Learning Attack Features from Static and Dynamic Analysis of Malware

Ravinder R. Ravula, Kathy J. Liszka, and Chien-Chung Chan

Department of Computer Science, University of Akron, Akron, OH, U.S.A.
{liszka,chan}@uakron.edu

Abstract. Malware detection is a major challenge in today's software security profession. Works exist for malware detection based on static analysis such as function length frequency, printable string information, byte sequences, API calls, etc. Some works also applied dynamic analysis using features such as function call arguments, returned values, dynamic API call sequences, etc. In this work, we applied a reverse engineering process to extract static and behavioral features from malware based on an assumption that behavior of a malware can be revealed by executing it and observing its effects on the operating environment. We captured all the activities including registry activity, file system activity, network activity, API Calls made, and DLLs accessed for each executable by running them in an isolated environment. Using the extracted features from the reverse engineering process and static analysis features, we prepared two datasets and applied data mining algorithms to generate classification rules. Essential features are identified by applying Weka's J48 decision tree classifier to 1103 software samples, 582 malware and 521 benign, collected from the Internet. The performance of all classifiers are evaluated by 5-fold cross validation with 80-20 splits of training sets. Experimental results show that Naïve Bayes classifier has better performance on the smaller data set with 15 reversed features, while J48 has better performance on the data set created from the API Call data set with 141 features. In addition, we applied a rough set based tool BLEM2 to generate and evaluate the identification of reverse engineered features in contrast to decision trees. Preliminary results indicate that BLEM2 rules may provide interesting insights for essential feature identification.

Keywords: Malware, Reverse Engineering, Data Mining, Decision Trees, Rough Sets.

1 Introduction

Malware, short for malicious software, is a sequence of instructions that perform malicious activity on a computer. The history of malicious programs started with "Computer Virus", a term first introduced by [6]. It is a piece of code that replicates by attaching itself to the other executables in the system. Today, malware includes viruses, worms, Trojans, root kits, backdoors, bots, spyware, adware, scareware and any other programs that exhibit malicious behaviour.

Malware is a fast growing threat to the modern computing world. The production of malware has become a multi-billion dollar industry. The growth of the Internet, the

A. Fred et al. (Eds.): IC3K 2011, CCIS 348, pp. 109–125, 2013.

advent of social networks and rapid multiplication of botnets has caused an exponential increase in the amount of malware. In 2010, there was a large increase in the amount of malware spread through spam emails sent from machines that were part of botnets [12]. McAfee Labs have reported 6 million new botnet infections in each month of 2010. They also detected roughly 60,000 new malware for each day of 2010 [13]. Symantec discovered a daily average of 2,751 websites hosting malware in January 2011 [14]. Antivirus software, such as Norton, McAfee, Sophos, Kaspersky and Clam Antivirus, is the most common defense against malware. The vendors of these antivirus programs apply new technologies to their products frequently in an attempt to keep up with the massive assault. These programs use a signature database as the primary tool for detecting malware. Although signature based detection is very effective against previously discovered malware, it proves to be ineffective against new and previously unknown malware. Malware programmers bypass the known signatures with techniques like obfuscation, code displacement, compression and encryption. This is a very effective way to evade signature based detection. Antivirus companies are trying hard to develop more robust antivirus software. Some of the techniques include heuristics, integrity verification and sandboxing. However, in practice, they are not really very effective in detecting new malware. We are virtually unprotected until the signature of each new threat is extracted and deployed.

Signature detection is mostly accomplished using manual methods of reverse engineering. This is timely and work intensive. With the staggering number of malware generated each day, it is clear that automated analysis will be imperative in order to keep up. Hence, we cannot depend solely on traditional antivirus programs to combat malware. We need an alternative mechanism to detect unidentified threats.

In an effort to solve the problem of detecting new and unknown malware, we have proposed an approach in the present study. The proposed approach uses reverse engineering and data mining techniques to classify new malware. We have collected 582 malicious software samples and 521 benign software samples and reverse engineered each executable using both static and dynamic analysis techniques. By applying data mining techniques to the data obtained from the reverse engineering process, we have generated a classification model that would classify a new instance with the same set of features either as malware or a benign program.

The rest of the paper is organized as follows. Section 2 discusses previous work based on detection of malware using data mining techniques. Section 3 presents the reverse engineering techniques used in our work. Section 4 explains the data mining process and the machine learning tools we used for the experiments. Here we present and discuss the results and finally, section 5 concludes the study and suggests possible future work.

2 Literature Review

Significant research has been done in the field of computer security for the detection of known and unknown malware using different machine learning and data mining approaches.

A method for automated classification of malware using static feature selection was proposed by [8]. The authors used two static features extracted from malware and benign software, Function Length Frequency (FLF) [7] and Printable String Information

(PSI) [23]. This work was based on the hypothesis that "though function calls and strings are independent of each other they reinforce each other in classifying malware". Disassembly of all the samples was done using IDA Pro and FLF, PSI features were extracted using Ida2DB.

The authors used five classifiers; Naive Bayes, SVM, Random Forest, IB1 and Decision Table. The best results were obtained by AdaBoostM1 with Decision Table yielding an accuracy rate of 98.86%. It was also observed that the results obtained by combining both features were more satisfactory than using each kind of features individually.

[19] used different data mining techniques to detect unknown malware. They used three approaches for static analysis and feature identification; binary profiling, strings and byte sequences. Binary profiling was only applied to PE files. Other approaches were used for all programs.

