File Type Detection Technology

Douglas J. Hickok
Computer Science and Software Engineering Department
University of Wisconsin – Platteville
Platteville, WI 53818
hickokd@uwplatt.edu

Daine Richard Lesniak
Computer Science and Software Engineering Department
University of Wisconsin – Platteville
Platteville, WI 53818
lesniakd@uwplatt.edu

Michael C. Rowe, Ph.D.
Computer Science and Software Engineering Department
University of Wisconsin – Platteville
Platteville, WI 53818
rowemi@uwplatt.edu

Abstract

File type identification is a difficult but increasingly important task. There are no global standards for file types and there are catalogues of several thousand known file types. Filters, such as email SPAM, virus blockers, adware/spyware, and steganalysis detectors are generally file type specific and may need accurate file type knowledge to work effectively. Corporations and government agencies block specific file types in an attempt to keep sensitive data from leaving or worms and viruses from entering their networks. Another important use of file type is as a means of limiting "recreational" file use that can absorb significant quantities of network bandwidth and storage. Some of the more commonly filtered file types for the above considerations include: ".vb", ".vbs", ".exe", ".mp3", ".wav", ".mpeg", ".wmv", and ".avi".

Introduction

With privacy, security, and wise use of computational resources becoming increasingly important to organizations, the technologies that address these concerns are increasingly being faced with the problem of file type detection.

One method of safeguarding privacy, security, and resource use is email attachment filtering. By blocking certain attachments, security can be protected by blocking in bound attachment types that are capable of harboring malicious content, privacy can be safeguarded by filtering items on the way out that may contain potentially revealing information, and resource management can be facilitated by blocking items that are primarily used for entertainment. Preventing the dissemination of revealing information has become especially important for healthcare providers as of late with the enactment of HIPAA, the health insurance portability and accountability act. Under HIPAA, health care providers can face lawsuits and monetary penalties if they do not make sufficient effort to protect privacy of patient information [1]. Filtering attachments successfully depends on reliable and efficient file type detection.

A method that is central to any effort to protect security is virus scanning. While virus scanning does not rely as heavily on file type detection as attachment filtering, virus scanners often skip file types, which are believed to be incapable of harboring viruses. If the file type detection technique is not reliable, viruses may not be detected until it is too late.

Steganography is the art of hiding in plain sight [2, p5]. Steganography tools can hide messages inside of pictures or other files without changing their appearance. Steganalysis, which detects hidden content, is an extremely important field at the moment, spurred by not only traditional security concerns but also by reports that terrorists may communicate through steganography [3]. Companies' sensitive information is also at risk due to steganography and it is an attractive method for smuggling sensitive data out of organizations. Analysts estimate that between 70% to 90% of all corporate network attacks come from the inside, and that the number of "insiders" is growing exponentially [4]. The FBI estimates that corporations loose $24 billion to information theft each year [5]. Due to these concerns, there is an increasing interest in steganalysis, which in turn relies heavily on file type detection. Automated steganalysis depends on file type detection due to the fact that steganography methods are often dependent of file type, with different methods of embedding being used on pallet based images vs. jpegs, and different methods being used on images vs. other binaries. If the wrong steganalysis method was applied to the wrong file type, then steganographic content would not be detected or a false alarm would be signaled. Using every steganalysis method on every file would be computationally inefficient, and inappropriate for an automated system.

We have identified three main methods of file type detection:

1. identifying file types by file extension,
2. identifying file type by magic bytes, and
3. identifying file types by character distribution.

Each of these detection methods has strengths and weaknesses, and no one method is comprehensive or foolproof enough to satisfy file type detection needs. This indicates that file type detection is an area that warrants additional attention.

File Type Detection using File Extension

The simplest and most common file type detectors employ the naïve method of looking only at file extensions. This naïve file type detector can easily be spoofed by simply renaming a file. There are multiple ways of determining file type other than trusting the file extension.

File type detection based on extension does not even have to open the file in order to detect the type it claims to be, and is by far the fastest way to classify a file. Speed of file type detection is important due to the volume of files that must pass through these utilities, and is a major reason for the use of file type detection based on extension.

