Classifying Malware into families Based on File Content

Patrick Rand, Reynier Ortiz
Florida International University
Miami, Florida, USA

Abstract

We present a mechanism to classify a malware file into one of nine families: Rammit, Gatak, Tracur, Vundo, Simda, Kelihos_ver1, Obfuscator.ACY, Lollipop and Kelihos_ver3. The system uses a large training set of disassembled malware executables, both in their binary and assembly forms. We ran two separate experimental processes: the first involved extracting n-grams from the binary files using the kfNgram tool, and the second used a shell script to parse the assembly files for method calls to external API libraries. In both cases, the attributes are merged to form a master list and later reduced by selecting the top 500 with the highest information gain (IG). *These 500 attributes are the boolean attributes to determine if they are present or not in each of the malware executables. The training set, using the selected attributes, was transformed into an .arff file required by Weka [17] to run several classifiers, i.e. nave bayes, decision trees and support vector machines (SVM). We then compared the different algorithms using some common measures: accuracy, error rate, true and false positive error rates.*

1. Introduction

The malware industry continues to evolve in complexity, causing increased personal and commercial damage. Traditional methods of malware detection usually involve matching a piece of malware to a signature that has been predefined in the system. However, malware programmers have learned to evade signature detection by polymorphically obfuscating their code. This means, that, for any given piece of malware, the program itself will be recompiled with randomized variable and string names, rendering standard signature detection near useless in identifying an instance of malware [15]. This has given rise to another goal of both research and industry: given a malware file, can we classify it into a common malware family, such that each instance of that family was morphed from the same original malware program? The ability to categorize a malicious program into a known family provides great importance because it allows for a more focused recovery and repair process. Researchers have developed methods for identifying malware by only considering static portions of the malicious code that are unaffected by polymorphism. In particular, these attributes are the method calls of the external libraries used in the code, and the byte sequence of executable. While the research behind these evaluation processes have shown to have good results, much of this research was performed on small sample size of data [12], while others have low accuracy of prediction [15]. The small data sizes is most likely attributed to the difficulty involved with building a data set of malware. First, a program will need to be trapped and identified as malware. From there, researchers can examine the code either dynamically, or statically. The former involves tracing the program's execution in secure, virtual sandbox environment, while the latter involves running the program through a disassembler, and outputting its source code. This paper and its referenced work focus on static analysis of programs, which can be generalized into a form of text/document classification, of which there is extensive research in the machine learning community. The purpose of this paper will be to survey and examine previous research in the field, and evaluate how these methods scale and perform on a very large data set. Microsoft has provided a sample data set of 10,838 disassembled malware executables, of which each program is given in both its hexadecimal byte form, and its assembly form [7]. Each of these training examples have been labeled as belonging of one of the aforementioned malware families. The results of our experiment showed similar, and at times, improved classification accuracy compared to previous research. Also, our modified system, designed to handle our 400 GB training performed well and showed its ability to scale with the large data set. Given more time, we would have liked to explore how are system handles classifying both programs that do not belong to one of the 9 origin families, as well as those that are not malware.

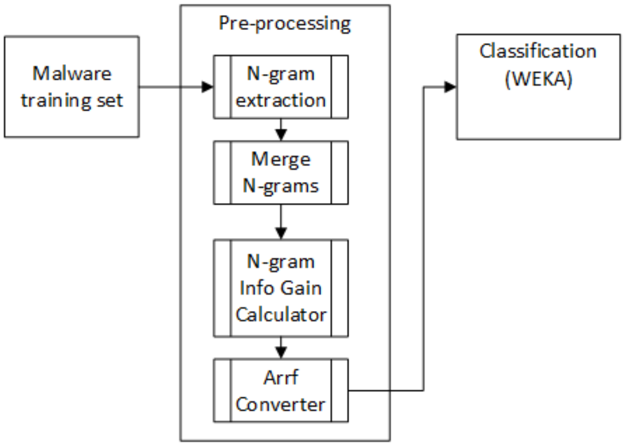


Figure 1. System architecture.