Binary profiling was used to extract three types of features; 1) list of Dynamic Link Libraries (DLL) used by the PE, 2) function calls made from each DLL and 3) unique function calls in each DLL. The "GNU Strings" program was used to extract printable strings. Each string was used as a feature in the dataset. In the third method for features extraction, the hexdump [15] utility identified byte sequences, which were used as features.

The authors applied the rule based learning algorithm RIPPER [7] to the 3 datasets with binary profiling features, Naïve Bayes classifier to data with string and byte sequence features and finally six different Naïve Bayes classifiers to the data with byte sequence features. To compare the results from these approaches with traditional signature based method, the authors designed an automatic signature generator.

With RIPPER they achieved accuracies of 83.62%, 89.36%, and 89.07% respectively for datasets with features DLLs used, DLL function calls and Unique Calls in DLLs. The accuracies obtained with Naïve Bayes and Multi-Naïve Bayes were 97.11% and 96.88%. With the Signature method they achieved 49.28% accuracy. Multi-Naïve Bayes produced better results compared to the other methods.

In [23], the information in PE headers was used for the detection of malware, based on the assumption that there would be a difference in the characteristics of PE headers for malware and benign software as they were developed for different purposes. Every header (MS DOS header, file header, optional header and section headers) in the PE was considered as a potential attribute. For each malware and benign program, position and entry values of each attribute were calculated. In parallel, attribute selection was performed using Support Vector Machines. The dataset was tested with an SVM classifier using five-fold cross validation. Accuracies of 98.19%, 93.96%, 84.11% and 89.54% were obtained for virus, email worm, Trojans and backdoors respectively. Detection rates of viruses and email worms were high compared to the detection rates of Trojans and backdoors.

In [10], multiple byte sequences from the executables were extracted from PE files and combined to produce n-grams. Five hundred relevant features were selected by calculating the information gain for each feature. Several data mining techniques like IBk, TFIDF, Naïve Bayes, Support Vector Machine (SVM) and decision trees applied to generate rules for classifying malware. The authors also used "boosted" Naïve Bayes, SVM and decision tree learners. The boosted classifiers, SVM and IBk produced good results compared to the other methods. The performance of classifiers

was improved by boosting and the overall performance of all the classifiers was better with the large dataset compared with the small dataset.

Komashinskiy and Kotenko [11] used position dependent features in the Original Entry Point (OEP) of a file for detecting unknown malware. Decision Table, C4.5, Random Forest, and Naïve Bayes were applied on the prepared dataset. Three assumptions were made for this work. 1) Studying the entry point of the program known as Original Entry Point (OEP) reveals more accurate information. 2) The location of the byte value of OEP address was set to zero. The offsets for all bytes in OEP were considered to be in the range [-127,127]. 3) Only a single byte can be read for each position value. The dataset contained three features; Feature ID, position and byte in position.

Feature selection was performed to extract more significant features. The resulting data was tested against all classifiers and the results were compared based on ROC-area. Random Forest outperformed all the other classifiers.

A specification language was derived in [5] based on the system calls made by the malware. These specifications are intended to describe the behaviour of malware. The authors also developed an algorithm called MINIMAL that mines the specifications of malicious behaviour from the dependency graphs. They applied this algorithm to the email worm Bagle.J, a variant of Bagle malware.

Clean and malicious files were executed in a controlled environment. Traces of system calls were extracted for each sample during execution. The dependencies between the system call arguments were obtained by observing the arguments and their type in sequence of calls. A dependency graph was constructed using system calls and their argument dependencies. A sub graph was then extracted by contrasting it with the benign software dependence graph such that it uniquely specifies the malware behaviour. A new file with these specifications would be classified as malware.

The Virus Prevention Model (VPM) to detect unknown malware using DLLs was implemented by [22]. Malicious and benign files were parsed by a program "dependency walker" which shows all the DLLs used in a tree structure. The list of APIs used by main program directly, the DLLs invoked by other DLLs other than main program and the relationships among DLLs which consists of dependency paths down the tree were collected. In total, 93,116 total attributes were obtained. After preprocessing there were 1,398 attributes. Of these, 429 important attributes were selected and tested. The detection rate with RBF-SVM classifier was 99.00% with True Positive rate of 98.35% and False Positive rate of 0.68%.

A similarity measure approach for the detection of malware was proposed by [21], based on the hypothesis that variants of a malware have the same core signature, which is a combination of features of the variants of malware. To generate variants for different strains of malware, traditional obfuscation techniques were used. The source code of each PE was parsed to produce an API calling sequence which was considered to be a signature for that file. The resulting sequence was compared with the original malware sequence to generate a similarity measure. Generated variants were tested against eight different antivirus products. The detection rate of SAVE was far better than antivirus scanners.

In [2], a strain of the Nugache worm was reverse engineered in order to study its underlying design, behaviour and to understand attacker's approach for finding vulnerabilities in a system. The authors also reverse engineered 49 other malware

executables in an isolated environment. They created a dataset using features such as the MD5 hash, printable strings, number of API calls made, DLLs accessed and URL referenced. Due to the multi-dimensional nature of the dataset, a machine learning tool, BLEM2 [3], based on rough set theory was used to generate dynamic patterns which would help in classifying an unknown malware. As the size of the dataset was small, a very few number of decision rules were generated and the results were generally not satisfactory.

In another work by Ahmed et al. [1] based on dynamic analysis, spatio-temporal information in API calls was used to detect unknown malware. The proposed technique consists of two modules; an offline module that develops a training model using available data and an online module that generates a testing set by extracting spatio-temporal information during run time and compares them with the training model to classify run time process as either benign or malicious. In the dynamic analysis, spatial information was obtained from function call arguments, return values and were divided into seven subsets socket, memory management, processes and threads, file, DLLs, registry and network management based on their functionality. Temporal information was obtained from the sequence of calls. The authors observed that some sequences were present only in malware and were missing in benign programs.

