user to click on a textual item and each of the other displays would automatically change to reflect the new offset.

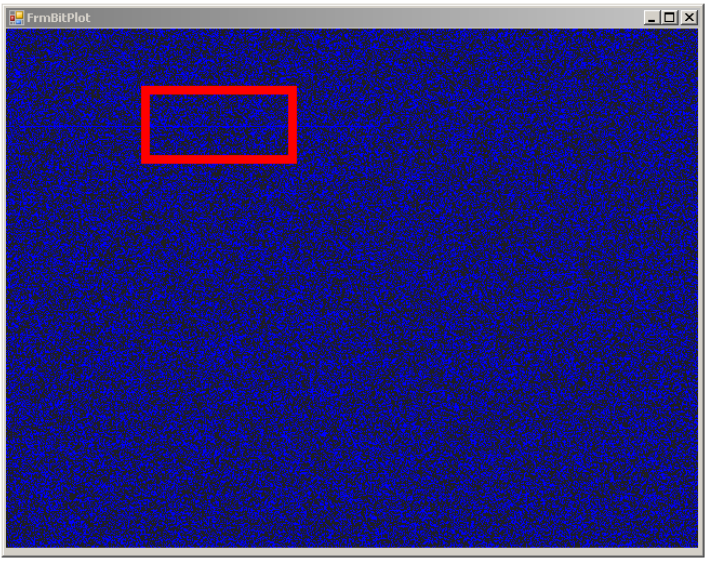
We use color coding to highlight specific attributes of the file under examination. In our system, color coding is accomplished using a small toolbar consisting of four buttons, see Figure 1(h). Our long-term intent is to allow individual analysts to create coloring rules of their own choosing and influence each display, but in our current implementation, we hard coded four, one per button, and they only affect the byte view visualization. These rules include highlighting printable ASCII characters (blue for bytes in the printable ASCII range and gray for all others), displaying byte frequency (blue/low frequency to red/high frequency), inverting the color scheme of the display, and finally a rule for the default color scheme.

## 5 Case Studies

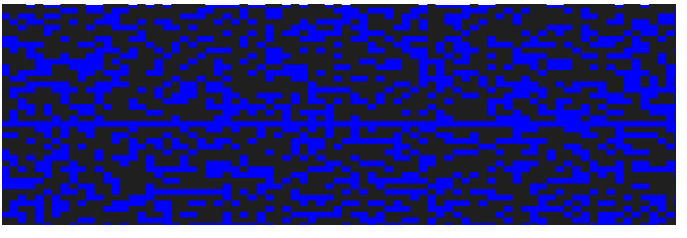
In this section we demonstrate the utility of our approach by using the system in four scenarios of increasing complexity: locating a hidden message contained within an MP3 file, identifying fixed and variable length records contained in database files, reverse engineering of a Microsoft Word document, and analyzing process memory of a Firefox browser running under Windows XP.

### 5.1 Hidden Message in an MP3 File

This example was inspired by Johnny Long's "Death of 1,000 Cuts" talk at the Defcon 14 hacker conference. Long demonstrated numerous ways to hide information from forensic investigators by creatively placing digital information in obscure locations. He showed that it is possible to hide a textual message inside an MP3 audio file by manually altering the file with a hex editor. The file could then be stored on an MP3 player. Short messages, on the order of several hundred bytes or less, cause little to no discernable distortion in the audio playback. Using this technique, we inserted a 331 byte message composed of a sequence of ASCII values in a 3.2MB, 3.5 minute




(a) Full screen display of file



(b) Detail of message region

**Fig. 2.** Byte view visualization of an MP3 containing an ASCII message (a), the detail image (b) more clearly illustrates the message as a horizontal line

song, see Figure 2. Because we were searching for ASCII characters, we turned on the ASCII encoding filter to help highlight sequences of characters. As you examine the figure, note that the remainder of the MP3 file format appears as visual noise, due to the format's compression algorithm, which allows the regularity of the embedded message to become noticeable. By pointing to the suspected message and clicking, the analyst can learn the offset and view the message in the text view window.

