# Fileprints: Identifying file types by n-gram analysis

**Conference Paper** · July 2005

DOI: 10.1109/IAW.2005.1495935 · Source: IEEE Xplore



**4 authors**, including:

Wei-jen Li
Columbia University
**15** PUBLICATIONS **476** CITATIONS

SEE PROFILE

Salvatore J. Stolfo
Columbia University
**315** PUBLICATIONS **20,638** CITATIONS

SEE PROFILE

Proceedings of the 2005 IEEE
Workshop on Information Assurance
United States Military Academy, West Point, NY June 2005

# Fileprints: Identifying File Types by n-gram Analysis

Wei-Jen Li, Ke Wang, Salvatore J. Stolfo, Benjamin Herzog *Columbia University*
{Wei-Jen, Kewang, Sal} @cs.columbia.edu

***Abstract – We propose a method to analyze files to categorize their type using efficient 1-gram analysis of their binary contents. Our aim is to be able to accurately identify the true type of an arbitrary file using statistical analysis of their binary contents without parsing. Consequently, we may determine the type of a file if its name does not announce its true type. The method represents each file type by a compact representation we call a fileprint, effectively a simple means of representing all members of the same file type by a set of statistical 1-gram models. The method is designed to be highly efficient so that files can be inspected with little or no buffering, and on a network appliance operating in high bandwidth environment or when streaming the file from or to disk.***

## I. INTRODUCTION

Files typically follow naming conventions that use standard extensions describing its type or the applications used to open and process the file. However, although a file may be named *Paper.doc*[1], it may not be a legitimate Word document file unless it is successfully opened and displayed by Microsoft Word, or parsed and checked by tools, such as the Unix *file* command, if such tools exist for the file type in question.

The Unix *file* command performs several syntactic checks of a file to determine whether it is text or binary executable, otherwise it is deemed data, a "catch all" for just about anything. These tests include checking of header information, for example, for "magic numbers", to identify how the file was created. One test performed by *file* considers whether the bulk of byte values are printable ASCII characters, and hence such files are deemed text files.

The magic numbers serve as a "signature" to identify the file type, such as in the case of .PDF files where the header contains "25 50 44 46 2D 31 2E". However, a pure signature-based (or string compare) file type analyzer [1, 2] runs several risks. Not all file types have such magic numbers. The beginning of the file could be damaged or purposely missing if obfuscation is used. For example, malicious code can be hidden by many techniques, such as the use of binders, packers or code obfuscation [3, 4]. In the case of network traffic analysis, due to different packet fragmentation, the beginning portion of a file may not be entirely contained in a single packet datagram or it may be purposely padded and intermixed with other data in the same packet to avoid signature-based detection. Finally, not all file types have a distinct "magic number".

In this paper we propose a method to analyze the contents of exemplar files using efficient statistical modeling techniques. In particular, we apply n-gram analysis to the binary content of a set of exemplar "training" files and produce normalized n-gram distributions representing all files of a specific type. Our aim is to determine the validity of files claiming to be of a certain type (even though the header may indicate a certain file type, the actual content may not be what is claimed) or to determine the type of an unnamed file object. In our prior work, we exploited this modeling technique in network packet content analysis for zero-day worm detection [5]. We extend that work here for checking file types, whether in network traffic flows or on disk. In our prior work we generate many models conditioned on port/service and length of payload. This generates a set of models that very accurately represent normal data flow and identifies different data quite accurately.

McDaniel and Heydari [6] introduced algorithms for generating "fingerprints" of file types using byte-value distributions of file content that is very similar in spirit to the work reported here. There are, however, several important differences in our work. First, they compute a single representative fingerprint for the entire class of file types. Our work demonstrates that it is very difficulty to produce one single descriptive model that accurately reporesents all members of a single file type class. Their reported experimental results also demonstrate this. Hence, we introduce the idea of computing a set of centroid models and use clustering to find a minimal set of centroids with good performance. Furthermore, the McDaniel paper describes tests using 120 files divided among 30 types, with 4 files used to compute a model for each type. We have discovered that files within the same type may vary greatly (especially documents with embedded objects such as images), and hence so few a number of exemplars may achieve poor performance.

[1] For our purposes here, we refer to .DOC as Microsoft Word documents, although other applications use the .DOC extension such as Adobe Framemaker, Interleaf Document Format, and Palm Pilot format, to name a few.