Three datasets were created by combining benign program API traces with each malware type. The three datasets were combinations of benign-Trojan, benign-virus and benign-worm. They conducted two experiments. The first one studied the combined performance of spatio-temporal features compared to standalone spatial or temporal features. The second experiment was conducted to extract a minimal subset of API categories that gives same accuracy as from the first experiment. For this, the authors combined API call categories in all possible ways to find the minimal subset of categories that would give same classification rate as obtained in first experiment. For the first experiment, the authors obtained 98% accuracy with naive Bayes and 94.7% accuracy with J48 decision tree. They obtained better results with combined features as compared standalone features. The detection rate of Trojans was less compared to viruses and worms.

In the second experiment, combination of API calls related to memory management and file I/O produced best results with an accuracy of 96.6%.

In some of the above mentioned works, only static features such as byte sequences, printable strings and API call sequences were used. Though effective in detecting known malware, they would be ineffective if the attackers use obfuscation techniques to write malware. To solve this problem, some other works [1, 2] used dynamic detection methods. The work done in [1] used only dynamic API call sequences. Using only API calls may not be effective in detecting malware. In the work, malwares were reversed to find their behaviour and applied data mining techniques to the data obtained from reversing process. A very small number of rules were generated and the results were not effective as the experiments were conducted on very few numbers of samples.

Our work is different from all the above works as we combined static and behavioral features of all malware and benign software. It is an extension of the work done in [1] but it differs significantly, as we performed rigorous reverse engineering of each executable to find their inner workings in detail. We also used a large number (582 malicious and 521 benign) of samples which would facilitate determining more accurate behaviour of malicious executables.

3 Reverse Engineering

Reverse engineering malware can be defined as an analysis of a program in order to understand its design, components and its behavior to inflict damage on a computer system. The benefit of reverse engineering is that it allows us to see the hidden behavior of the file under consideration, which we can't see by merely executing it [18].

In the reverse engineering process we used both static and dynamic analysis techniques. There are many different tools available for each technique. All the tools used for our work are open source. In total, we reverse engineered 1103 PE (Portable Executable) files of which 582 were malicious executables and 521 were benign executables. All malicious executables were downloaded from Offensivecomputing.net and all benign executables were downloaded from Sourceforge.net and Brothersoft.com.

3.1 Controlled Environment

For static analysis of executables, we do not require a controlled environment. In this case, we do not run the executable to collect features. In the case of dynamic analysis, the code to run is malicious and dangerous. The environment for the reversing process must be isolated from the other hosts on the network. We apply the industry common standard, a virtual machine. Due to a strong isolation between the guest operating system in VM and host operating systems, even if the virtual machine is infected with a malware, there will be no effect of it on the host operating system.

For the analysis of malware we needed virtualization software that would allow quick backtrack to the previous system state after it has been infected by the malware. Each time a malware is executed in dynamic analysis process, it would infect the system. Analysis of subsequent malware had to be performed in a clean system. We chose VMware Workstation as virtualization software for our work.

3.2 Static Analysis

In general, it is a good idea to start analysis of any given program by observing the properties associated with it and predicting the behavior from visible features without actually executing it. This kind is known as static analysis. The advantage with static analysis is that it gives us an approximate idea of how it will affect the environment upon execution without actually being executed. However, most of the time, it is not possible to predict the absolute behavior of a program with just static analysis.

There are many different tools available that aid in static analysis of executables for example, decompilers, disassemblers, Source code analyzers and some other tools that help in extracting useful features from executables. The tools we used were Malcode Analyst Pack from idefense.com, PEiD from peid.has.it and IDA Pro Disassembler hex-rays.com.

3.2.1 Cryptographic Hash Function

A unique cryptographic hash value is associated with each executable file. This value differentiates each file from others. We started our reverse engineering process of each executable by calculating its hash value.

The reason for calculating the hash value is twofold. First, there is no unique standard for naming malware. There may be multiple names for a single piece of malware so by calculating hash value of each sample we know that all of them are indeed the same. This results in eliminating ambiguity in the reverse engineering process. The second reason is that if an executable is modified, its hash value will also be changed. That way we can identify that changes were made to the executable and thereby analyzing it to detect the changes made.

MD5, SHA1 and SHA256 are the widely used hash functions. We used Malcode Analyst Pack (MAP) tool to compute the MD5 (Message Digest 5) hash value of each PE file that we analysed.

3.2.2 Packer Detection

Malware authors employ various techniques to obfuscate the content of the malware they have written and making it unable to be reversed. Using packers is one of them. A packer is a program that helps in compressing another executable program, thereby hiding the content. Packers help malware authors hide actual program logic of the malware so that a reverse engineer cannot analyze it using static analysis techniques alone. Packers also help evade detection of the malware from antivirus programs.

In order to execute, a packed malware must unpack its code into memory. For this reason, the authors of the malware include an unpacker routine in the program itself. The unpacker routine is invoked at the time of execution of the malware and converts the packed code into original executable form. Sometimes they use multiple levels of packing to make the malware more sophisticated [9].

Detection of a packer with which a malware is packed is very important for the analysis of the malware. If we know the packer, we can unpack the code and then analyze the malware. We used the PEiD tool which is a free tool for the detection of packers. It has over 600 different signatures for the detection of different packers, cryptors and compilers. It displays the packer with which the malware is packed by simply opening the malware using PEiD. If the signature of the packer or compiler with which the malware is packed is not present in the PEiD database it will report that it didn't find any packers.