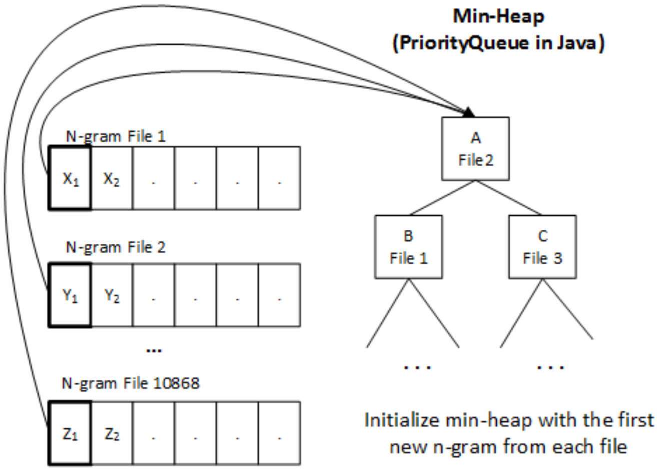


Figure 2. Initializing PriorityQueue to merge N-grams.

2. Problem Definition and Methods

2.1. Task Definition

Our approach to feature extraction and classification is based on a combination of the methods from the common text classification techniques developed in previous research [12, 15]. Where our process differed from these methods is our focus on the design of the complete system architecture, in particular, creating a system that was not only able to replicate the successful results of the referenced research, but to do so on a data set of a large magnitude of size, and by trying to classify malware into families. The size of our training set required us to design the feature extraction process in a MapReduce-styled pattern [10], but also to only consider machine learning classifiers that were able to scale with our large input. Thus, classifiers such as k-nearest-neighbors and Hidden Markov Models [6] were not considered due to their performance on large data-sets and the impracticality of scaling them in regards to our limited computing resources. For both types of feature-sets, we evaluated the classification performance of three classifiers: Naive Bayes, Decision Trees, and Support Vector Machines. Of these classifiers, Decision Trees and Support Vector Machines performed best, and from there we extended our analysis into the boosted forms of these two algorithms.

2.2. Algorithms and Methods

2.2.1 System Architecture

As shown in figure Fig 1, the architecture of the system is rather simple. The main tasks were the preprocessing of the data which involved extracting the N-grams and transform the training set into a format that Weka [17] accepts as input. And then apply several learning methods.

2.2.2 Data Preprocessing

The preprocessing part was a process that required a considerable effort because of the size of the training set, 10,868 files. The first step was to extract the hexadecimal N-grams from the binary files using the kfNgram tool [11], we used n=4.

It was then necessary to merge the resulting sorted N-gram files to effectively calculate the *information gain (IG)* of each N-gram. Because these files were extremely long, we could not use the merge functionality built in the kfN-gram tool, and we have to halt the application after running for three days. To optimize this process we merged the files maintaining a min-heap (or *PriorityQueue* in Java) in memory with the first distinct N-gram from each file, as shown in Fig 2. We were also able to keep 10,878 Java scanners in memory to speed up I/O. Then, to merging process would consist in each iteration to perform a `pop()` operation from the `PriorityQueue`, append the element to merged file, and insert the next N-gram from the same file of the element that was popped to the `PriorityQueue`. This process is represented in Fig 3.

Once merging was complete, we obtained a total of N-grams was 255,942,370, therefore we had to reduce them by calculating the *information gain (IG)* of each N-gram and selecting the top 500 (Fig. 5), the same approach used in [12]. The *average mutual information* [18] was used to select the most relevant attributes:

$$IG(j) = \sum_{v_j \in \{0,1\}} \sum_{C_i} P(v_j, C_i) \log \frac{P(v_j, C_i)}{P(v_j)P(C_i)} \qquad (1)$$

where C_i is the ith class,v_j is the value of the jth attribute, $P(v_j, C_i)$ is the proportion that the jth attribute has the value v_j in the class C_i, $P(v_j)$ is the proportion that the