ISBN 555555555/$10.00 

Indeed, they report 3 variant algorithms achieving an accuracy of 28%, 46% and 96%. The results we report using a clustering strategy produces better results. In the case of the test producing an accuracy of 96% in their work, they analyze the leading header portion of files. Our work shows that each file type consists of fairly regular header information and we achieve a near 100% accuracy in file type classification. However, some file types do not have consistent header information. When more data is used as in our case, their results are rather poor.

There is also a significant difference in the method used to normalize their data. They state "*Once the number of occurrences of each byte value is obtained, each element in the array is divided by the number of occurrences of the most frequent byte value. This normalizes the array to frequencies in the range of 0 to 1, inclusive."* In their work, they seek to build a model invariant to file size. This may not be a good strategy. We believe a more proper normalization strategy would be to compute the byte value frequencies normalized by the length of the file. We demonstrate that this achieves more accurate centroid models.

a

Figure 1 displays a set of plots of example 1-gram distributions for a collection of popular file types. The 1-gram distribution shows the average frequency of each byte value over all training files represented as a 256-element histogram. The plots show the byte values in order from 0, 1, 2,…, 255. Notice how distinct each distribution is for each file type. These trained models serve to classify unknown files, or to validate the extension of a file by comparing the byte value distribution of the file in question to one or more model file distributions.[2] The histograms may contain the actual byte count, or it may be normalized so that the percentage of each byte value is represented. The choice is a subtle technical issue. For example, normalized histograms allow different length files to be compared directly.

Notice that the full 1-gram distribution, which is at most two 256-element vectors (representing the average byte frequency, and their variance), is very space efficient. We conjecture that these simple representations of file types serve as a distinct representation of all members of a single type of file, and hence refer to this concept as a *fileprint*. A fileprint may be a set of such histograms to represent a variety of example files of the same type.

[2] Since the byte value 0 is used often to pad files in various formats, one may ignore this value and focus on the remaining byte value distribution without loss of accuracy.

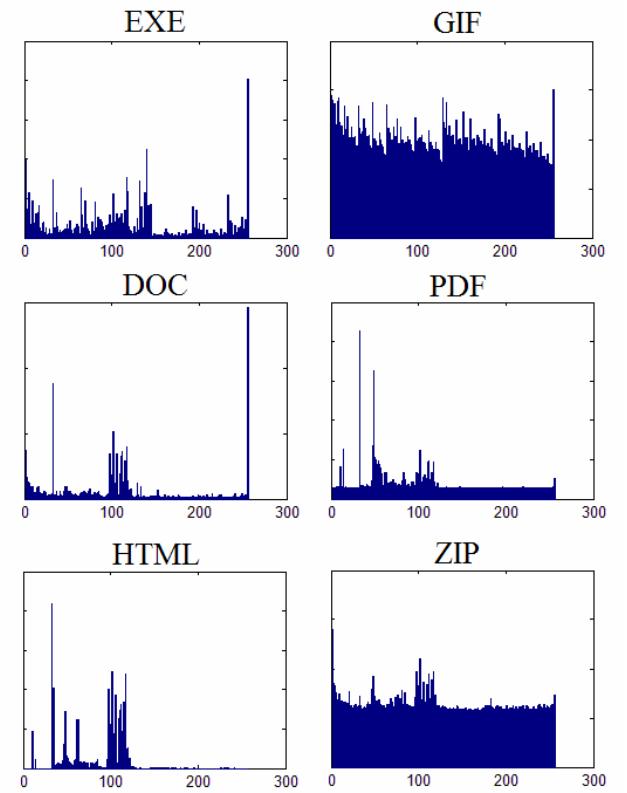


Figure 1: File binary distribution. X axis: bytes from 0 to 255, Y axis: normalized frequency of byte values (as %).

There are many potential uses of fileprints. As a network application, one may be able to quickly analyze packet data to identify the likely type of information being transmitted. Network integrity applications may be supported, and security policies may therefore be checked. For example, the transmission of Word documents outside of a LAN may be prohibited by company policy. Users quickly learn how to thwart such policies, by renaming their Word documents to .DDD, for example, to avoiding detection based upon tests of the file name in email attachments. However, a quick test of the 1-gram distribution of the content of the packet datagrams suggesting the content matches a Word document distribution would prevent such obfuscation. Thus, any Word document file seen on the wire but without the .DOC extension could be identified quickly by its fileprint and filtered before leaving the LAN.