3.2.3 Code Analysis

The next step for better understanding the nature of malware is to analyze its source code. Although there are many decompilers that help in decompiling executables into high level languages, analyzing the malware by keeping the source code in low level language reveals more information. IDA Pro disassembler from DataRescue is a popular choice for the disassembly of executable program into Assembly Language.

We used the IDA Pro Disassembler for the code analysis of malware. In this step, we have gone through the assembly code of each PE file to find useful information and to understand the behavior of it. Following is the list of features that we were able to extract from the assembly code of PE files.

- Type of file from the PE header. If it was not a PE file, we discarded it.
- List of strings embedded in the code that would be useful for predicting the behavior of the PE.

- The programming language with which the PE was written.
- Compile date and time.

3.3 Dynamic Analysis

In static analysis of executables, we only analyze the static code of the executable and approximately predict its properties and behavior. We know that the authors of malware use techniques such as binary obfuscation and packing to evade static analysis techniques. To thoroughly understand the nature of the malware we cannot rely on static analysis techniques alone. If a program has to be run, the whole code must be unpacked and loaded into primary memory. Every detail of the executable program is revealed at run time regardless of how obfuscated the code is and what packer the executable is packed with [20]. In dynamic analysis, we observe the full functionality of the malware and its effect on the environment as it executes.

Tools that help in dynamic analysis of executables include debuggers, registry monitors, file system monitors, network sniffers and API call tracers. The tools we used in this step were Filemon, Regshot and Maltrap.

3.3.1 File System Monitor

When a program is executed it makes changes to the file system. The file system activity made by the program helps partly in determining its behavior. We used File Monitor (Filemon) from Microsoft Sysinternals to monitor file system activity of all the processes running on a windows system. It installs a device driver which keeps track of all the file system activity in the system. However, we only need the information related to a particular process under consideration, therefore, we can use a filter which lets us select a particular process for which we want to monitor file system activity by removing all the other processes from the list. Each file system activity made by the PE on the file system produces one line of output in the Filemon GUI window.

3.3.2 Registry Monitor

Windows Registry is a huge database hosting configuration details of the operating system and the programs installed on it. Registry entries are divided into hives and represented in a tree structure on Windows systems. Most applications use two hives frequently; HKLM and HKCU. HKLM stands for Hive Key Local Machine and contains settings of the operating system. HKCU stands for Hive Key Current User and contains configuration details for the user currently logged into the system.

Malware authors frequently use registries to infect systems. One usual technique employed is to insert an entry at the location HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\RUN so that each time system boots up the malware is executed. There is an extensive list of such keys in the Windows registry and they are used by attackers for their malicious purposes.

Regshot is a product of Microsoft Sysinternals that helps in the reverse engineering process by monitoring the Windows Registry. It lists the changes made in the

windows registry upon installation of software. We used this tool to record the changes in the windows registry made by malware and benign software in the Windows Registry.

3.3.3 Api Call Tracer

Windows API (Application Program Interface) also known as Win API, is a long list of functions that provide system level services to the user programs. Every Windows application is implemented using the Win API functions.

Keeping track of the sequence of API Calls made by an application helps in the reverse engineering process. It allows us to go through each call and thereby predicting the behavior of that software. Maltrap is a software that lists the sequence of calls made by the software while execution.

4 Malware Detection Mining

In this section, we explain our implementation of the data mining process to find patterns that would help in classifying malware from benign software.

4.1 Feature Extraction

From static analysis of each sample in the reverse engineering process we have the MD5 hash of the file, file size in bytes, the packer used (if any, a determination of whether it contains unique strings, a time stamp and the programming language used to write the file.

From the dynamic analysis, for each file, we stored a log of file system activity, registry activity and the sequence of API calls made by the sample while running.

From the file system activity log we were able to extract three important features; the decisions of whether the sample under consideration attempted to write to another file, if the sample accessed another directory and all unique DLLs accessed during execution.

From the registry activity we extracted three features; registry keys added, registry keys deleted and registry values modified. The log contains all the registry keys modified by executables with their modified values. We removed the key values and recorded only the keys.

From the API call log we extracted the unique API calls made by each sample. We combined all the unique API calls made by each file and removed duplicates. Over the entire sample space, we identified 141 unique API calls which will be used as features to create a data set. We also noted if a sample contained any URL references or attempted to access the Internet, making. With this step we completed processing of raw data for feature selection.

4.2 Data Sets

We prepared two datasets: the first dataset with 15 features, 1103 instances (582 malicious and 521 benign) and the second dataset with 141 features, 1103 instances. We

named the first dataset DRF (Dataset with Reversed Features) and the second dataset DAF (Dataset with API Call Features). Table 1 shows the list of 15 attributes and decision label in DRF. All the 141 attributes in DAF are of type binary except the decision label which is Boolean. The first three attributes File Name, File Size, and MD5 Hash do not provide useful information for classification purpose. We collected this information and retain it for tracking and other research purposes, however, they are removed from data sets used in the following experiments.