A second reason extensions are examined in file type detection is that all file types are generally accompanied by an extension, at least in Windows based systems. This allows this type of method to be applied to both binary and text based file types.

Problems with Using File Extension

The major problem with using file extensions for file type detection is that extensions are easily spoofed and altered. It is painless to change the extension in Windows, and requires little more than a couple mouse clicks and a few keystrokes. Just as it is not necessary to open a file when classifying based on extension type, it is not necessary to open a file to mislead this classification technique.

Linux/Unix systems introduce another complication with file type detection based on extension type, in that an extension is not required on Linux systems. Not only is the fact that extensions are not necessarily present cause a problem, but Linux allows optional extensions of any string regardless of file type. This allows executables and scripts to be hidden from inexperienced administrators. An example of this would be a malicious executable named evil.mp3 and appear at first glance to be a music file. This script would not even need to have its extension changed, and would be run by typing "./evil.mp3". Windows is a little more dependant on file extensions. If you double-click on a renamed image file called "bitmap.mp3", it will try opening it in a music player and fail. However, if you use Paint to open it, it will open without a problem [6, p105].

Attachment filtering based on file extension is not only the most common method, but also the most easily bypassed [7]. When attachment filtering is first implemented, the first thing people naturally try is to get around the filter by changing the attachment file extension to that of a file type allowed. This transforms an attachment based email filter from a useful tool in improving security, privacy, and resource use into a mild annoyance to those who wish to bypass it.

File Type Detection with Magic Bytes

A more sophisticated method of file type detection uses what are referred to as "magic bytes". The magic bytes are specific to binary files and rely on matching signatures that vary in length from two to 46 bytes in file headers. Some files also have signatures in their tails or in other places. Many graphics formats, ".bmp", ".jpeg" and others have color pallets with specific structures. The pallets are associated with particular encoding methods, and thus, identify file type. There are several hundred file types for which magic bytes are defined and there are multiple lists of magic bytes, although they are not always consistent.

Additionally, magic byte techniques may give information regarding the tool and/or the tool's version used to produce the file. For example,

09 02 06 00 00 00 10 00 B9 04 5C 00 and
09 04 06 00 00 00 10 00 F6 05 5C 00

are both Microsoft ".xls" files with the first produced by Excel v2 and the second by v4.

Interestingly, in files "*captured in the wild*" we have found files (9 out of 474 in our test sample) that have jpeg file extensions, behave as jpeg files with respect to common viewers, but do not conform to usual magic byte conventions.

Magic bytes do not always give a very specific answers, for example 4D 5A are the magic bytes for executable files and is commonly used by ".exe", ".com", ".386", ".ax", ".acm", ".sys", ".dll", ".drv", ".flt", ".fon", ".ocx", ".scr", ".lrc", ".vxd", ".cpl", and ".x32" file types.

The UNIX *file()* command uses magic bytes as one mechanism to determine the character set (ASCII, Unicode, EBCDIC, etc.), whether a file is executable, whether it is a binary data file, along with other characteristics. In UNIX, the magic numbers can generally be found at /usr/share/file/magic.mgc [8].

How Magic Bytes Work

Magic bytes are typically the first couple of bytes in a file. Since there are no standards for what a file can contain, the creators of a new file type will usually include something to uniquely identify files of their type. For example, in 1986 a company called PKWARE invented the ZIP file format for compressing files [9]. Since then, the letters "PK" have been at the beginning of every ".zip" file in order to identify it as a file in the

ZIP file format. Now, even though many different types of software work with ".zip" files, the first two bytes remain "PK" to follow PKWARE's original standard.

Checking the magic bytes of a file is a little slower than just checking the file's extension, because the file must be opened and a small number of bytes must be read. Once the bytes are read, they can be compared to what is expected. If a file's extension is ".zip", then the first two bytes of the file must be read. If the bytes are anything other than "PK", the file is flagged as suspicious.

Table 1 shows a small list of file types with their magic bytes at the beginning and sometimes at the end of the file. Figure 1 shows a hex view of a typical ".wav" sound file. To verify an unknown ".wav" file, read in the first 12 bytes of the file, then see if it matches the pattern "RIFF????WAVE" where "?" could be any byte value.