Furthermore, infected file shares may be detected if they do not conform to an expected file type distribution. Hence, virus and worm propagation may be thwarted if certain files do not match their announced type. n-gram analysis of file content was first proposed for the detection of malicious virus code in our earlier work on the Malicious Email Filter project [7]. (That work has recently been extended by Kolter and Maloof [8] who evaluate a variety of related techniques.) In that prior

ISBN 555555555/$10.00 

work, a supervised training strategy was applied to model known malicious code and detect members of that class in email attachments. The n-gram distributions we used as input to a supervised Naïve Bayes machine learning algorithm to compute a single classifier of "malicious file content". In this work, we extend these techniques by calculating the entire 1-gram distributions of file content and use these as models for a set of file types of interest. The *distribution* of byte values of a file are compared to the models using well known statistical distribution distance measures. Here, we restrict our attention to only two, Mahalanobis and Manhattan distance.

For concreteness, suppose we have a file, $F$, of unknown type. In general, to distinguish between file-types $A$ and $B$, we compute two models, $M_A$ and $M_B$, corresponding to file types $A$ and $B$, respectively. To test the single file $F$, we check the distribution of its content against both models, and see which one it most closely matches. We assert its name as $F.B$ if the contents of $F$ closely match model $M_B$, i.e., the $D(\hat{H}(F.B), M_B)$ is less than some threshold, where $\hat{H}$ refers to the 1-gram distribution (a histogram), and $D$ is a distance function.

Alternatively, given file $F.A$, we may check its contents against $M_A$ and if the 1-gram distribution of $F.A$ is too distant from the model $M_A$, we may assert that $F.A$ is anomalous and hence misnamed. Thus, $D(\hat{H}(F.A), M_A) > T_A$ for some preset threshold $T_A$. We may suspect that $F.A$ is infected with foreign content and thus subject it to further tests, for example, to determine whether it has embedded exploit code.

In this paper, we test whether we can accurately classify file types. This test is used to corroborate our thesis that file types have a regular 1-gram representation. We apply a test to 800 normal files with 8 different extensions. We compute a set of fileprints (or "centroid" models) for each of the 8 distinct types, and test a set of files for correct classification by those models. Ground truth is known so accurate measurements are possible.

Several modeling strategies are explored. First, we use a "one-class" training evaluation strategy. A set of files of the same type are used in their entirety to train a single model. For example, given 5 different file types, we compute 5 distinct fileprints characterizing each type. A test file with an extension of one of these types is thus compared to the corresponding fileprint. This validation is computed by the *Mahalanobis* distance function applied to the distributions. In the second case, we compute *multiple models* for each file type by clustering the training files using $K$-means. The set of models are considered the fileprint. In this case, a test file is compared to all of the models of all of the types to determine the closest model. The latter case produces more models, but each provides a finer grained view of the training file type distributions, and hence may provide a more accurate fileprint classifier with fewer false positives. We extend this strategy to the extreme case. Rather than computing cluster centroids, we consider a set of exemplar files of a certain type as the fileprint. Each test file is compared to this set of pre-assigned exemplar files. The performance results of each of these tests show remarkably good results. We also perform these same tests using different portions of the files, a strategy we call truncation.

In section II we briefly describe n-gram analysis and an overview of the modeling techniques used in this study. Section III details the data sets and the detailed experimental results. Section IV concludes the paper.

## II. FILEPRINTS

### *A. n-gram Analysis*

Before demonstrating and graphically plotting the fileprints of the file contents, we first introduce n-gram analysis. An n-gram [9] is a subsequence of N consecutive tokens in a stream of tokens. n-gram analysis has been applied in many tasks, and is well understood and efficient to implement.

By converting a string of data to n-grams, it can be embedded in a vector space to efficiently compare two or more streams of data. Alternatively, we may compare the distributions of n-grams contained in a set of data to determine how consistent some new data may be with the set of data in question.

An n-gram distribution is computed by sliding a fixed-size window through the set of data and counting the number of occurrences of each "gram". Figure 2 displays an example of a 3-byte window sliding right one byte at a time to generate each 3-gram. Each 3-gram is displayed in the highlighted "window".