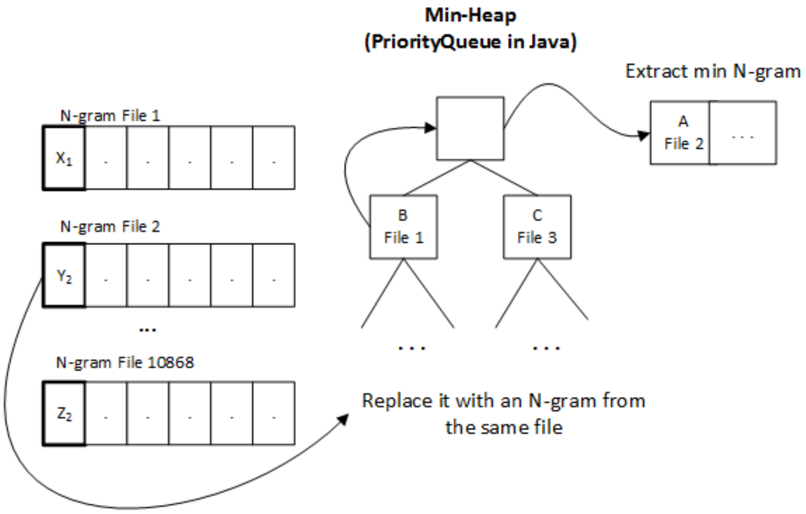


Figure 3. Process of merging N-grams. The minimum element is extracted and replaced with another N-gram from the same file

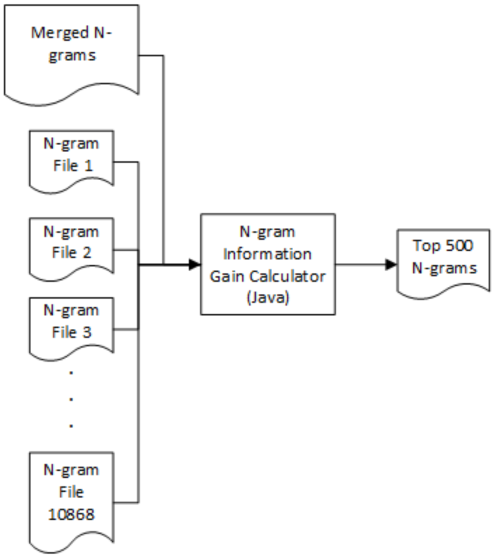


Figure 4. Process of calculating the information gain of each N-gram.

jth n-gram takes the value v_j in the training data, and $P(C_i)$ is the proportion of the training data belonging to the class C_i [12].

The final step, as shown in Fig **??**, in the data preprocessing was to transform the training set in .arff format where each example was represented by the presence (or absence) of the top 500 N-grams. This .arff file is one of the formats supported by Weka [17], which facilitated to experiment with different machine learning algorithms.

Our system design for extracting the external API strings followed a near identical design, with the only difference being in the method use to parse the files (.asm) for the method calls. We used a simple shell-script that was designed to parse the files for any API methods using regular expressions. From there the we followed the same feature-set building pattern as for the n-grams.

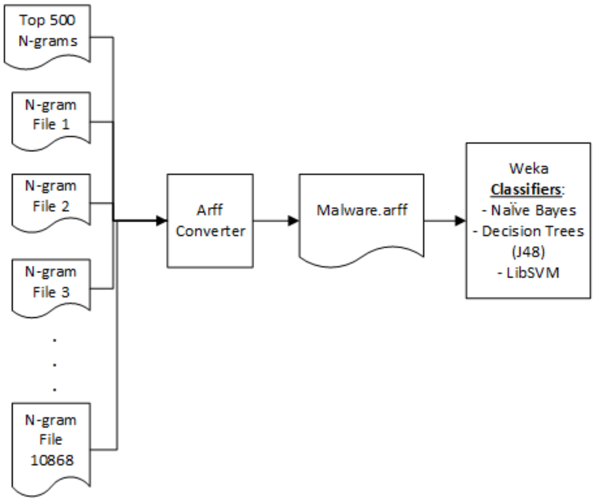


Figure 5. Transforming training set into .arff format.

2.2.3 Classification algorithms

The Nave Bayes classifier uses Bayes' Theorem to label examples based on the assumption of conditional independence between features. Given a data set consisting of word counts, the training of this algorithm is very simple, and only requires the computation of the prior probability of each class, $P(C_i)$, and the conditional probability of each word given a class, $P(v_j|C_i)$. For a given instance $\mathbf{v}$, we aim to classify it into some class, C_i, thus computing