Table 1: A list of common file types and their magic bytes. If the bytes happen to be alpha-numeric, they are enclosed in quotes for readability.

File Type	Header Magic Bytes	Footer Magic Bytes
RTF	"{\rtf1\"	"}}"
PDF	"%PDF-<version>"	"%%EOF" plus optional CR/LF
JPG	FF D8	None
GIF	"GIF87a" or "GIF89a"	None
PNG	89 50 4E 47 0D 0A 1A 0A	None
WAV	"RIFF" plus "WAVE" at offset 0x08	None
ZIP	"PK"	None
EXE/DLL/SCR etc.	4D 5A	None

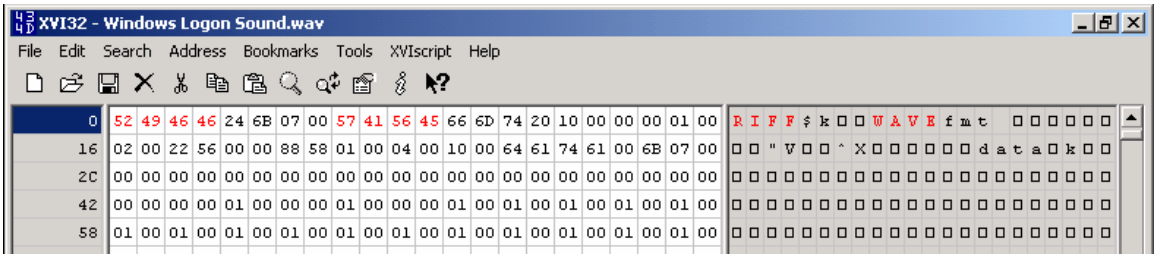

Figure 1: A hex view of the contents of a typical ".wav" file. The magic bytes are in red.

Problems with using Magic Bytes for File Detection

Magic bytes are not without their problems. For file type detection, magic bytes have one major flaw: they only work on binary file types, and only if those file types do in fact have magic bytes associated with them. This means that file type detection based on magic bytes can only be performed if the file is of a certain type, leaving a large number

of files that would have to be classified as unknown. This can be very problematic when one considers the risks associated with ignoring detection of ASCII based files. Malicious scripts are a major security threat to any Linux or Windows system, and simply ignoring C++ files leaving a software development station over email may be an expensive oversight.

A second problem associated with magic bytes is that they are not an enforced or regulated aspect of file types. Often the developer of a new file type uses their initials as the magic bytes; for example, “PK” are the initials of the PKWARE’s founder Phil Katz, thus they are Phil Katz’s magic numbers. Also, ".exe” files start with 4D 5A or “MZ” which are the initials of the file format designer, Mark Zbikowski [10]. While some formats specify headers that can be used as magic bytes, other magic bytes come simply from de facto standards, i.e. “Well, everyone else throws these bytes in this file type, I guess I should too.” The problem is that some programs will not follow the standards when creating or editing files or the developers may never have known of such standards to begin with.

Adding to the problem of magic bytes being a soft standard, if in fact a standard at all, is the fact that many sources of magic byte information do not agree 100%. In our work we came across several tables that claimed to be the definitive magic byte compilation, but not only were there omissions of file types from some that were covered instead by others, but the varying tables disagreed on what the magic bytes were for certain file types. In general, the information would not be contradictory so much as of varying degrees of specificity. An example of this would be some of the listed magic bytes for jpeg files. While [11] states that the magic bytes are FF D8 FF, [12] lists the .jpg magic bytes as either FF D8 FF FE 00 or FF D8 FF E0 00. As previously stated, the main difference is that of specificity.

This raises an interesting problem, if a file of file type ‘A’ can be opened and edited by all applications that deal with file type ‘A’, but the file lacks the standard magic bytes that file type ‘A’ files usually have, what is to be done?