Table 1. Attributes in DRF

S.NO	Attribute Name	Type
1	FILE NAME	NOMINAL
2	FILE SIZE	NOMINAL
3	MD5 HASH	NOMINAL
4	PACKER	NOMINAL
5	FILE ACCESS	BINARY
6	DIRECTORY ACCESS	BINARY
7	DLLs	NOMINAL
8	API CALLS	NOMINAL
9	INTERNET ACCESS	BINARY
10	URL REFERENCES	BINARY
11	REGISTRY KEYS ADDED	NOMINAL
12	REGSITRY KEYS DELETED	NOMINAL
13	REGISTRY VALUES MODIFIED	NOMINAL
14	UNIQUE STRINGS	BINARY
15	PROGRAMMING LANGUAGE	NOMINAL
16	DECISION LABEL	BOOLEAN

In addition, values of attributes Registry Keys Added, Registry Keys Deleted, Registry Keys Modified, API CALLs and DLLs varies widely from very small to very large. They were discretized by using tools available in Weka. The ranges we obtained for each attribute after transforming the dataset are shown in Table 2.

Table 2. Discretized values

	ATTRIBUTE NAME	**DISCRETIZED VALUES**
1	KEYS ADDED	(-INF-1] (1-INF)
2	REGISTRY VALUES MODIFIED	(-INF-12.5] (12.5-INF)
3	API CALLS	(-INF -5.5] (5.5-22.5] (22.5-41.5] (41.5- INF)
4	DLLs	(-INF -16.5] (16.5- INF)

We prepared second dataset from DRF by replacing the discrete values with the discretized values shown in Table 2. We call it DDF (Dataset with Discretized Features).

4.3 Experimental Results

We conducted experiments on the datasets derived from reversed features: DRF and DDF, and data set derived from API call features: DAF. The J48 decision tree and Naïve Bayes algorithms in WEKA [24] were used to generate classifiers. We applied 5-fold cross validation with 80-20 splits for training and testing.

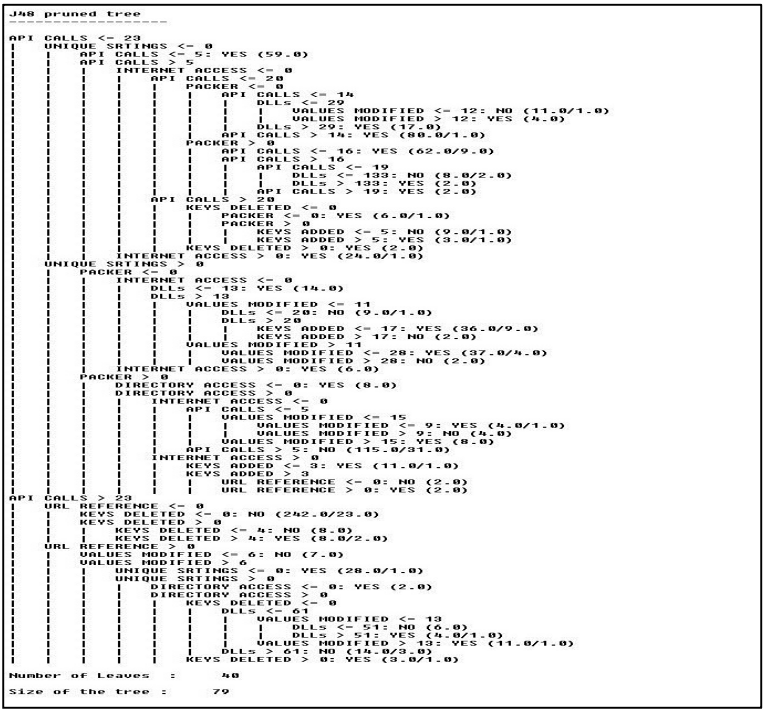

Fig. 1. J48 decision tree for DRF

Figure 1 shows that the number of API Calls made by a PE was selected as the root node of the decision tree. API Calls, Unique Strings, URL References, Internet Access, Packer, Registry Keys Deleted, Directory Access and Registry Keys Modified were the most used attributes in the classification model although we note that it used all the other attributes in the dataset.

The decision tree in Figure 2 shows that 9 attributes, API Calls, Unique Strings, URL References, Packer, Registry Keys Deleted, Registry Keys Modified, Directory Access, Registry Keys added and Internet Access were used in the classification model. In this case, the attributes DLLs and File Access were not used in the decision rules for classification.

Decision tree classifiers such as J48 are top-down learners that select an attribute to split a tree node in each step of the decision tree building process. In contrast, the machine learning algorithm BLEM2 [3] is a learner for generating if-then decision rules from bottom-up. In each step, an attribute-value pair is selected to form the condition part of a decision rule. Each rule learned by BLEM2 is minimal, and the entire