$P(C_i|v)$. Using Bayes' Theorem, this can be decomposed into:

$$P(C_i|\mathbf{v}) = \frac{P(C_i)\, P(\mathbf{v}|C_i)}{P(\mathbf{v})} \tag{2}$$

We can ignore the $P(\mathbf{v})$ term, and estimate the remaining feature probabilities as:

$$P(C_i) = \frac{\text{\# of occurences of class } i}{\text{\# of total attributes}} \tag{3}$$

$$P(v_j|C_i) = \frac{\text{\# of occurences of attribute } v_j \text{ with class } C_i}{\text{\# of occurences of class } i} \tag{4}$$

From here, we can define the classification of training example **v** as:

$$\hat{y} = argmax\ P(C_k) \prod_{i=1}^{n} p(v_i|C_k) \tag{5}$$

Decision trees are a method of classification that use tree-like graphs to model the classification of an example as a path of probabilities from the root node attribute, to one of the leaf (class) attributes. The beauty of decision trees lies in their simplicity and easy interpretation. Likewise, decision trees can be thought of as a ”white-box” classification model, because once the tree has been built, the researchers can visualize and analyze each sub-path of an example's classification path.The basic implementation of a decision tree works as follows: for each node of tree, we select the attribute that best splits the our training examples into their respective classes using the entropy gain for choosing that attribute. For our system, we used the J48 decision tree algorithm in Weka, which is an implementation of the ID3 decision tree algorithm [14]. We define the entropy for a data set S, and its set of attributes V, as:

$$H(S) = -\sum_{v \in V} P(v) \log_2 P(x) \tag{6}$$

Once we obtain the corresponding decision tree, we prune the tree to help minimize over-fitting that may have occurred as a result of tree's greedy construction.

Support Vector Machines (SVMs) produced the best classification results of our tested algorithms. The goal of an SVM is to find an optimal hyperplane that separates the training examples by their respective classes, while maximizing the margin between theses classes. The hyperplane is represented by a vector w and a scalar value b. Thus, the solution of the SVM can be represented by the minimzation of:

$$P(\mathbf{w}, b, \boldsymbol{\xi}) = \frac{1}{2}\, \mathbf{w} \cdot \mathbf{w} + C \sum_{i=1}^{n} \xi_i \tag{7}$$

such that:

$$\forall_{i=1}^{n}\ ,\ \xi_i \geq 0\ :\ y_i(\mathbf{w} \cdot \mathbf{x}_i + b) \geq 1 - \xi_i \tag{8}$$

where ξ_i is the upper bound on the number of training errors and C is the parameter that controls the trade-off between the margin of the hyperplane and the training error. We use the LibSVM [8] implementation of this algorithm.

We used pieces from two papers to design our system and framework. We expand the method of determining whether a given file is benign or malignant, i.e. whether or not it can be classified as malware [12]. We also draw our n-gram extraction process directly from their paper. We used this feature extraction process to build our feature set which we then pushed through our algorithms. From ¡i¿Learning and classification of malware behavior¡/i¿, by K. Rieck et al, we derive our method of classifying malware into one of nine malware families.

3. Experimental Evaluation

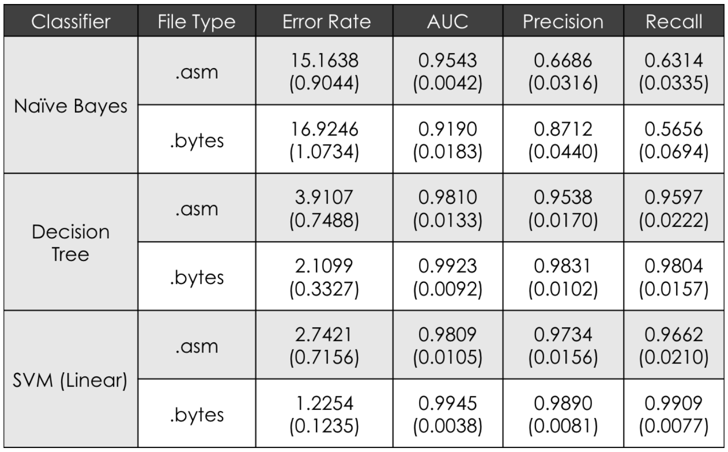
3.1. Methodology

Our experimental method blends the approaches of experiments in existing literature [12, 15].We analyzed the system by running three machine learning classifiers on both of the feature sets we extracted- assembly and binary. The classification algorithms that we ran were the Naive Base Classifier, the Decision Tree ID3 algorithm, and the linear Support Vector Machine Classifier. The data we will present is our test error rate, Area Under the Curve (AUC), the precision and recall rates. All of these metrics were derived using 10-Fold cross-validation. We will delineate in the next section how these measurements relate to our ability to classify a malware file into one of the nine origin families. It is useful to determine the origin family of malicious file because each malware family describes their behavior and the damage they cause. This allows one to begin to diagnose and repair a machine. This information can also help one to prevent malware from executing undetected. It also provides a road map for locating corrupted files and other deleterious deletions or insertions.