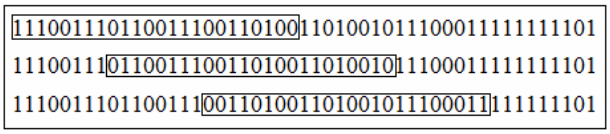


Figure 2: Sliding window (window size = 3)

The choice of the window size depends on the application. First, the computational complexity increases exponentially as the window size increases. Data is considered a stream of tokens drawn from some alphabet. If the number of distinct tokens (or the size of the alphabet) is $X$, then the space of grams grows as $X^N$. In the case of 3-grams computed over English text composed of the 26 letters of the alphabet, the space is $26^3$ distinct

ISBN 555555555/$10.00 

possible 3-grams. A string of $M$ letters would thus have ($M$-2) 3-grams with a distribution that is quite sparse.

In this work, we focus initially on 1-gram analysis of ASCII byte values. Hence, a single file is represented as a 256-element histogram. This is a highly compact and efficient representation, but it may not have sufficient resolution to represent a class of file types. The results of our experiments indicate that indeed 1-grams perform well enough, without providing sufficient cause requiring higher order grams to be considered.

### *B. Mahalanobis Distance*

Given a training data set of files of type A, we compute a model $M_A$. For each specific observed file, $M_A$ stores the average byte frequency and the standard deviation of each byte's frequency.

Note that the training and model computation of the byte value mean frequency, $\bar{x}$, may be computed in real-time as an incremental function as

$$\bar{x} = \frac{\bar{x} \times N + x_{N+1}}{N+1} = \bar{x} + \frac{x_{N+1} - \bar{x}}{N+1},$$

and similarly for the computation of the standard deviation. Hence, the models may be computed very efficiently while streaming the data without the need to fully buffer the file.

Once a model has been computed, we next consider the comparison of a test file against this model, either to validate the file's purported type, or to assign a type to a file of unknown origin. We use *Mahalanobis Distance* for this purpose. Mahalanobis Distance weights each variable, the mean frequency of a 1-gram, by its standard deviation and covariance. The computed distance value is a statistical measure of how well the *distribution* of the test example matches (or is consistent with) the training samples, i.e. the normal content modeled by the centroids. If we assume each byte value is statistically independent, we can simplify the Mahalanobis Distance as:

$$D(x, y) = \sum_{i=0}^{n-1} (| x_i - y_i | / (\sigma_i + \alpha))$$

where $x$ is the feature vector of the new observation, $y$ is the averaged feature vector computed from the training examples, $\sigma_i$ is the standard deviation and $\alpha$ is a smoothing factor. This leads to a faster computation, with essentially no impact on the accuracy of the models. The distance values produced by the models are then subjected to a test. If the distance of a test datum, the 1-gram distribution of a specific file, is closest to some model computed for a set of files of a certain type, the file is deemed of that type.

Alternatively, in some applications we may compute a distinct threshold setting, $T_A$ for each model $M_A$ computed. If the distance of the test file and $M_A$ is at or bellow $T_A$, the test file will be classified as type A. An initial value of $T_A$ may simply be the maximum score of the model distance to its training data, plus some small constant, $\epsilon$.

Since the type of a test file is unknown, we need a precise context in order to build a set of models for a set of *expected* file types. Suppose we are interested in building a virus detector for some host system, such as a Windows client machine. That client may regularly produce or exchange MS Office documents, PDF files, compressed archives, photographs or image files, and raw text files. In this example, we would need to model probably about 10 representative file types expected for the client machine in question. These 10 models would serve to protect the machine, by validating all files loaded or exchanged at run time. Recall, our goal is to ensure that a file claiming to be of type *A* actually matches the corresponding fileprint for *A*. For example, when receiving a file with extension .DOC that contains non-ASCII characters, it should be checked against the MS Word fileprint. In order to compute such models, we use the existing store of the client for training data to compute the fileprints. We follow this strategy in the experiments performed and described in the following sections. However, for some file types, we searched the internet using Google to prepare a set of "randomly chosen" representatives of a file type, to avoid any bias a single client machine may produce, and to provide the opportunity for other researchers to validate our results by accessing the same files that are also available to them.

### *C. Modeling and Testing Technique*

In this section, we describe several strategies to improve the efficiency and accuracy of the technique: *truncation*, reducing the amount of data modeled in each file, and *multiple-centroids* computed via clustering, a finer-grained modeling of each file type.