set of rules learned is also minimal. It is a revision of the LEM2 algorithm [4], which is based on rough set theory introduced by Pawlak [16]. The major difference is that rules learned by BLEM2 are associated with four factors: support, certainty, strength, and coverage. These factors can be interpreted as Bayesian probabilities derived from the training data which was first introduced in [17]. We say that an instance is covered by a rule if it matches the condition of the rule. The support of a rule is the number of training instances covered by the rule. Certainty of a rule is the conditional probability that an instance covered by the rule will have the same decision label as the rule. The strength of a rule is the probability of an instance covered by the rule. The coverage of a rule is the inverse probability that an instance of the same decision label as the rule will be covered by the rule.

```
J48 pruned tree
------------------

API CALLS = (5.5-23.5]
|   UNIQUE SRTINGS <= 0: YES (230.0/37.0)
|   UNIQUE SRTINGS > 0
|   |   PACKER <= 0: YES (84.0/22.0)
|   |   PACKER > 0
|   |   |   URL REFERENCE <= 0
|   |   |   |   INTERNET ACCESS <= 0: NO (118.0/34.0)
|   |   |   |   INTERNET ACCESS > 0
|   |   |   |   |   KEYS ADDED = (-inf-1]: YES (5.0)
|   |   |   |   |   KEYS ADDED = (1-2.5]: YES (0.0)
|   |   |   |   |   KEYS ADDED = (2.5-inf): NO (2.0)
|   |   |   URL REFERENCE > 0: YES (8.0/1.0)
API CALLS = (23.5-41.5]
|   URL REFERENCE <= 0
|   |   KEYS DELETED <= 0: NO (239.0/23.0)
|   |   KEYS DELETED > 0
|   |   |   VALUES MODIFIED = (12.5-inf)
|   |   |   |   KEYS DELETED <= 4: NO (2.0)
|   |   |   |   KEYS DELETED > 4: YES (7.0/1.0)
|   |   |   VALUES MODIFIED = (-inf-12.5]: NO (7.0)
|   URL REFERENCE > 0
|   |   VALUES MODIFIED = (12.5-inf): YES (13.0/1.0)
|   |   VALUES MODIFIED = (-inf-12.5]
|   |   |   KEYS ADDED = (-inf-1]: NO (7.0)
|   |   |   KEYS ADDED = (1-2.5]: YES (2.0/1.0)
|   |   |   KEYS ADDED = (2.5-inf)
|   |   |   |   UNIQUE SRTINGS <= 0: YES (4.0)
|   |   |   |   UNIQUE SRTINGS > 0: NO (7.0/2.0)
API CALLS = (41.5-inf)
|   UNIQUE SRTINGS <= 0: YES (20.0/1.0)
|   UNIQUE SRTINGS > 0
|   |   DIRECTORY ACCESS <= 0: YES (2.0)
|   |   DIRECTORY ACCESS > 0
|   |   |   KEYS ADDED = (-inf-1]
|   |   |   |   VALUES MODIFIED = (12.5-inf): YES (3.0)
|   |   |   |   VALUES MODIFIED = (-inf-12.5]: NO (2.0)
|   |   |   KEYS ADDED = (1-2.5]: NO (2.0)
|   |   |   KEYS ADDED = (2.5-inf): NO (16.0/4.0)
API CALLS = (-inf-5.5]: YES (102.0/8.0)

Number of Leaves  :     23

Size of the tree :     40
```

Fig. 2. J48 decision tree for DDF

In our experiment, we applied BLEM2 to generate decision rules from the DDF dataset. Rules with a minimum support of 7 were selected for analysis. There are 11 attributes selected by BLEM2 to form decision rules. Tables 3 shows the attributes selected by BLEM2 together with the frequencies of attribute-value pairs appeared in the rules. Table 4 shows the 21 decision rules generated by BLEM2 for the decision label "YES", and Table 5 shows the 13 decision rules for the decision label "NO".

From Table 3, API CALLS and REGISTRY KEYS ADDED are identified as the most relevant attributes by BLEM2. In contrast, API CALLS is selected as the root of the decision tree in Figure 2. It is also clear that URL REFERENCES is not relevant in classifying benign software. It seems that both FILE ACCESS and URL REFERENCES may not be essential due to their low frequency; however, they cannot be ignored, since they have high support and certainty factors according to rules shown in Table 4 and 5. In general, we will need to consider the Bayesian factors of BLEM2 rules to derive a threshold for identifying essential attributes. This will be part of our future works.

Table 3. Attribute-value pair frequencies in BLEM2 rules with minimum support = 7

Attribute	Attribute Name	Freq. of YES (21 rules total)	Freq. of NO (13 rules total
C1	PACKER	17	12
C2	REGISTRY KEYS ADDED	20	13
C3	REGISTRY KEYS DELETED	9	10
C4	REGISTRY VALUES MODIFIED	19	12
C5	API CALLS	20	13
C6	DLLs	11	4
C7	FILE ACCESS	2	1
C8	DIRECTORY ACCESS	8	3
C9	INTERNET ACCESS	11	13
C10	URL REFERENCES	2	0
C11	UNIQUE STRINGS	16	12

Table 4. BLEM2 rules for DDF for the decision label "YES" with minimum support = 7

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C 11	Supp	Cer	Stren	Cov
0	(-inf-1]	?	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	1	0	?	0	43	0.811	0.039	0.074
0	(2.5-inf)	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	?	?	0	30	0.909	0.027	0.052
?	?	?	?	?	?	0	?	?	?	?	26	1	0.024	0.045
1	(2.5-inf)	0	(12.5-inf)	(5.5-23.5]	(16.5-inf)	?	1	0	?	?	21	0.553	0.019	0.036
0	(-inf-1]	?	(12.5-inf)	(5.5-23.5]	(16.5-inf)	?	1	?	?	0	20	1	0.018	0.034
0	(-inf-1]	?	(-inf-12.5]	(5.5-23.5]	?	?	1	0	?	1	20	0.769	0.018	0.034
1	(2.5-inf)	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	0	17	0.654	0.015	0.029
0	(-inf-1]	?	?	(5.5-23.5]	?	?	0	?	?	?	16	1	0.015	0.028
1	(-inf-1]	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	1	1	0	?	0	15	0.455	0.014	0.026
?	(-inf-1]	?	(-inf-12.5]	(-inf-5.5]	(-inf-16.5]	?	1	?	?	0	14	1	0.013	0.024
0	(2.5-inf)	?	(-inf-12.5]	(5.5-23.5]	?	?	?	0	?	1	13	0.542	0.012	0.022
1	(2.5-inf)	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	1	13	0.236	0.012	0.022
0	(2.5-inf)	?	(12.5-inf)	(5.5-23.5]	?	?	?	?	?	0	12	1	0.011	0.021
0	(2.5-inf)	0	(12.5-inf)	(5.5-23.5]	?	?	?	?	?	1	11	0.917	0.01	0.019
0	(-inf-1]	?	(12.5-inf)	(41.5-inf)	?	?	?	?	?	?	10	1	0.009	0.017
0	(-inf-1]	0	(12.5-inf)	(5.5-23.5]	(16.5-inf)	?	1	?	?	1	10	0.769	0.009	0.017
1	(-inf-1]	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	1	10	0.2	0.009	0.017
1	(-inf-1]	0	(12.5-inf)	(5.5-23.5]	(16.5-inf)	?	?	0	?	0	9	0.9	0.008	0.016
1	(-inf-1]	?	(12.5-inf)	(-inf-5.5]	?	?	?	?	?	1	7	1	0.006	0.012
?	(2.5-inf)	?	(-inf-12.5]	(5.5-23.5]	?	?	?	1	0	0	7	1	0.006	0.012
?	(-inf-1]	?	(-inf-12.5]	(5.5-23.5]	?	?	?	1	0	?	7	1	0.006	0.012