When using magic bytes for file type detection, it is possible that a false positive will occur simply due to chance. The magic byte scheme of bitmaps is that the first two bytes are 42 4D, which is “BM” in ASCII text. If all character combinations were equally probable, this translates into a 1/65536 chance that a non-bitmap file with the first two bytes being randomly selected would test positive for being a bitmap if magic byte based file type detection is used. While this may seem like a small chance, when one considers the sheer number of file types and files in circulation, it is clear that this is a potential problem. It is interesting to note that a text file that contains nothing more than “BMW’s are good cars.” would be flagged as a bitmap by magic byte based detection. It should be noted that the possibility of a chance based false positive drops off quickly as the number of magic bytes examined increases. Using three magic bytes results in a 1/16777216 chance and using four magic bytes results it a 1/4294967296 chance, but it is the file type itself that dictates how many magic bytes can be used.

In addition to other problems with magic bytes, they are not immune to being spoofed intentionally to defeat file type detection. Although more effort must be taken to spoof magic bytes than an extension, a simple hex editor and knowledge of what magic byte conventions are is all that is required. Altering magic bytes of a file does not disturb its functionality, so it is possible to create an mp3 that would appear to be a jpeg in an attempt to thwart a magic byte based mp3 filter, and not even have to change the magic bytes back to the original configuration in order to listen to the song.

File Type Detection using Distribution of Characters

A third method that may be used to help verify file type is to examine the distribution of ASCII values in a file. The method that is used is to tally the distribution of each ASCII value in a file is called a histogram method [2, p192]. This is used for checking different types of character-based files, but also may be used on all or parts of binary files. A histogram, unlike the previous two file detection methods, could reveal that "normal.txt" is actually a malicious JavaScript instead of a normal text file.

A histogram is made by reading the contents of a file, and counting the number of times each ASCII character occurs. Normal text consists of all the letters of the alphabet, numbers, spaces, some symbols and punctuation, and the ASCII representations of *tab* and *enter*. Figure 2 shows a normal text file, which is the story of *Captain Midnight*. The two bars on the left represent the *enter* key (CR and LF), the highest single bar which extends out of the picture is the *space*, and the section on the right are the lower case alphabet. The highest bars in the lower case alphabet are the most common letters in the text file: 'a', 'e', 'i', 'n', 'o', 's', and 't'. Many of these letters correspond with the list of most commonly used letters in the English language. Figure 3 shows a C++ file, which has an unusually high number of symbols like '(,)', '{', '}', '[', ']', '_', '*', '+', '=', and ';' and tabs. All of these symbols are common in code files and scripting languages like JavaScript. This technique is not limited to text files, some binary files have easy to spot patterns as well, like the ".wav" file in figure 4. The peak ASCII character is 128, which is considered to be silence (or low decibel samples) in wave file data.

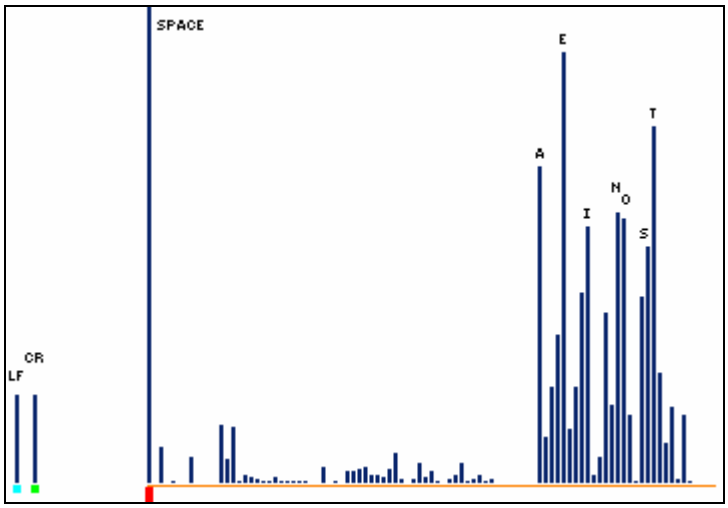

Figure 2: Histogram of the character occurrences in the story of *Captain Midnight*. Note that the peaks represent high frequency characters in the English language in addition to linefeed, *carriage-returns* and *spaces*.