#### 1. Truncation

Truncation simply means we model only a fixed portion of a file when computing a byte distribution. That portion may be a fixed prefix, say the first 1000 bytes, or a fixed portion of the *tail* of a file, as well as perhaps a middle portion. This has several advantages:

- For most files, it can be assumed that the most relevant part of the file, as far as its particular type, is located early in the file to allow quick loading of meta-data by the handler program that processes the file type. This avoids analyzing a good deal of the

ISBN 555555555/$10.00 

payload of the file that is not relevant to distinguishing their type and that may be similar or identical to several different file types. (For example, the bulk of the data of compressed images, .JPG, may appear to have a similar distribution – a uniform byte value distribution – to that of encrypted files, such as .ZIP.)

- Truncation dramatically reduces the computing time for model building and file testing. In network applications this has obvious advantages. Only the first packet storing the prefix of the file may be analyzed, ignoring the stream of subsequent packets. If a file whose size is 100MB is transmitted over TCP, only 1 out of thousands of packets would therefore be processed.

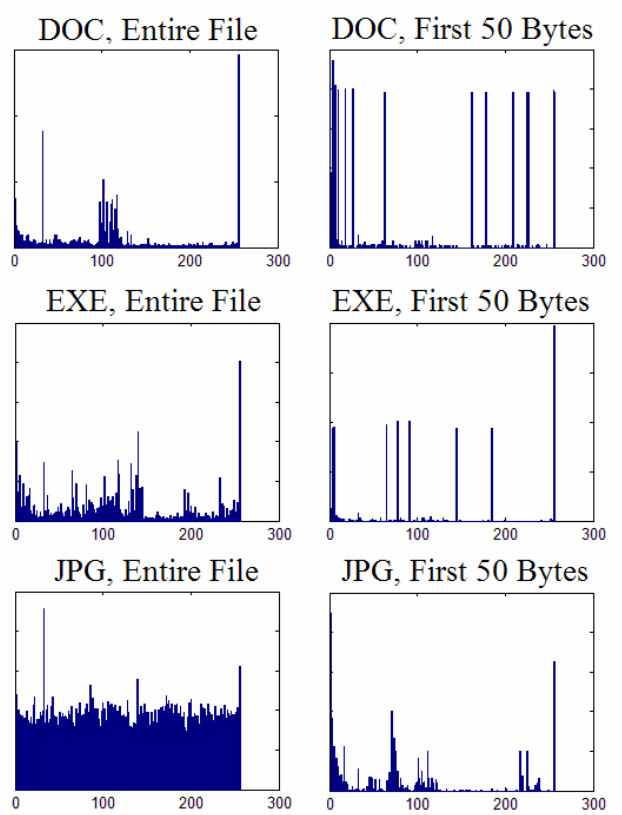


Figure 3: The byte value distributions of entire file (left column) and the first 50 bytes (right column) of the same file types. X-axis: bytes from 0 to 255, Y-axis: normalized frequency of byte values (as a %).

Figure 3 displays the effect of truncation over a few exemplar file type models. Notice the distributions change quite noticeably from the full file detail (the scale of the histograms has also changed.) Even so, the models computed under truncation may still retain sufficient information to characterize the entire class of files to distinguish different file types. In the next section, we present the results of experiments on both truncated and non-truncated files to test this conjecture.

## 2. Centroids

There are good reasons why some file types have similar distributions. Figure 4 compares MS Office formats (Word, PowerPoint, and Excel). The formats are similar, and the technique presented in this paper would certainly not be sufficient to distinguish the different sub-types from one another. However, it may be the case that any one of the models, or all of them at once, can be used to distinguish any MS Office document from, say, a virus.

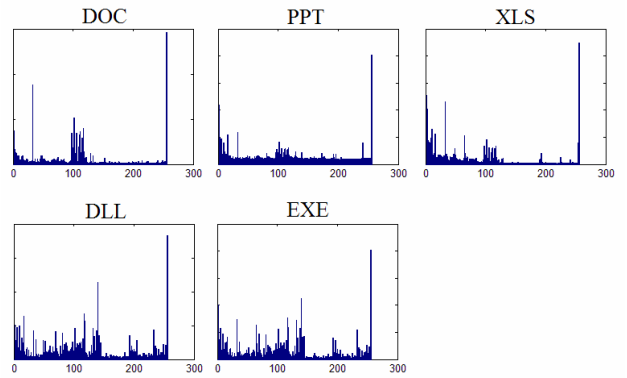


Figure 4: The bytes distribution of DLL and EXE files. X axis: bytes from 0 to 255, Y axis: normalized frequency of byte values (as a %).

