File Type Analysis Using Signal Processing Techniques and Machine Learning vs. file Unix Utility for Forensic Analysis

Serguei A. Mokhov and Mourad Debbabi
Concordia Institute for Information Systems Engineering
Faculty of Engineering and Computer Science
Concordia University, Montréal, Québec, Canada
{mokhov, debbabi@ciise.concordia.ca}

Abstract:
The Unix file utility determines file types of regular files by examining usually the first 512 bytes of the file that often contain some magic header information or typical header information for binary files or common text file fragments; otherwise, it defers to the OS-dependent stat() system call. It combines that heuristics with the common file extensions to give the final result of classification. While file is fast and small, and its magic database is “serviceable” by expert users, for it to recognize new file types, perhaps with much finer granularity it requires code and/or magic database updates and a patch release from the core developers to recognize new file types correctly. We propose an alternative file-like utility in determining file types with much greater flexibility that can learn new types on the user’s side and be integrated into forensic toolkits as a plug-in that relies on the file-like utility and uses signal processing techniques to compute the “spectral signatures” of file types. We present the work-in-progress of the design and implementation of such a tool based on MARF’s collection of algorithms and the selection of the best combination and the integration of the tool into a forensic toolkit to enhance the tool, called fileType with the automatic machine learning capabilities of the new file types. We compare the advantages and disadvantages of our approach with the file utility in terms of various metrics and apply the new tool to learn known stego files to attempt to classify potential unknown stego files and compare the results with stegdetect.

1 Introduction

We introduce the topics and tools that are central for this research (in their adequacy) along with the motivation for this work and considered techniques. The first version of a functioning proof-of-concept prototype is expected to be released as open-source at the IMF’08 conference in September 2008.
The main motivation for this work is to be able to determine automatically file types and being very flexible in learning new file types, perhaps with stego information and others, automatically, and inclusion of such a tool in Java-based forensic toolkits. For a large part the common Unix file utility does a very good job, but is inflexible in self-learning of the new file types requiring the developers or advanced users and the like to construct and contribute “magic” entries to its database. file in particular not very informative at its last stage of recognition saying simply “data” when nothing else matches – this is where we are trying to especially improve in our analysis using classical machine learning techniques. We perform our implementation in Java, to ease the integration as a plug-in to exiting toolkits and other Java-based technologies.

1.2 file

The file utility [DGC+08, DGC+07] uses algorithmic approach and ruleset wired into its code base to determine file types. It looks for magic numbers at certain offsets, commons structures, ASCII string patterns, etc. to determine file types. file tests each argument in an attempt to classify it. There are three sets of tests, performed in this order: file system tests, magic number tests, and language tests. By default, the first test that succeeds causes the file type to be printed [DGC+07]. It has a “magic” file database typically in /usr/share/file/magic that advanced users can customize to their needs and put into their home directories. The magic file contains patterns and offsets that map to the file type and description. Such entries can be in plain text or compiled versions and are usually contributed by people, such as developers and others in the know of the details of a particular file type. The official database is updated typically with file releases.

1.3 Digital Signatures vs. Spectral Signatures

Traditional digital signatures, such as MD5, SHA1, or any RSA-, DSA-, El-Gamel or any other PKI-based digital signatures are not applicable to our task due to their requirement being highly collision-resistant, so they cannot be used in a fuzzy-like imprecise matching approach where one can have a degree of similarity or difference – as a general rule if a single bit in the particular file changes, the entire signature should change, which is not very useful in our context so we put the cryptographic digital signatures to rest. Instead, we embark onto “spectral” signatures of signal-processing techniques, such as linear-predictive coding (LPC) and Fast Fourier Transform (FFT) combined with machine-learning techniques to classify our file types with some degree of likelihood of a given file type.
1.4 MARF

MARF Overview. MARF is an open-source collection of pattern recognition APIs and their implementation for (un)supervised machine learning and classification, including biometric forensic identification, written in Java [MCSN03, The08, Mok08b, Mok08a, Mok08c, Mok07]. As one of its roles, it serves as a testbed to verify common, found in literature, and novel algorithms for the sample loading, preprocessing, feature extraction, training and classification stages. In this role MARF provides researchers with a tool for the practical comparison of the algorithms in a homogeneous environment and allows for the dynamic module selection based on the wide array of configuration options supplied by applications. Within few years MARF accumulated a fair number of implementations for each of the pipeline stages allowing reasonably comprehensive comparative studies of algorithm combinations, studying their behavior and other properties when used for various pattern recognition tasks. MARF, its derivatives, and applications were also used beyond audio processing tasks, as in this work, due to the generality of the design and implementation in [Mok06, MHL07, Mok08d, Mok08f] and several other works.