While this is a straight forward example, it does illustrate a key aspect of the byte view visualization technique - internal structure is readily apparent. In this case, the deviations from apparent randomness due to compression are easily discerned. It is important to note that the ASCII encoding filter was not required to detect the region, but we chose to use the filter in this example to demonstrate one possible use case. Other means of encoding alphanumeric characters are also discernible using this visualization. For example, alphanumeric characters from the Basic Latin Unicode Set

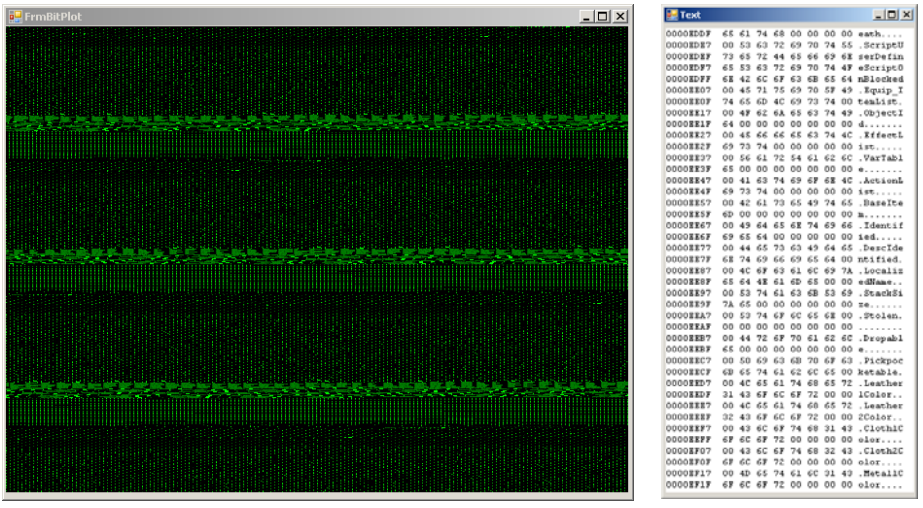



**Fig. 3.** Byte view of a Neverwinter Nights database file (left). Notice the regularity of the fixed length record structure. The text view (right) allows the analyst to see the low level details.

are 16-bit, but are otherwise the same values as ASCII. These byte values appear as alternating vertical lines in the byte view visualization. Another important insight is that while the byte view visualization we implemented was 640x480 resolution, larger display resolutions, such as a 1920x1200, are computationally feasible. Because preattentive processing allows analysts to rapidly identify patterns, a 1920x1200 display would allow far more rapid detection of embedded messages using Long's technique.

### 5.2 Identifying Fixed and Variable Length Records

As the preceding example illustrated, the byte view visualization allows users to view internal structure. This trait is particularly valuable when viewing files containing regions of fixed or variable length records. Record structure is immediately visible, as seen in Figure 3(left), which depicts a fixed length structure storing data from the game Neverwinter Nights. Figure 4 depicts variable length packets stored in the PCAP file format. After the analyst identifies the record structures, they can then explore the details using the text view display, Figure 3 (right).

### 5.3 Microsoft Word Analysis

The Microsoft Word binary file format is extremely complex.[5] To gain a better understanding of its inner workings, we used our visualization system to explore the internal structure of a large (10.3MB) Microsoft Word document, containing approximately 5,000 words, 16 embedded images and 36 footnotes. Because of the file's size, the entire document required just over 33 pages in the byteview visualization to examine

[5] The Microsoft Word specification document is 210 pages long and may be downloaded at microsoft.com.