The second row of Figure 4 presents another example of how two different file extensions have similar 1-gram distributions. These types should be grouped together as a logically equivalent file type, here .DLL's and .EXE's.

On the other hand, files with the same extension do not always have a distribution similar enough to be represented by a single model. For example, .EXE files might be totally different when created for different purpose, such as system files, games, or media handlers. Another example is documentation files that may contain a variety of mixed media. Thus, an alternative strategy for representing files of a particular type is to compute "multiple models". We do this via a clustering strategy. Rather than computing a single model $M_A$ for files of type A, we compute a set of models $M^k_A$, $k>1$. The multiple model strategy requires a different test methodology, however. During testing, a test file is measured against all centroids to determine if it matches at least one of the centroids. The collection of such centroids is considered a fileprint for the entire class. The multiple model technique creates more accurate models, and separates foreign files from the normal files of a particular type in more precise manner.

In the experiments reported here, the multiple models are computed by the *K-Means* algorithm under *Manhattan*

ISBN 555555555/$10.00 

*Distance*. The Manhattan Distance is defined as follows. Given two files A and B, with byte frequency distributions, *Ai* and *Bi*, i = 0,…,255, their Manhattan Distance is defined as:

$$D(\mathrm{A},\mathrm{B}) = \sum_{i=0}^{255} | A_i - B_i |$$

The *K-means* algorithm that computes multiple centroids is briefly described as follows.

1. Randomly pick *K* files from the training data set. These *K* files (their byte value frequency distribution) are the initial seeds for the first *K* centroids representing a cluster.
2. For each remaining file in the training set, compute the Manhattan Distance against the *K* selected centroids, and assign that file to the closest seed centroid.
3. Update the centroid byte value distribution with the distribution of the assigned file.
4. Repeat step 2 and 3 for all remaining files, until the centroids stabilize without any further substantive change.

The result is a set of *K* centroid models, $M^k_A$ which are later used in testing unknown files.

## III. File Type Classification and Anomaly Detection: Experimental Results

In this section we describe several experiments to test whether fileprints accurately classify unknown files or validate the presumed type of a file. The experiments performed include tests of single models computed for all files of a single type, multiple models computed over the same type, and models computed under truncation of files. We report the average accuracy over 100 trials using cross validation for each of the modeling technique.

### *A. File Sets*

We used 2 groups of data sets. The first data set includes 8 different file types, EXE, DLL, GIF, JPG, PDF, PPT, DOC and XLS. Models for each were computed using 100 files of each type. The files were collected from the internet using a general search on *Google*. For example, .PDF files were collected from Google by using the search term ".pdf". In this way, the files can be considered randomly chosen as an unbiased sample. (The reader can also repeat our experiments since the same files may be available through the same simple search.)

In our earlier experiment, we found that EXE and DLL have essentially the exact same header information and extremely similar 1-gram distributions. They are used to similar purpose in MS system. We consider that they are in the same class in this paper. The contents of MS Office file types are also similar (see Figure 4). They have the same header, which is "D0 CF 11 E0 A1 B1 1A E1". We thus assign all files of DOC, PPT and XLS files as a single class represented by .DOC in the figures below.

The files vary in size, each are approximately from 10K to 1MB bytes long. To avoid a problem of sample bias, we only compare files with similar size in the following experiments. For example, a 100K bytes file can be compared to a 200K file, but not a 1MB bytes file.

### *B. File Type Classification*

1. One-centroid file type model accuracy

In this section, we seek to determine how well each fileprint accurately identifies files of its own type using both of the entire and truncated content of the files in question.

For each file type *x*, we generated a single (one-centroid) model $M_x$. For example, we computed $M_{pdf}$ by using 80% of the collected PDF files. Since we had 8 different types of files (EXE and DLL are considered as one type, and DOC, PPT, and XLS are in one group), we generated 5 models totally. The rest of 20% files of each type are used as the test data.

In the truncated cases, we modeled and tested the *first 20, 200, 500 and 1000 bytes* of each file. This portion includes the "magic numbers" of the file type if it exists. Such analysis was used to establish a baseline and determine whether all the files tested in question contain essentially common header information.

