Mac Malware Detection via Static File Structure Analysis

Elizabeth Walkup

University of Stanford
ewalkup@stanford.edu

I. Introduction

It is widely acknowledged in the security community that the current signature-based approach to virus detection is no longer adequate. More recently, antivirus software has been doing dynamic malicious behavior detection. While this is more effective, it is computationally expensive, so they cannot do very much of it or the performance of the user's computer will suffer.

Static executable analysis offers a possible solution to the problems of dynamic analysis. Static analysis looks at the structures within the executable that are necessary in order for it to run. Since these structures are mandated by the file type, they cannot be removed, encrypted (although their code contents may be), or obfuscated easily. Also, because it simply involves parsing structures, it is much less computationally expensive than dynamic analysis. Some research has already been done into static executable analysis for Windows portable executable (PE) files.

Although Mac malware (the general term for malicious software) is not nearly as prolific as those for Windows, the number is steadily increasing as Macs gain a greater market share. Kaspersky has seen the amount of Mac malware double in the last two years, and increase by a factor of ten in the last four years [10], yielding an exponential increase curve. Of particular note, the last 4 or 5 years has seen a number of advanced persistent threat (APT) malware campaigns targeted specifically at Macs. Researching tools to face these threats now ensures that we will be able to better handle the increasing threat in the future.

II. Dataset

Malware samples were gathered from open research sites like Contagio[8] and VirusTotal[9]. Benign Mach-O files (for comparison) were gathered from OS X Mavericks and open-source programs for OS X. The samples themselves are Mach-O binary executable files. This is the format of choice for OS X programs, though there are one or two other formats used in older systems. For legal reasons, open-source malware repositories are hard to find, so increasing the malware samples by any substantial amount beyond this is difficult.

The current data set consists of 420 malware samples and 1000 goodware (benign programs) samples. This data was randomly divided into a training set (roughly 75%) and a testing set (roughly 25%). By number, the samples are divided as follows:

	Training Set	Testing Set
Malware	333	87
Goodware	798	202

III. Features

The Mach-O features are pulled out of the binaries using a Python script that parses the file structures. The information is then stored in a SQLite database. Since machine learning was done using Weka [7], another Python script converts the information in the database into a Weka-supported format. The features are diverse: some are strings, some are continuous-valued numbers, some are discrete numbers, and some are boolean (0 or 1).

I. Structure Features

The script pulls 50 structural features out of the database, mostly continuous valued, size-

based statistics of various structures. There are other structural features within the database that could be used, but these were chosen to maximize information gain while decreasing model size. Using only structural features, the top model had a testing error of 7.3691%.

I.1 Mach-O File Structure

Mach-O executables[6] have three main parts: a mach header, load commands, and data. The data section is made up of the actual executable code, which can be compressed or encrypted, so no features are gathered from there. The mach header contains information about the platform the executable was built for, as well as the file type (dynamic library, etc).

The majority of the structural features come from the load commands. These contain headers for each code segment in the data, as well as information about import libraries and how to load the file in memory. Each segment header also contains information about a substructure within the segment called a section.

II. Import Features

Every executable file imports files and libraries to use within its program. These cannot be encrypted, so their names can be extracted from the binary. However, there are literally thousands of names that could be present, which is not easy to incorporate into a model. To compromise, feature selection was performed on models built solely from import features, and the top features were then incorporated back into the main set of features. When these import features were added, the testing error of the top model decreased from 7.3691% to 2.7682%, a nearly 5% improvement.

III. Feature Selection

Two methods were used to trim features: forward search selection and information gain ranking. These two algorithms had many top features in common, particularly when used to choose import features. By combing the top results from these two methods, thousands of possible import features were reduced to 60 dynamic library (dylib) imports and 117 function imports. Combined with the 50 structural features, there were 227 features for all of the models. Some of the top features are shown in Table 1.

IV. Models

For the sake of brevity, only the top five models will be described, although over 20 were tested.

I. Rotation Forest

Rotation Forest[1] builds an ensemble (or forest) of decision trees, in this case C4.5 trees. The feature set is split into K disjoint subsets. In this case, K was 75, which corresponds to roughly 3 features per set. For each classifier, principal component analysis (PCA) is run using the subset of features chosen. The coefficients found using PCA are put into a sparse rotation matrix, which is rearranged to match the chosen features. This new matrix multiplied by the training data comprises the model. To classify a new data point, the confidence for each individual model is calculated, and the output class is the one with the highest confidence. This model had a confidence factor of 0.25.

II. Random Forest

Random Forest[2], similar to Rotation Forest, is an ensemble of different decision tree models. However, instead of using a confidence measure, the different models vote on the output class. The training sets for these models are chosen randomly, but with the same distribution. In this case, there were ten decision tree models and the models used the full feature set.

III. PART

PART[3] is a rule-based algorithm. The data set is divided into several new sets. A partial pruned C4.5 decision tree is built based on each set. The leaf of this tree with the largest data coverage (meaning it classifies the most instances) is made into a rule. A leaf must cover at least two instances to be considered for a rule.