We tested our method using 10,838 disassembled malware executables in both their binary and assembly formats, which we acquired from the Microsoft malware data set [7]. The information in this data set is compelling because it contains real-world viruses acquired and dissassembled by Microsoft. In addition, the data set is large enough to account for a wide variety of polymorphic changes to any of given malware family.

3.2. Results

The results of our evaluation will show how the feature extraction methods outlined in the Kolter paper and the family classification methods in the Rieck paper are scaled, and how we replicated and improved upon their results. We had the most success with the accuracy of the SVMs for both the



Classifier	File Type	Error Rate	AUC	Precision	Recall
Naïve Bayes	.asm	15.1638 (0.9044)	0.9543 (0.0042)	0.6686 (0.0316)	0.6314 (0.0335)
	.bytes	16.9246 (1.0734)	0.9190 (0.0183)	0.8712 (0.0440)	0.5656 (0.0694)
Decision Tree	.asm	3.9107 (0.7488)	0.9810 (0.0133)	0.9538 (0.0170)	0.9597 (0.0222)
	.bytes	2.1099 (0.3327)	0.9923 (0.0092)	0.9831 (0.0102)	0.9804 (0.0157)
SVM (Linear)	.asm	2.7421 (0.7156)	0.9809 (0.0105)	0.9734 (0.0156)	0.9662 (0.0210)
	.bytes	1.2254 (0.1235)	0.9945 (0.0038)	0.9890 (0.0081)	0.9909 (0.0077)

Figure 6. Experiment results (mean and standard deviation)

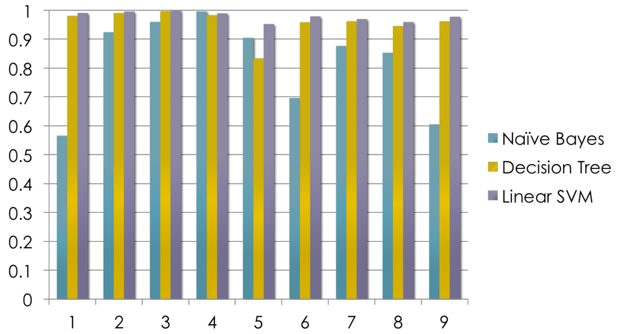


Figure 7. Family Accuracy using n-gram feature set

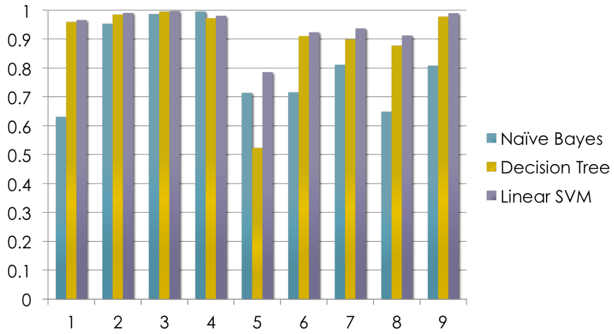


Figure 8. Family Accuracy using API strings feature set

API strings and the n-gram feature sets as seen in Fig 6. The error-rate analysis, AUC, precision and recall had the best results for both of these feature sets. These findings were in line with the cited literature and even showed improved results in regards to the Rieck paper.

Two of the stronger classifiers, SVM and Decision Trees, not only had improved accuracy for the n-gram approach but also a much smaller standard deviation which shows there is less variance in those results as compared to the API strings.

In Fig 7 and Fig 8, the per-class accuracy of both methods was strongest for the linear SVM classifiers as well with the n-gram approach having all around better class accuracy results in general compared to the API strings method. We can attribute these findings to the fact that the API strings are not as unique a feature identifier as the n-grams.