The results are quite amazing. There was only a few misclassified file when we used truncated files. The classification accuracy results are shown in the top row of Table 1. In the row of 20 and 200 bytes, the results are almost perfect. There are some common problems. First, image, GIF and JPG, types are sometimes similar. The second, document files (PDF and MS office types) may include images. These may also cause misclassification error. The last, PDF files (with or without images) may be classified to the GIF category.

In cases where file boundaries are easily identifiable, it is rather straightforward to identify the file type from header information alone. This serves as a baseline and a first level of detection that should work well in practice. However, we next turn our attention to the more general case where header information is damaged or missing or purposely replaced to avoid detection of the true file type. We thus extend the analysis to the entire file content.

| **One-centroid file type classifying accuracy** | | | | | | |
|---|---|---|---|---|---|---|
| Truncation | EXE | GIF | JPG | PDF | DOC | AVG. |

ISBN 555555555/$10.00 

| Size | | | | | | |
|---|---|---|---|---|---|---|
| 20 | 98.9% | 100% | 99% | 100% | 98.3% | 98.9% |
| 200 | 98.3% | 91.1% | 97% | 82.8% | 93.7% | 93.6% |
| 500 | 97% | 97% | 93.4% | 80.4% | 96.7% | 94.3% |
| 1000 | 97.3% | 96.1% | 93.5% | 83.4% | 82.6% | 88.2% |
| All | 88.3% | 62.7% | 84% | 68.3% | 88.3% | 82% |
| | | | | | | |
| **Multi-centroids file type classifying accuracy** | | | | | | |
| Truncation Size | EXE | GIF | JPG | PDF | DOC | AVG. |
| 20 | 99.9% | 100% | 98.9% | 100% | 98.8% | 99.4% |
| 200 | 97% | 98.3% | 96.6% | 95% | 97.2% | 96.9% |
| 500 | 97.2% | 98.4% | 94.8% | 90% | 96.9% | 96% |
| 1000 | 97% | 95.1% | 93.5% | 90.7% | 94.5% | 94.6% |
| All | 88.9% | 76.8% | 85.7% | 92.3% | 94.5% | 89.5% |
| | | | | | | |
| **Classifying accuracy using exemplar files as centroids** | | | | | | |
| Truncation Size | EXE | GIF | JPG | PDF | DOC | AVG. |
| 20 | 100% | 100% | 100% | 100% | 98.9% | 99.6% |
| 200 | 99.4% | 91.6% | 99.2% | 100% | 98.7% | 98.2% |
| 500 | 99% | 93.6% | 96.9% | 99.9% | 98.5% | 98% |
| 1000 | 98.9% | 94.9% | 96.1% | 86.9% | 98.6% | 96.4% |
| All | 94.1% | 93.9% | 77.1% | 95.3% | 98.9% | 93.8% |

Table 1: The average accuracy of file type classifying test. First Column: the truncation size, first 20, 200, 500 1000 byte, and the entire file. Other Columns: "EXE" represents the group which includes .EXE and .DLL. "DOC" represents the group which includes .DOC, .PPT and .XLS. "AVG." represents the overall performance.

2. Multi-centroids for classifying file types

The next experiment tests the multi-centroid model. Recall, in this strategy rather than building one model for each file type, we compute multiple models by *K*-means clustering of example files into separate centroids. The union of these centroids represents the entire file type.

We generated $K$ models for each of the types of files, $M^k_{exe}$, $M^k_{doc}$, $M^k_{pdf}$, for example. If $K$ = 10, a total of 50 models (5 groups of test files) are tested using Mahalanobis Distance to determine the closest file type model. The results are shown in the middle row of Table 1.

Compare each of these results of the multi-centroids test to the previous one centroid case. The results are better. We also tested several sizes of K. Basically, the results are similar.

3. Exemplar files used as centroids

We may extend the multi-centroids method without using *K*-means. In this experiment we test each file against the distributions of a randomly chosen set of exemplar files. The same technique was used as described in the previous tests, but here we randomly choose 80% of the files as the representative samples of their file type. The other 20% of the files are test files. In this case we compare the 1-gram distribution of an individual file and hence there is no variance computed. We thus cannot apply Mahalanobis, and instead use *Manhattan Distance*.