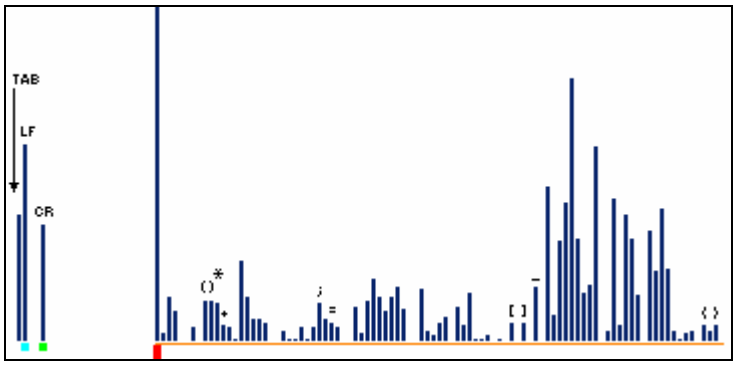

Figure 3: Histogram of the character occurrences in a C++ file. Note the unusually high number of non-alphabetic characters. Note that there are pairs of corresponding characters that have equal occurrences.

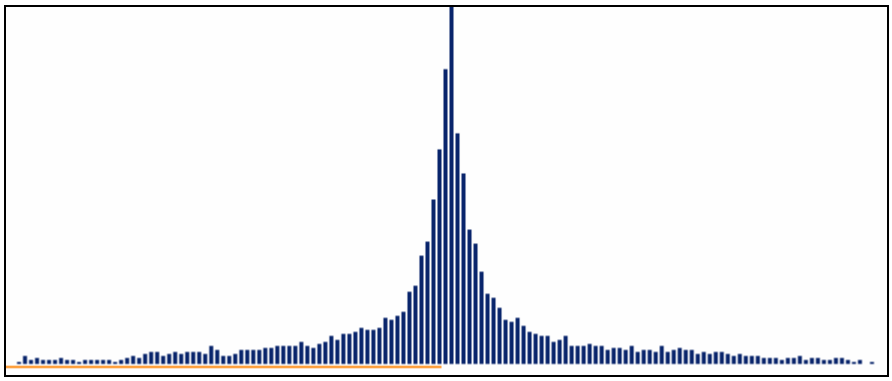

Figure 4: Histogram of a .wav file. The peak is from low-decibel samples.

Problems with using Distribution of Characters for File Detection

Like all methods, file type detection based on distribution of characters has its flaws. Among these flaws are that some file types do not have a characteristic distribution, and the risk of false alarms due to files with unique or irregular content. The most challenging aspect of file type detection based on character distribution is that of computational efficiency.

In most instances, file type detection must be relatively quick. In attachment scanners that may have to filter through an entire organization's incoming email in a timely manner, file type detection must be fast or it will hinder daily business. Virus and adware scanners are required to scan through more and more files and hard drive sizes and the amount of media stored by individuals increases, requiring the file type detection to be efficient if these utilities are to remain useful. Automated steganography is especially reliant in quick file type detection, as it needs to check unobtrusively each file as it is created on a system. File type detection based on character distribution can be inappropriate for such uses since it must open and scan every file, while keeping statistics on the file as it scans.

Our Work

The end purpose of this work is to develop a front-end processor for a steganalysis framework. This paper describes our design and implementation of a file type verifier. This work is supported by U.S. Air Force Research Laboratory, Phase I STTR – FA9550-04-C-0109. File type detection is a necessary first step in this process, which enables the most likely steganalysis techniques to be tried first.

The issue with the use of file type classification in a steganalysis system is that individuals who are sophisticated enough to try hiding a message through the use of steganography certainly have the knowledge to disguise the file types.

Current Work

Our current file type detection scheme is based on a combination of extension and magic bytes, with magic bytes having the most significance. We utilized magic bytes for a variety of reasons. One reason we focus on magic bytes for our file type detection needs is the balance it provides between speed of classification and difficulty to spoof. We found that doing a full byte distribution analysis would be unacceptable for an automatic process that would need to monitor and scan all files as they were created, as automated steganalysis does. We decided early on that merely relying on the extension was ruinous, due to not only the simplicity of spoofing but by the fact that we would be limiting ourselves to Windows, and automated steganalyis may be put into service on individual workstations, which may or may not be running Windows, or on network servers, which often run Linux/Unix.