While the Rieck data set is comparable in size to the Microsoft data set, we achieved better accuracy and a better AUC measure. On average Rieck et al were only able to accurately classify 88% of their sample file [15], where we were able to correctly assign 98% to the proper malware family. Furthermore, where Rieck et al focus their work on a dynamic behavioral analysis of the API class, we focused our method on a more static paradigm, looking solely at the code contained within each malignant file. This dynamic behavioral modeling approach is not applicable to the average consumer and is therefore not as marketable. Whereas our detection system is transferable in the sense that it has real world applications. One only needs to run the file through a disassembler,such as IDA [?], then parse the output for the n-grams or the API streams. This can then be run through the SVM classifier, all of which can be done in real time and does not require the virtual sandbox environment that is required for the Rieck dynamic-behavior experiment.

3.3. Discussion

Our results showed the importance of selecting the N-grams that would provide more information about the underlying family by calculating the IG, instead of choosing the most frequent N-grams. With this method we intuitively consider that irrelevant N-grams are avoided, which are those that are product of mutation engines and other techniques used by hackers, and also we assume that by calculating the information gain we somewhat extracted the most common instructions per malware family. This, we think, is the reason why we obtained a high accuracy with Decision Trees and Support Vector Machines, about 99% in both. The results are also inspiring in the sense that, at difference with other papers, our data set is considerable large, and using several evaluation methods as cross-fold validation we presume we are not incurring in overfitting.

This method is far from perfection, as there is still a considerable change of selecting irrelevant N-grams, which could have been mitigated by using sequential pattern extraction or Markov Models. The reasoning is that the mutation engines, we believe, would not be capable of generating sequences of instructions that would mislead the classification if we apply Markov Models or sequence pattern extraction correctly.

4. Related Work

Malware detection and classification is a problem being addressed through several angles. In [9], the goal is to detect if a program exhibits a specified malicious behavior by determining if a set of templates of sequence of instructions are present in the executable files. This approach requires to have knowledge on semantics of each of the malware families. Although the results are promising, there are several reasons we could not follow this approach, for example, as inferred by the title, it would be necessary to create a set of sequence of instructions for each of the nine families we need to classify the malwares into. This is infeasible due to the length of the course, but also it is outside of the scope of the course. As exhibited in the results, this strategy is resilient to obfuscation and showed improvements when compared to McAfee VirusScan.

Several malware classification algorithms are based on n-grams extracted from the executable files. In [13], instead of byte sequences, the n-grams extracted are formed by machine codes. After obtaining the n-grams from the malwares, a centroid for each family is created by selecting the most frequent n-grams. Then, the strategy to classify a malware into one of the families, is to determine the centroid which the malware is more similar to by counting the number of matching n-grams. When considering this approach, one of the possible limitations analyzed was that selecting the most frequent n-gram could implicate choosing an n-gram irrelevant to the malware family.

We also explored sequential pattern extraction of n-grams. The proposed methodology in [16] outlines a procedure to use the n-grams patterns to classify the malware by family. The kfNgram tool [11] was used to extract the n-grams from the disassembled files with n=1, n=2, n=3 and n=4, obtaining the best results with n=4. In [12] the accuracy achieved was higher with n=4, therefore we skipped this evaluation and use only n-grams with n=4. The sequential pattern extraction technique in [19] was used to generate frequently occurred sequences of n-grams to represent the data [16]. Then the patterns significance was calculated using the term frequency-inverse document frequency (TF-IDF) where the term refers to the n-gram pattern and a document to the malware file. Since the number of patterns was too large, the sequential floating forward selection (SFFS) procedure was applied to reduce the number of features, in this case, n-gram sequential patterns. With all the features extracted, three classification algorithms were used, C4.5, multilayer perceptron and support vector machine. The training set was randomly split into two partitions using 80% for training and 20% for testing achieving a 96.64% of accuracy [16]. Because of the duration of the course and the complexity of sequential pattern extraction, we were not able to experiment with this approach.

Following Occam's razor, suggesting that the simplest hypothesis is the best, we applied an approach similar to the one described in [12]. The n-grams extracted from the executable files represented boolean features, present (i.e., 1) or absent (i.e., 0). Since the n-grams list was too large, it was necessary to select the most relevant attributes (i.e., n-grams) by computing the *information gain (IG)* described in [18] for each, also called *average mutual information*. Through pilot studies, it was determined to use the top 500 n-grams, and then applied classifiers implemented in the Wakaito Environment for Knowledge Acquisition (WEKA) [17]: IBk, Nave Bayes, SVM, and J48 (decision tree), and also *boosted* the last three of these learners [12]. The results indicated 98% the highest accuracy using boosted decision trees.