For concreteness, assume we had $N$ files of type x, $M^i_x$, and $M$ files if type y $M^j_y$ where i = 1, 2,…$N$ and j = 1, 2,…$M$. Then, for each test file $F_k$, we compute the Manhattan Distance against each $M^i_x$, and $M^j_y$.

We record the smallest distance of $F_k$ to each of the training files. If the closest file was of type x, $F_k$ was classified as that type. The results are displayed in the bottom of Table 1. In general, the results are better than both of the previous two methods. The average accuracies of all the three methods are shown in Figure 5.

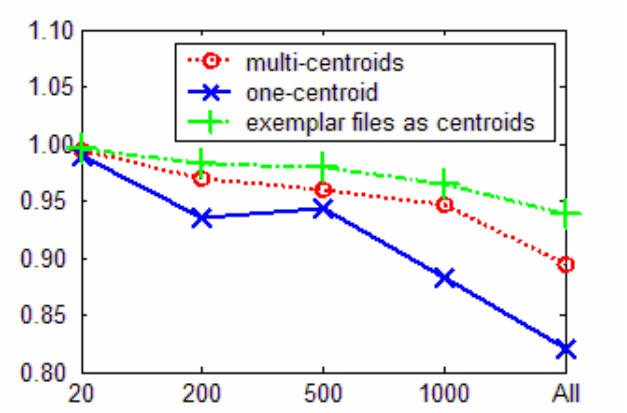


Figure 5: The classification accuracy -- comparison of three different methods. X-axis: Size of truncation (in bytes). Y-axis: accuracy.

## IV. Conclusion and Future Work

In this paper, we demonstrate the 1-gram binary distribution of files for different file types. The experiments demonstrate that every file type has a distinctive distribution that we regard as a "*fileprint*". This observation is important. The centroid models representing the byte value distributions of a set of training files can be an effective tool in a number of applications including the detection of security policy violations. Techniques that may be used by attackers to hide their malware from signature-based systems will have a tougher time being stealthy to avoid detection using these techniques.

Moreover, we found that the truncated modeling technique performs as well if not better than modeling whole files, with superior computational performance. This implies real-time network-based detectors that accurately identify the type of files flowing over a network are achievable at reasonable cost.

As future work, a number of interesting alternative

ISBN 555555555/$10.00 

modeling techniques should be explored. The truncation to the tail of a file might be interesting to determine if common shared files are infected. Furthermore, as noted, several of the file types (.DOC, .PPT and .XLS) are each so similar to a single type, MS Office. What features may be available to tease these sub-types apart? We believe bigram models are a natural extension to explore for this purpose. We have also tested these techniques comparing normal Windows OS files (both groups are EXE files) against a collection of viruses and worms. The results are quite good but also preliminary. A wider collection of test sets is required which is part of our ongoing work.

## V. ACKNOWLEDGEMENTS

This work has been partially supported by an SBIR subcontract entitle "Payload Anomaly Detection" with the HS ARPA division of the Department of Homeland Security.

## VI. REFERENCES

[1] FileAlyzer, http://www.safer-networking.org/en/filealyzer/index.html

[2] FILExt – the file extension source http://filext.com/

[3] C. Nachenberg. "Polymorphic virus detection module." United States Patent # 5,826,013, October 20, 1998.

[4] P. Szor and P. Ferrie. "Hunting for metamorphic". In Proceedings of Virus Bulletin Conference, pages 123 – 144,September 2001.

[5] Ke Wang, Salvatore J. Stolfo. "Anomalous Payload-based Network Intrusion Detection". RAID, Sept., 2004.

[6] McDaniel and M. Hossain Heydari. "Content Based File Type Detection Algorithms." 6th Annual Hawaii International Conference on System Sciences (HICSS'03)

[7] Matthew G. Schultz, Eleazar Eskin, and Salvatore J. Stolfo. "Malicious Email Filter - A UNIX Mail Filter that Detects Malicious Windows Executables." Proceedings of USENIX Annual Technical Conference - FREENIX Track. Boston, MA: June 2001.

[8] Jeremy Kolter and Marcus A. Maloof. "Learning to Detect Malicious Executables in the Wild." ACM SIGKDD, 2004

[9] M. Damashek. "Gauging similarity with n-grams: language independent categorization of text." Science 1995

ISBN 555555555/$10.00 