![Figure 1: MARF’s Pattern Recognition Pipeline](image)

Pattern Recognition Pipeline. The conceptual pattern recognition pipeline design presented in Figure 1 depicts the core of the data flow and transformation between the stages in MARF [The08, MCSN03, Mok07, Mok08b]. The inner boxes represent most of the available concrete module implementations or stubs. The grayed-out boxes are either the stubs or partly implemented. The white boxes signify implemented algorithms. Generally, the classical pattern recognition process starts by loading a sample (e.g. an audio recording,
Figure 2: MARF’s Pattern Recognition Pipeline Sequence Diagram
text, image file, or virtually any regular file), preprocessing it somehow (e.g. normalization and filtering out noisy and “silent” data), then extracting the most prominent features, and, finally either training the system such that it learns a new set of features of a given subject or actually classifies what/who the subject is. The outcome of training is either a collection of some form of feature vectors or their mean or median clusters [Mok08c], called training sets, which are stored per every learned subject. The outcome of classification is an instance of the ResultSet data structure, which is a sorted collection of IDs (int) and their corresponding outcome values (double); the sorting is done from most likely outcome to least likely. The most likely one is the ultimate outcome to be interpreted by the application. Some of the details of such processing of classification are illustrated on the actual sequence of events and method calls within the main MARF module is shown in Figure 2. MARF is designed to be a standalone marf.jar file required to be usable and has no dependencies on other libraries. Optionally, there is a dependency for debug versions of marf.jar when JUnit [GB04] is used for unit testing. MARF has an internal compiler support, which by itself is unrelated to the pattern recognition pipeline but helps with the natural language parsing and forensic biometric tasks [Mok08b, Mok08a, Mok08c]. In our work here we rely on the same pattern recognition pipeline of MARF presented in Figure 2.

1.5 Forensic Toolkits

There are several forensic toolkits in the open-source, academic, and commercial worlds. We project to have our work included into one or more of those either as a plug-in if the host environment is in Java like Forensic Toolkit as JPF Plug-ins [DAS+08, AD07, ADSS07], Ftklipse [LMBT08], or others [htt06], or a standalone tool by itself for inclusion into more general forensic toolsets for binary data analysis extracted in files for preliminary classification file data alongside stegdetect [Pro04] and many others. We are considering to make it work with the Linux Sleuth kit and commercial tools, such as FTK, Encase as a part of our future work.

2 Methodology

In this work, we focus on our methodology and techniques to determine file types. We present an approach using the MARF’s collection of algorithms to determine file types in various ways and compare them using signal processing and NLP techniques, both supervised and unsupervised machine learning, and various file format loaders. MARF and its application SpeakerIdentApp [MSC+08] were shown to be used as a proof-of-concept for biometric forensic analysis of the phone-quality audio recordings to classify the identities of speakers irrespective of what speakers say on voice recordings, their gender, and spoken accent [Mok08b, Mok08a, Mok08c]. We adopt the MARF’s pattern recognition pipeline, the SpeakerIdentApp application, and the magic database of file to be used together, in our new application called FileTypesIdentApp, which has a shorthand invocation of fileType.
MARF conveniently has a class, ByteArrayFileReader that can read a file from a file system or an URI (or any Reader or InputStream for that matter). We employ this class to read our file data (either first 512 bytes or entire file as options) and the values of the array become “features” for classification (“spectral” or otherwise). We then may optionally do the regular signal pattern recognition techniques of preprocessing and feature extraction to remove all the unwanted “noise” and “silence” and extract more discriminating features.