In our implementation, we created plug-ins for specific file types. Each was designed to verify a specific type of file by reading and comparing magic bytes. Instead of reading in the maximum number of bytes that could be magic bytes and figuring out what the file type is, we started off by trusting the file's extension, and passed the filename to the corresponding plug-in that could handle it. If the file's magic bytes matched what was expected, it was assumed to be ok. If not, it was flagged as suspicious.

At the moment we are accepting certain downfalls of relying heavily on magic bytes, such as not being able to verify ASCII files and the fact that they can still be spoofed. We are however looking into ways to improve our current work, and create even more robust and reliable file type detection.

Future Work

Possible ways to improve our current file type detection method would be to mitigate the resource impact of byte distribution scanning, the use of string searches to find strings that are often, but not always, embedded in certain file types, and the possible use of machine learning in file type detection.

Finding methods to make byte distribution scanning fast enough to be used would greatly improve file type detection capabilities. One possible way would be to sample a random selection of a file's bytes and verify the profile of the random selection against known signatures. This would increase speed while decreasing certainty of results, making it crucial that the correct sample size be selected to balance these factors.

Certain files can sometimes contain strings that indicate what tools were used on the file, what versions, and other information that could possibly indicate file type. By examining different files we have found strings that tag files on which operating system the file was created, what programs created the file, and in the case of digital photos what camera was used to take the picture. While these are not necessarily hard and fast indications of file type, they are valuable clues and could possibly be utilized to facilitate more flexible file

type detection. Unfortunately scanning a file for strings is an even slower process than a byte distribution analysis, and offers far less benefit.

One method that may hold promise for file type detection is the use of machine learning techniques. Supervised training could be used to construct a file type classifier, but several questions remain. One question would be what method of machine learning would be flexible enough to learn new file types as they became important, and not forget how to classify the previously learned file types; one way around this may be to train many smaller more specific file type detectors. Another issue with machine learning is the question of what the feature vector should be. If one large classifier was utilized the vector would need to contain information useful for differentiating all types, while using many smaller file type classifiers would mean each file type would need to be examined as to what the useful features are. Machine learning is both the most promising and most challenging method of file type detection, and will no doubt be investigated in the future.

Summary

File type identification is a difficult but increasingly important task. SPAM email filters, virus scanners, adware/spyware blockers, and steganalysys detectors are generally file type specific and may need accurate file type knowledge to work effectively. File type detection using only the file extension, which is commonly used in all the previously mentioned programs, may not be enough. By simply renaming the file, the items that are blocked, filtered, or scanned are usually passed through. Detection with the use of magic bytes provides a much more accurate way to prove that a file is what it claims to be. However, magic bytes aren't standardized and reliable, and can trigger some false alarms. Also, magic bytes only work with most binary files, which leaves out the possibility of detecting potentially malicious text-based script files. File type detection using distribution of characters has the potential to classify text-based files, but is too resource intensive to be used on an increasingly large number of files in a timely manner.

References

[1] http://www.cms.hhs.gov/hipaa/hipaa2/enforcement/default.asp#penalties

[2] Cole, E., *Hiding in Plain Sight,* Wiley, Indianapolis, 2003.

[3] http://www.thebulletin.org/article.php?art_ofn=mj01auer

[4] http://www.cryptek.com/Downloads/corporate.pdf

[5] http://www.securitymagazine.com/CDA/ArticleInformation/features/BNP__Features__Item/0,5411,77194,00.html

[6] McNamara, J., *Secrets of Computer Espionage*, Wiley, Indianapolis, 2003.

[7] http://ask-leo.com/how_can_i_send_someone_an_attachment_if_its_blocked_by_their_copy_of_outlook.html

[8] http://www.die.net/doc/linux/man/man1/file.1.html

[9] http://www.pkware.com/company/background/

[10] http://en.wikipedia.org/wiki/Magic_number_%28programming%29

[11] http://filext.com/detaillist.php?extdetail=JPG

[12] http://www.techpathways.com/uploads/headersig.txt

Acknowledgements

The end purpose of this work is to develop a front-end processor for a steganalysis framework. This paper describes our design and implementation of a file type verifier. This work is supported by U.S. Air Force Research Laboratory, Phase I STTR – FA9550-04-C-0109.