IV. LMT

LMT[4] stands for Logistic Model Tree. This is a basic decision tree except that instead of

Table 1: *Top Ten Features by Informaton Gain*

dylibs	functions	structural
mbukernel	_exit	num_imports
mbuinstrument	_system	std_segsize
QtCore	_kCFHTTPVersion1_1	file_type
wlmstrings	_umask	avg_segsize
mbustrings	_fread	std_segvmsize
MicrosoftComponentPlugin	_sysctlnametomib	min_segsize
QtGui	_CFReadStreamCreateForHTTPRequest	avg_r1
libstdc++	_IORegistryEntryFromPath	avg_segvmsize
QtNetwork	_gmtime	std_secsize
Netlib	_sysctl	avg_secalign

having classes at the leaf nodes, it has logistic regression functions. A node must cover at least fifteen instances to have child nodes. The functions at the leaf nodes model the class probability:

$$Pr(G = j|X = x) = \frac{e^{F_j(x)}}{\sum_{k=1}^{J} e^{F_k(x)}} \quad (1)$$

$$F_j = \alpha_0^j + \sum_{v \in V_T} \alpha_v^j v \quad (2)$$

where V_t is a subset of all the features, α_i's are weights, and k is the number of possible feature values.

V. IBk

IBk[5] is an instance-based learning implementation of k-Nearest Neighbors (in this case k=1). For each data point, it calculates a similarity for each cluster. That data point is then assigned to the cluster with the highest similarity.

V. Results

Many different supervised learning algorithms were evaluated to see which one could classify the data the best. The algorithms were trained on the full 227 attributes. The top ten results are listed in Table 2. A false positive is a goodware sample that was classified as malware, and a false negative is a malware sample that was classified as goodware.

Some interesting facts that can be gleaned from the data itself:

- Goodware tends to import more libraries than malware
- Malware tends to have more segments than goodware and use more varied segment flags
- Malware and goodware have the same distribution of load com- mand sizes
- Malware is more likely to use Java libraries
- Malware is more likely to use HTTP stream functions (probably for control/exfiltration)
- Goodware uses more memory access controls

VI. Discussion

The ensemble methods had the best performance, perhaps due to the complex nature of the input features or the varied data set. Since all of the import features were nominal features, decision tree models were particularly well suited for classification.

The fact that the testing error can be reduced to a few percent means that there are actually tangible differences in static structures between malware and goodware. This was expected going into the project because substantial results have been produced using a similar approach with Windows portable executable (PE) files and malware.

Table 2: *Top Ten Supervised Learning Algorithms Results*

Algorithm	Training Error	Test Error	False Positive	False Negative	ROC Area
Rotation Forest	0.2653%	2.7682%	0.02	0.046	0.994
Random Forest	0.1768%	4.1522%	0.035	0.057	0.99
PART	0.9726%	4.4983%	0.025	0.092	0.949
LMT	0.9726%	5.1903%	0.025	0.115	0.98
IBk	0%	5.1903%	0.035	0.092	0.944
SMO	3.8019%	6.2284%	0.04	0.115	0.923
FT Tree	1.4147%	6.5744%	0.35	0.138	0.921
J48/C4.5 Tree	1.1494%	6.5744%	0.05	0.103	0.92
Regression	3.4483%	9.3426%	0.05	0.195	0.966

However, the results are not quite what would be considered "production level" for malware detection. As a general rule, the goal is to keep false positives below 0.001% and false negatives below 1%. At best, among these models the false positive was 2% and the false negative was 4.6%. False positives are especially important to keep low, because customers will not use an antivirus that blocks their legitimate programs.

The models built in this project are still useful as triaging tools. In cybersecurity, incident response teams at businesses are swamped with email security alerts every day, especially with regards to suspicious email attachments. Running these models on the applicable executables can allow the incident response team to focus only on those that have a high probability of being malware. So the conclusion is that static analysis shows substantial promise, even if it is not quite refined enough to be deployed as a standalone antivirus.

VII. Future Work

The one major roadblock to this project is a lack of malware samples. The 420 samples here took weeks to obtain, whereas, by contrast, thousands of goodware samples can be gathered in a day. Expanding the sample set would probably cut down on the generalization error. One possible approach would be to catch malware using a honeypot setup, though that often comes with its own set of legal issues.

There are also more features that can be used within the Mach-O files. Adding more features could possibly increase the accuracy of the classification, but it comes at the price of a larger model.

This static analysis approach could also be used to try and classify malware into "families" (for example, all polymorphic Flashback executables), which is an open problem in the security industry.

References

[1] J. Rodriguez and L. Kuncheva, "Rotation Forest: A New Classifier Ensemble Method", *IEEE Trans. Pattern Analysis and Machine Intelligence* , vol. 28, no. 10, pp. 1619-1630, Oct. 2006.

[2] Leo Breiman, "Random Forests", *University of California*, Jan. 2001.

[3] Eibe Frank and Ian H. Witten, "Generating Accurate Rulesets Without Global Optimization", *University of Waikato*.

[4] N. Landwehr, M. Hall, and E. Frank, "Logistic Model Trees", *14th European Conference on Machine Learning*, Jun. 2004.

[5] D. Aha, D. Kibler, and M. Albert, "Instance-Based Learning Algorithms", *Machine Learning* , vol. 6, pp. 37-66, 1991.

[6] "OS X ABI Mach-O File Format Reference". Mac Developer Library. https://developer.apple.com/library/mac/documentation/DeveloperTools/Conceptual/MachORuntime/index.html.

[7] "Weka". The University of Waikato. http://www.cs.waikato.ac.nz/ml/weka/.

[8] "Contagio Malware Dump". Mila. http://contagiodump.blogspot.com/.

[9] "VirusTotal". https://www.virustotal.com/.

[10] "Mac Malware". Kaspersky Press Release 2014. https://press.kaspersky.com/files/2014/11/Mac-threats.png.