The Raw preprocessing and RawFeatureExtraction modules are used to allow for “no-preprocessing” and feature extraction in a bypass mode (as a no-op) such that the whole testing can be done in a consistent pipelined manner. In the final application however, if this approach is preferred, it can be avoided altogether by going directly to the Classification stage if the run-time interactive performance is at issue (doubtly so with the modern hardware).

The FileTypesIdentApp application, a.k.a file type, is capable of understanding some of the file’s options [DGC+07], and the work is under way to be able to experiment with file’s magic database. file type, as its own database that it can learn throughout its lifetime automatically using machine learning techniques. As a more advanced experiment in the works, we train our system on known steganographic files in an attempt to classify the “unseen” stego files automatically. We compare the test runs of file and our applications on a set of known files and unknown files, and compile the accuracy statistics. The statistics of the algorithm combinations tried and their recognition accuracy performance along with the run-time are stored in a comma-separated values (CSV) file, per each major technique (see further).

2.1 Related Work

In part our methodology has some similarities in common with the related work on automatic classification of new, unknown malware and malware in general such as viruses, web malware, worms, spyware, and others where AI pattern recognition and expert system techniques are successfully used for automatic classification [RM08, BOA+07, PMM+07, SEZS01, SXC+04, GH99, CXZH07, SML07, CTS+07, HJ07, HRSS07, Sue07].

2.2 Training

We perform two-phase learning in our approach: initial and refining. The initial one is to collect the database of known types in the MARF’s understood format using a Perl script, collect-file-types-meta, wrapped around the classical file utility: we traverse a file system from the root directory, search for regular files (i.e. in the first round we ignore special files, directories, named pipes, block devices, etc.), produce a hash by file type mapping to the the list of file names as “samples” for training and testing. We then take the training set and test it on another, unrelated file system. The filenames
collected only for the purpose of preventing to train on the same file more than once; this is an internal requirement of MARF [The08]. After having built the meta-database in the MARF-compatible format, we train the system regularly, which represents the standard training phase. The refining phase is when we add new file types not found in file’s magic database (that are usually classified as “data” by file) and we re-train our tool on the new types. The latter part is often manual.

2.2.1 n-grams

The n-gram models come from the natural language processing (NLP) and cryptanalysis of classical ciphers techniques, where languages (natural or programming or binary) can be identified by computing frequencies of n characters occurring one after another. These serve as “features” of a language. This creates a relatively short “dictionary” (a fixed-size matrix) of n-grams where a typical n is usually between 1–3. Since the statistical counts can produce sometimes sparse matrices (the dictionaries), they are smoothed using typical statistical smoothing likelihood estimate techniques. The default processing model is unigram – i.e. a sequence of single bytes are considered to form probabilities and weights for the learned feature vectors. The other two, slower and more demanding in space/time requirements, but more accurate, are bi-gram and tri-gram models that consider sequences of two and three bytes respectively. We use the n-gram models as feature learning and extracting tools and for refinement of the plain-text file types as well as binary sequences for comparison with the other techniques. n-gram modules of MARF add another degree or dimension to our analysis and as a side-effect on MARF required adjustments to be “first-class” citizens within the MARF’s pipeline, which they weren’t prior to this work, so we augment MARF’s code based to make it so.

2.2.2 Meta-Data

At the present moment, the FileTypeIdentApp application (similarly to its predecessor applications such as SpeakerIdentApp [MSC+08], LangIdentApp [Mt08], and WriterIdentApp [Mok08f, Mok08e]) manages the training and testing metadata as a CSV file. Each entry in there is a tuple of four fields:

1. ID – a unique integer to represent the file type within the system.
2. Name – human description of the ID (the file type).
3. A set of filenames of training files.
4. A set of filenames of testing files.

The bulk composition of such a file we build via a script by traversing a host operating system in search for regular files, invoking file on them, capturing the output as a Name, and filling in the training and testing sets as absolute pathnames. The unique ID is later generated for each Name in a sequence using the mentioned Perl script,
collect-file-types-meta. We then manually add known stego files (through a “scripted” command-line) for training and testing from various projects, such as honeynet [Hon03a, Hon02, Hon03b]. We also manually review some file types that were knowingly misclassified by file.