Table 5. BLEM2 rules for DDF for the decision label "NO" with minimum support = 7

C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C 11	Supp	Cer	Stren	Cov
1	(2.5-inf)	0	(-inf-12.5]	(23.5-41.5]	?	?	1	0	?	1	94	0.99	0.085	0.18
1	(1-2.5]	0	(-inf-12.5]	(23.5-41.5]	?	?	?	0	?	1	52	0.981	0.047	0.1
1	(2.5-inf)	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	1	42	0.764	0.038	0.081
1	(-inf-1]	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	1	40	0.8	0.036	0.077
?	(-inf-1]	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	1	1	0	?	0	28	0.326	0.025	0.054
1	(2.5-inf)	0	(12.5-inf)	(23.5-41.5]	?	?	?	0	?	1	27	0.9	0.025	0.052
1	(2.5-inf)	0	?	(23.5-41.5]	?	?	1	0	?	0	27	0.9	0.025	0.052
1	(2.5-inf)	0	(12.5-inf)	(5.5-23.5]	?	?	1	0	?	1	14	0.667	0.013	0.027
0	(2.5-inf)	?	(-inf-12.5]	(5.5-23.5]	?	?	?	0	?	1	11	0.458	0.01	0.021
1	(-inf-1]	?	(-inf-12.5]	(23.5-41.5]	?	?	?	0	?	?	9	1	0.008	0.017
1	(2.5-inf)	0	(-inf-12.5]	(5.5-23.5]	(16.5-inf)	?	?	0	?	0	9	0.346	0.008	0.017
1	(1-2.5]	0	(-inf-12.5]	(23.5-41.5]	?	?	?	0	?	0	7	1	0.006	0.013
0	(2.5-inf)	?	(-inf-12.5]	(23.5-41.5]	?	?	?	0	?	1	7	1	0.006	0.013

In the experiment with the DAF data set, out of 141 attributes, 31 attributes were selected by J48 in the decision tree. The API Call "IsDebuggerPresent" is used as the root node in the tree. The most used attributes in the classification model are IsDebuggerPresent, WriteProcessMemory, RegSetValuesExW, GetVolumeInformationW, bind, CreateProcessW and Connect. While IsDebuggerPresent may seem surprising, it is actually very revealing. Malware writers often include code to check for the presence of a debugger during execution in order to detect reverse engineering attempts on their software. At that point they terminate the program in order to make analysis of their code more difficult.

The performance of the decision tree classifiers is shown in Table 6 where TP, TN, FP, FN denote True Positive, True Negative, False Positive, and False Negative rates respectively.

Table 6. Performance of J48 decision trees

Data	TP	TN	FP	FN	ROC Area	Overall Accuracy
DRF	0.793	0.809	0.191	0.207	0.843	80.09%
DDF	0.847	0.782	0.218	0.153	0.815	81.45%
DAF	0.832	0.947	0.053	0.168	0.917	89.14%

The Naïve Bayes algorithm of WEKA was applied to the same data, and the performance is shown in Table 7.

Table 7. Performance of Naïve Bayes

Data	TP	TN	FP	FN	ROC Area	Overall Accuracy
DRF	0.856	0.773	0.227	0.144	0.889	81.45%
DDF	0.865	0.836	0.164	0.135	0.912	85.07%
DAF	0.701	0.947	0.053	0.299	0.921	82.81%

From Table 6 and 7, it shows that Naïve Bayes performs slightly better than J48 tree classifier in experiments with DRF and DDF data sets, which has 15 attributes. In experiments with the DAF data, which has 141 attributes, the feature selection algorithm used in J48 provides an advantage. From our results, it is clear that discretization will improve performance for reverse engineered features as shown in both J48 and NB. In addition, feature reduction is essential in API CALLS data as indicated by the better performance of J48 over NB.

We did not evaluate the performance of BLEM2 rules in current experiments. Currently, an inference engine based on BLEM2 is under development. It will be interesting to see how the feature reduction of BLEM2 with Bayesian factors compares to J48 and NB.

5 Conclusions

In this work, the problem of detecting new and unknown malware is addressed. Present day technologies and our approach for the detection of malware are discussed. An isolated environment was set up for the process of reverse engineering and each executable was reversed rigorously to find its properties and behavior. On the data extracted from the reversing process, different data mining techniques were used to procure patterns of malicious executables and thereby classification models were generated. To test the models, new executables were supplied from the wild with the same set of features. The results thus obtained proved to be satisfactory. BLEM2 rules seem to provide insightful information for essential features identification and for developing inference engines based on Bayesian factors associated with the rules. One of our future works is to apply rough set feature reduction tools and BLEM2 inference engines to evaluate their performance in malware detection.