5. Future Work

Malware classification is a topic that is always evolving because viruses' authors are constantly developing techniques to avoid been detected, for example by obfuscating the code or by designing polymorphic behaviours. The proposed approach is agnostic of these techniques, therefore, we assume intuitively that a set of the extracted N-grams are product of this ingenious methods. Mutation engines are capable of generating millions of variations of the same virus, therefore, to overcome this problem, it would be necessary to have knowledge of the malware behaviors to develop heuristics or detect specific sequence of instructions to find in the malicious files.

For time reasons, we could not research on the possibilities of removing the irrelevant computations in the malware files created to evade anti-viruses. This would help, if using sequential N-gram patterns extraction, to detect patterns that are truly correlated to the malware family. These patterns could, somewhat, describe the common semantics present in each class of malwares.

6. Conclusion

We presented a methodology for classifying malware binary files into nine families. The initial approach for the project was to extract the API libraries used in each file and make them the features in the classification models. But another approach was taken, using N-grams from the binary files. We described process of extracting N-grams, with *n=4*, from the executable files, then calculate the *information gain (IG)* of each of them in order to select the top 500 N-grams with the highest *IG*. We described how to preprocess the large data set in *reasonable* time, and how to transform it into .arff format to use it as input in Weka [17]. We used several classifiers (Decision Trees, SVM and Naive Bayes) and compared the results to the ones obtained using API libraries. Our classification algorithms in Weka reached about 98% of accuracy, better than the API libraries. But we proposed in a future work, to use sequential pattern extraction or Markov Model which intuitively should lead to better results.

References

[1] A. Alpher. Frobnication. *Journal of Foo*, 12(1):234–778, 2002. 3

[2] A. Alpher and J. P. N. Fotheringham-Smythe. Frobnication revisited. *Journal of Foo*, 13(1):234–778, 2003. 3

[3] A. Alpher, J. P. N. Fotheringham-Smythe, and G. Gamow. Can a machine frobnicate? *Journal of Foo*, 14(1):234–778, 2004. 3

[4] Authors. The frobnicatable foo filter, 2014. Face and Gesture submission ID 324. Supplied as additional material `fg324.pdf`. 2, 3

[5] Authors. Frobnication tutorial, 2014. Supplied as additional material `tr.pdf`. 2

[6] L. E. Baum and T. Petrie. Statistical inference for probabilistic functions of finite state markov chains. *Ann. Math. Statist.*, 37(6):1554–1563, 12 1966. 3

[7] M. M. P. Center. Microsoft malware classification challenge (big 2015), 2015. 1, 6

[8] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011. 5

[9] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant. Semantics-aware malware detection. pages 32–46, 2005. 6

[10] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008. 3

[11] W. H. Fletcher. Data mining: Practical machine learning tools and techniques, 2012. 3, 7

[12] J. Z. Kolter and M. A. Maloof. Learning to stop to detect and classify malicious executables in the wild. 7(Article 19):2721–2744, 2006. 1, 3, 4, 5, 7

[13] A. Pekta, M. Eri, and T. Acarman. Proposal of n-gram based algorithm for malware classification. 2011. 7

[14] J. R. Quinlan. Induction of decision trees. *Mach. Learn.*, 1(1):81–106, Mar. 1986. 5

[15] K. Rieck, T. Holz, C. Willems, P. Dussel, and P. Laskov. Learning and classification of malware behavior. pages 108–125, 2008. 1, 3, 5, 6

[16] O. Sornil and C. Liangboonprakong. Malware classification using n-grams sequential pattern features. 2013. 7

[17] I. H. Witten and E. Frank. Data mining: Practical machine learning tools and techniques, 2005. 1, 3, 4, 7, 8

[18] Y. Yang and J. O. Pederson. A comparative study on feature selection in text categorization. pages 412–420, 1997. 4, 7

[19] N. Zhong, Y. Li, and S. T. Wu. Effective pattern discovery for text mining. 24(Issue 1):30–44, 2012. 7