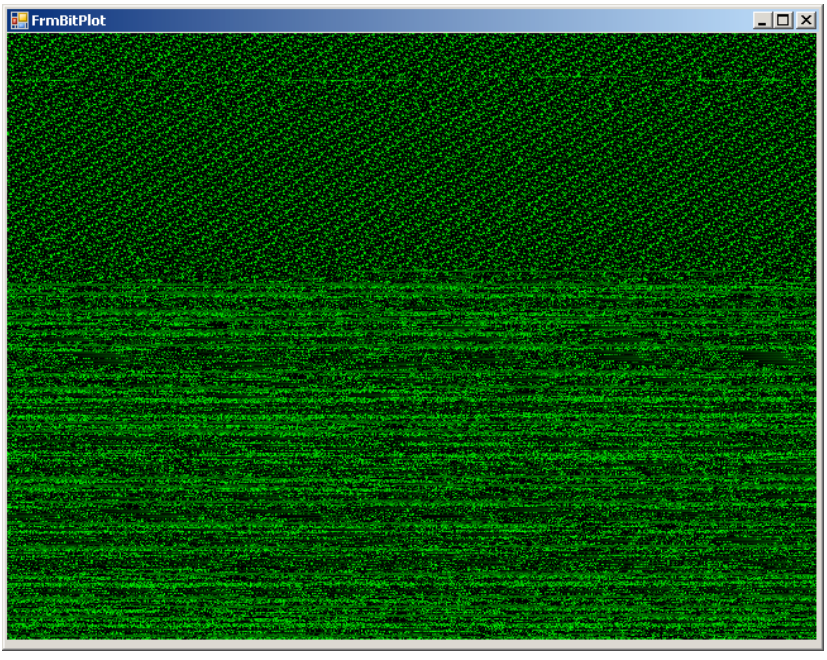
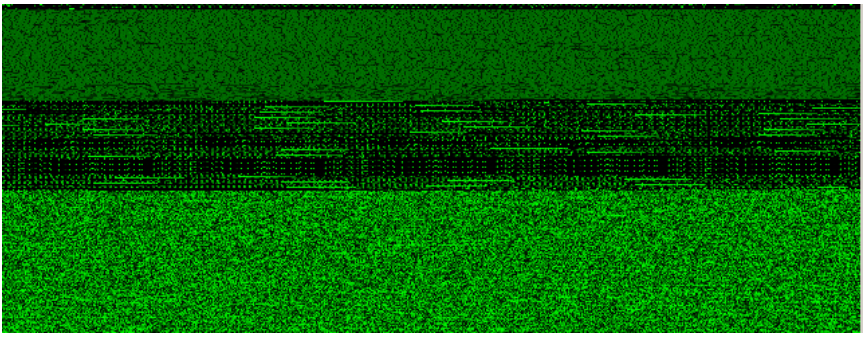


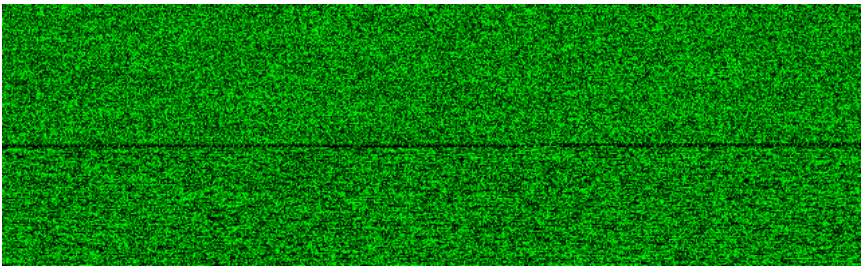

**Fig. 4.** Byte view of a PCAP file from the Defcon Capture the Flag competition. Notice the regularity of the fixed length record structure in the top half of the image and the variable length records in the bottom half.

the file in its entirety. However, this same size document would require approximately 1,024 pages when displayed in a textual hex editor-style format. In addition, the scroll bar on the VCR-like display greatly increased analysis speed. After initially loading the file and opening the byteview window, we used the scroll bar to scan the entire file, a process taking less than a minute. It quickly became apparent that the file contained a header region Figure 5(a), a large compressed or encrypted region, Figure 5(b), and a footer region, Figure 5(c). We used a combination of other visualization displays to provide deeper insight and confirm these initial assumptions. For example, by clicking on major structures in the header region and viewing the results in the text visualization, we confirmed the document's text was located in the top third of figure 5(a). Embedded images constituted the vast majority of the document and appeared as white noise. Each image was preceded by a short header, which was visible in the byteview visualization as a horizontal bar, see Figure 5(b). Closer examination of these image headers using the text view revealed that they were compressed PNG images. The footer contained a mixture of elements including a listing of all hyperlinks contained in the document stored as Unicode. Recall that basic Latin Unicode appears as vertical bars in the byteview visualization.