2.2.3 strings

As an interesting feature of reduction of data volume to test, we run the strings [Var06] utility to extract anything with string patterns from binary files prior running the MARF training and testing sequences – to determine if this has any performance and accuracy impact on the fileType's learning and classification using the previously mentioned signal processing and NLP techniques. Something similar is being done in the mentioned automated malware detection work, except for us we are being more general [RM08, Sue07], to cover all file types, perhaps containing virii or stego information with them as well.

2.2.4 Magic Database

Another ongoing effort is being made is to determine if we can use file’s magic database patterns for training as a shorthand to speed up the training process and what type of accuracy we can hope to achieve for. This part of work is however in the very preliminary stages, and is not expected to work in the first release of fileType.

2.3 Testing

Baseline testing is performed on the the same file system we were trained on to make sure we do not deviate from the known types; in itself however, it is considered “cheating”. To further validate the trained-on system we move on to a different system and run the classification there via a script using fileType’s and file's output for comparison. For example, we move on from one Linux distribution to another doing such a file analysis (filtering out the files that do not exist on either system). Then we test on Windows native and Cygwin [Var08], an then the Mac OS X. We test multiple combinations of the algorithms of MARF and compile their statistics to select the better algorithm combination for the task. Then, we do misclassification testing where file was wrong and where we were wrong to compare the accuracy of the file analysis.

3 Limitations and Drawbacks

In the current implementation there are several drawbacks and limitations that we are to eliminate or reduce as a part of the future work. These are listed in part as follows:

- The presented here technique of machine learning and the tool are more effective to
use with the regular files only, and are not suited for special files and devices that are file-system specific. For those, we rely on the same approach as \texttt{file} does using the \texttt{stat()} system call [Var94], but this is no longer machine learning, so we effectively defer such tasks to \texttt{file} that does it better than us.

- “Poisoning” of the trained data with incorrect training information can deteriorate the recognition accuracy of the tool. The “poisoning” can be either accidental (local) or malicious (system-wide) by supplying a file of one type for training, but telling it is another. This is a general problem with any machine-learning tools and applications. A way of dealing with this partly is to validate each incoming training samples by classifying them first and comparing with the class specified for training and in the case of mismatch, report to the user of a potential problem; in the safe mode refuse to train in the case of mismatch. This will prevent the accidental training on the wrong data for misclassification. The latter only partly solves the problem, as the system can be cheated at the beginning when the new file type being inserted for the first time and is mistrained on.

- To be seriously considered in a real investigation toolset and environment, legally, the tool has to be proved to be correct in its design and implementation as well as components it relies on, such as MARF. We plan on using JML [Lea07, LC06, ea05] and Isabelle [PN07] for this task later in the project.

4 Conclusion

The utility presented here, \texttt{fileType} is to be released as open-source, just as \texttt{file} and MARF are. It will be made available for download and released at the conference with the full demonstration of the results and statistics for Linux, Windows, and MacOS X files and possibly others. We believe spectral signatures and machine learning techniques are very beneficial for file type analysis, especially when there is a need to bulk-preprocess a large collection of files for preliminary classification of “files of interest” on suspect’s hard drive, etc. with the easier plug-in-like integration of the tool into Java-based plug-in frameworks, such as JPF, Eclipse, and others.

5 Future Work

The future work will focus on resolving of the known and unknown limitations and drawbacks as well as the few items below:

- Integration with Ftklipse [LMBT08].
- Integration with JPF [DAS+08, AD07, ADSS07].
- Export of Forensic Lucid expressions [MPD08, MP08] for case analysis.
- Integration with JDSF [MHLR07, MHLR08, Mok08d].
6 Acknowledgments

This research work was funded in part by the Faculty of Engineering and Computer Science of Concordia University, Montreal, Canada. We also acknowledge the reviewers who took time to constructively review this work.

References


[Var06] Various Contributors. strings – find the printable strings in a object, or other binary, file. [electronic], 2006. man strings(1).