From analyzing the experimental results, we can conclude that finding static and behavioral features of each malware through reverse engineering and applying data mining techniques to the data helps in detecting new generation malware. Considering the rapidly increasing amount of malware appearing each day, this method of detection can be used along with current practice detection techniques.

We have reversed each strain of malware and benign executables to extract all the features we could with the help of the tools used by the computer security profession. However, we were not able to analyze the process address space of the executables in the physical memory as the memory analysis tools were released after we completed the reversing step. Analyzing the address space would reveal more interesting information about the processes and thereby analyzing their behavior more accurately.

Reversing each malware manually is a time consuming process and requires much effort with the thousands of new malware being generated. One way to cope up with this problem is to automate the whole reverse engineering process. Although there are some tools for automated reverse engineering, they do not record the full details of malware. A more specific tool that does rigorous reversing would help in combating large amounts of malware. We consider these two tasks as the future work that aid in detecting new malware more efficiently.

References

1. Ahmed, F., Hameed, H., Shafiq, M.Z., Farooq, M.: Using spatio-temporal information in API calls with machine learning algorithms for malware detection. In: AISec 2009: Proceedings of the 2nd ACM Workshop on Security and Artificial Intelligence, pp. 55–62. ACM, New York (2009)
2. Burji, S., Liszka, K.J., Chan, C.-C.: Malware Analysis Using Reverse Engineering and Data Mining Tools. In: The 2010 International Conference on System Science and Engineering (ICSSE 2010), pp. 619–624 (July 2010)
3. Chan, C.-C., Santhosh, S.: BLEM2: Leaming Bayes' rules from examples using rough sets. In: Proc. NAFIPS 2003, 22nd Int. Conf. of the North American Fuzzy Information Processing Society, Chicago, Illinois, July 24-26, pp. 187–190 (2003)
4. Chan, C.-C., Grzymala-Busse, J.W.: On the two local inductive algorithms: PRISM, and LEM2. Foundations of Computing and Decision Sciences 19(3), 185–203 (1994)
5. Christodorescu, M., Jha, S., Kruegel, C.: Mining specifications of malicious behaviour. In: Proc. ESEC/FS 2007, pp. 5–14 (2007)
6. Cohen, F.: Computer Viruses. PhD thesis, University of Southern California (1985)
7. Cohen, W.: Learning Trees and Rules with Set-Valued Features. American Association for Artificial Intelligence, AMI (1996)
8. Islam, R., Tian, R., Batten, L., Versteeg, S.C.: Classification of Malware Based on String and Function Feature Selection. In: 2010 Second Cybercrime and Trustworthy Computing Workshop, Ballarat, Victoria Australia, July 19-July 20 (2010) ISBN: 978-0-7695-4186-0
9. Kang, M.G., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proc. Fifth ACM Workshop on Recurring Malcode, WORM 2007 (November 2007)
10. Kolter, J., Maloof, M.: Learning to detect malicious executables in the wild. In: Proc. KDD 2004, pp. 470–478 (2004)
11. Komashinskiy, D., Kotenko, I.V.: Malware Detection by Data Mining Techniques Based on Positionally Dependent Features. In: Proceedings of the 2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing, PDP 2010. IEEE Computer Society, Washington, DC (2010) ISBN: 978-0-7695-3939-3
12. Mcafee.com (2010a), `http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q3-2010.pdf` (retrieved)
13. Mcafee.com (2010b), `http://www.mcafee.com/us/resources/reports/rp-good-decade-for-cybercrime.pdf` (retrieved)
14. Messagelabs.com (2011), `http://www.messagelabs.com/mlireport/MLI_2011_01_January_Final_en-us.pdf` (retrieved)
15. Miller, P.: Hexdump. Online publication (2000), `http://www.pcug.org.au/millerp/hexdump.html`
16. Pawlak, Z.: Rough sets: basic notion. International Journal of Computer and Information Science 11(15), 344–356 (1982)
17. Pawlak, Z.: Flow graphs and intelligent data analysis. Fundamenta Informaticae 64, 369–377 (2005)
18. Rozinov, K.: Reverse Code Engineering: An In-Depth Analysis of the Bagle Virus. In: Information Assurance Workshop, IAW 2005. Proceedings from the Sixth Annual IEEE SMC, June 15-17, pp. 380–387 (2005)

19. Schultz, M.G., Eskin, E., Zadok, E., Stolfo, S.J.: Data Mining Methods for Detection of New Malicious Executables. In: Proceedings of the 2001 IEEE Symposium on Security and Privacy, pp. 38–49. IEEE Computer Society (2001)
20. Skoudis, E.: Malware: Fighting Malicious Code. Prentice Hall (2004)
21. Sung, A., Xu, J., Chavez, P., Mukkamala, S.: Static analyzer of vicious executables (save). In: Proc. 20th Annu. Comput. Security Appl. Conf., pp. 326–334 (2004)
22. Wang, T.-Y., Wu, C.-H., Hsieh, C.-C.: A Virus Prevention Model Based on Static Analysis and Data Mining Methods. In: Proceedings of the 2008 IEEE 8th International Conference on Computer and Information Technology Workshops, CITWORKSHOPS 2008, pp. 288–293 (2008)
23. Wang, T.-Y., Wu, C.-H., Hsieh, C.-C.: Detecting Unknown Malicious Executables Using Portable Executable Headers. In: Fifth International Joint Conference on INC, IMS and IDC, pp. 278–284 (2009)
24. Witten, I.H., Frank, E.: Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. (2005) ISBN: 0-12-088407-0