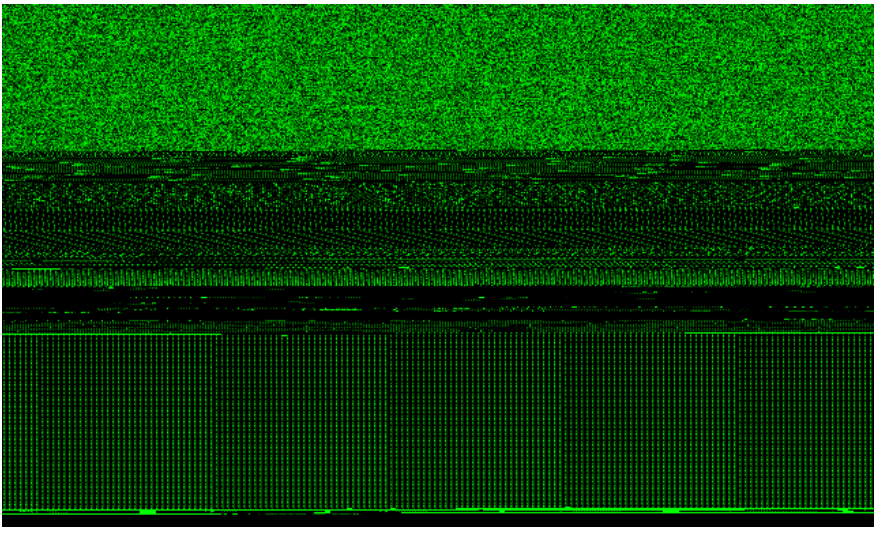
We believe our visual analysis approach bears great promise for analyzing documents stored in binary files. ASCII data, Latin Unicode, internal record structures, and compressed images are all readily apparent. Potential future applications include using visualization to help guide *fuzzing*, the stress testing of application parsers, by



(a) Header Region



(b) Embedded Image Region



(c) Footer Region

**Fig. 5.** Microsoft Word Binary. The byteview visualization allows the analyst to quickly discover the existence of three major regions in the file. A header region, which contains the text of the document, followed by a large region containing compressed images, and a footer region which includes hyperlinks stored as Unicode.

facilitating identification of internal structures. A common best practice in the fuzzing community is the study of complex file formats as the probability of discovering a vulnerability increases with complexity [13].

### 5.4 Firefox Core Dump

This final example is a core dump created by a Firefox browser during a crash and differs significantly from the preceding examples, as it is a snapshot of process memory and not a static file format. As such, additional structures not seen during static analysis become visible. For example, Figure 6, shows an image stored by the browser in its process memory, note the gradients (left) and the corresponding byte utilization in the byte presence view (right).

Additional analysis indicated that our visualization approach is useful for related, and potentially very large chunks of binary data, including page files, hibernation files and process memory. It is important to note however, that sharing byteplot images in these cases is a security concern, because it is possible to convert the image back to the raw byte values without loss.

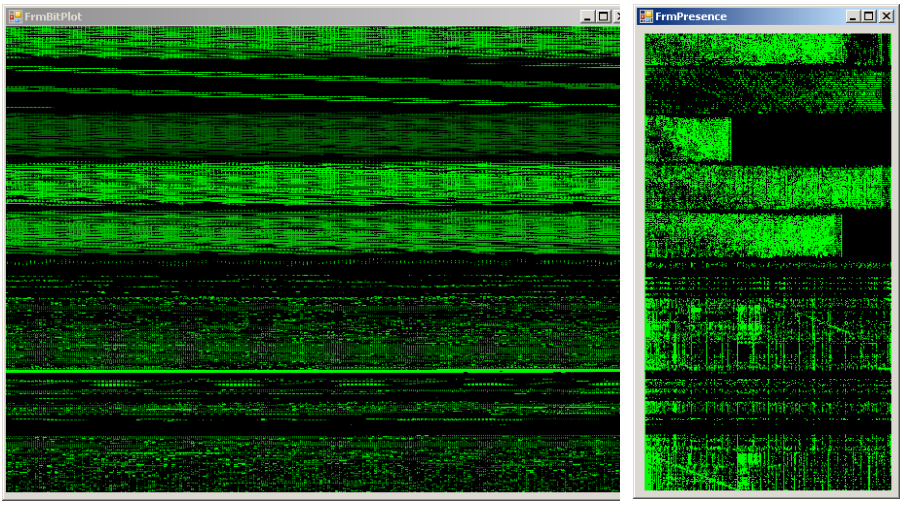



**Fig. 6.** Byteplot view of process memory dumped by Microsoft Windows after a Firefox browser crash. The left figure depicts an image stored in process memory (note the gradients) and the right figure shows the corresponding byte values.

## 6 Conclusions and Future Work

The future of visual analysis of binary data is promising, particularly when such visualization systems incorporate best practices from hex editors, a well-studied field for over 30 years. Our work demonstrated that it is possible to extend the current functionality of the hex editor metaphor by overcoming its significant constraint of a tiny textual window and helping fill the distinct gap between the hex editor and special case binary analysis tools such as disassemblers. Our intent was not to suggest

rejecting the hex editor, but instead buttress its weaknesses and complement its strengths via visualization and improved interaction. A key question we sought to answer was, "Is it possible to do better than the canonical hex/ASCII view provided by today's hex editors?" The answer is yes. Carefully crafted visualizations provide big picture context and facilitate rapid analysis of both medium (on the order of hundreds of kilobytes) and large (on the order of tens of megabytes and larger) binary files. The traditional hex editor is an inadequate tool for dealing with files of these sizes. However, the traditional hex editor view provides a useful means of providing precise detail. It is possible to create a visualization-enhanced analysis system that combines the functionality of the best hex editors with the strengths of visualization. Key to this approach is the continued exploration of interaction techniques to seamlessly blend visual displays with hex editor interaction best practices. To be most successful, such a system should be based on an extensible plug-in architecture that allows intermediate and advanced end-users to easily create and share both visualization techniques and search/filtering/coloring rules, tapping the combined insight of the user-community.

## References


1. Conti, G., Grizzard, J., Ahamad, M., Owen, H.: Visual Exploration of Malicious Network Objects Using Semantic Zoom, Interactive Encoding and Dynamic Queries. In: IEEE Symposium on Information Visualization's Workshop on Visualization for Computer Security (VizSEC) (October 2005)
2. Helfman, J.: Dotplot Patterns: A Literal Look at Pattern Languages. TAPOS Journal 2(1), 31–41 (1995)
3. Kaminsky, D.: Black Ops 2006. Blackhat USA (2006) (last accessed December 20, 2007), http://www.doxpara.com/slides/dmk_blackops2006.ppt
4. Yoo, I.: Visualizing Windows Executable Viruses Using Self-Organizing Maps. VizSec/DMSec (2004)
5. Carrera, E., Erdelyi, G.: Digital Genome Mapping – Advanced Binary Malware Analysis. In: Virus Bulletin Conference (2004)
6. Flake., H.: Structural Comparison of Executable Objects. Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), pp. 161–173 (2004)
7. A different look at Bagle. F-Secure Weblog (23 September 2005) (last accessed December 20, 2007), http://www.f-secure.com/weblog/archives/00000662.html
8. Graphing malware. F-Secure Weblog (25 October 2005) (last accessed December 20, 2007), http://www.f-secure.com/weblog/archives/00000324.html
9. Dullien, T., Rolles, R.: Graph-based comparison of Executable Objects. In: Symposium Sur La Securite Des Technologies De L'Information Et Des Communications (SSTIC) (2005)
10. Flake, H.: Diff, Navigate, Audit – Three applications of graphs and graphing for security, Blackhat USA (2004) (last accessed December 20, 2007), http://www.blackhat.com/presentations/bh-usa-04/bh-us-04-flake.pdf
11. Nolan, B., Sinda, M.: File Visualization Environment (FiVE). In: National Conference on Undergraduate Research (2008)
12. Kaminsky, D.: Black Ops 2006 : Viz Edition. Chaos Computer Congress (2006) (last accessed May 1, 2008), http://www.doxpara.com/slides/dmk_blackops2006_ccc.ppt
13. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley, Reading (2007)