Using String Information for Malware Family Identification

Prasha Shrestha[1], Suraj Maharjan[1], Gabriela Ramírez de la Rosa[2], Alan Sprague[1], Thamar Solorio[1], and Gary Warner[1]

[1] University of Alabama at Birmingham
Birmingham, Alabama
{prasha, suraj, gabyrr, sprague, solorio, gar}@cis.uab.edu
[2] Universidad Autónoma Metropolitana,
Unidad Cuajimalpa
gramirez@correo.cua.uam.mx

Abstract. Classifying malware into correct families is an important task for anti-virus vendors. Currently, only some of them will recognize a particular malware. Even when they do, they either classify them into different families or use a generic family name, which does not provide much information. Our method for malware family identification is based on the observation that closely related malware have heavy overlap of strings. We first created two kinds of prototypes from printable strings in the malware: one using term frequency–inverse document frequency (tf-idf) and the other using the prominent strings extracted from the vocabulary. We then used these prototypes for classification. We achieved an accuracy of 91.02% by considering the entire vocabulary and an accuracy of 80.52% by considering 20 prominent strings for each malware family. Our accuracy is high enough for our system to be used to classify even those malware that can confuse the anti-virus vendors.

Key words: malware, prototype based classification, prominent strings, tf-idf, cosine similarity

1 Introduction

Malware is defined as malicious or malevolent software that threatens computers and computer systems and often damages or disables them. Malware can also gain access to private and sensitive information like social security number, bank and credit card numbers. Malwares created by a hacker group and then modified and improved successively by the same group or other different groups fall under a malware family. Malware belonging to the same family exhibits similar behavior and performs similar system calls. For instance, Zeus family of malwares can steal a victim's bank credentials and other valuable information like Social Security Number. Users need to periodically scan and update their system by using anti-virus software to protect it from the hazardous attacks of malware. But these scans only make your machine as safe as the malware detecting and correct family labeling capacity of the anti-virus vendors that you are using.

Knowledge about malware has great importance for anti-virus vendors and there is a lot of research dedicated to this task. Being able to classify a malware into the correct family is crucial as one can predict the characteristics of the malware based upon its family. By using these characteristics, we can design better solutions to control malware. Manually classifying them is tedious, time consuming and does not scale well with their ever-growing quantity. Hence, automatic classification systems are needed to address this problem and ease the study of malware behaviours. Automatic classification systems have many applications, such as prediction of malware behaviour and of damage it may cause to the system, potential solutions to disable it, and even deeper analysis of its behaviour. Today, there is a lot of discrepancy between anti-virus vendors in assigning family labels to malware. Furthermore, sometimes these vendors are unable to recognize a particular malware.

Most of the earlier research on malware family identification is based on dynamic analysis of the malware by running it on a virtual sandbox environment. Park et al. (2010) have created system call dependency graph of the malware by running it on a virtual domain. They then measured edit distance based upon the maximal common subgraph between two dependency graphs and used a predefined threshold to classify them as being similar or different in behavior [1]. Bailey et al. (2007) used behaviour characteristics of malware rather than just sequences and patterns of system calls to classify malware [2]. They gathered the behavioural data from system logs after running malware in a virtual environment.

Some of the researches also focus on static analysis on disassembled malware. Recently, Tian et al. (2009) explored the classification of unpacked malware using features such as string frequency with AdaBoost and Random Forest classification algorithms [3]. In addition, Shabtai et al. concluded that a framework could be designed that could detect new malicious code with great accuracy [4]. They suggested using features like OpCode and byte n-grams with different classification algorithms and then ensemble results based on weights.

Our method is also a static method. A clear advantage that static methods have over dynamic ones is that dynamic analysis is harder than static since it requires running the malware on a virtual sandbox. Our method focuses on the information contained in the printable strings of an unpacked malware file. We assign a weight per family to each of these printable strings as a relative indication of its association with that family. By using these strings and their weights we construct the prototypes that will represent a family. All of the strings in the vocabulary of the training set along with their corresponding weights for a given family form a prototype for that family. The other prototype we use is based upon prominent strings and is a more concise representation of that family. The strings with the highest weights for a family are considered as the prominent strings for that family of malware. These prominent strings and their corresponding weights form the prototype for that family. In order to do the classification, we compute the cosine similarity between strings in the prototype and strings in the test malware file. The file gets classified into the

family that results in the highest similarity. Furthermore, by using our prominent strings prototype, we experimented with soft-string matching using Levenshtein distance and Jaccard coefficient. Additionally, we hypothesized that the absence of prominent strings in a given malware test sample will give us some clue about incorrect labeling of the malware family by the anti-virus vendors. We leveraged this idea to detect the wrongly labeled malware samples. In short, we used the information encoded in the printable strings to classify malware into their respective families.

2 Prototypes

The core of our method is prototypes. In this section we describe the creation of our prototypes and also the prototype based classification method that we have used.

2.1 Prototype Based Classification

To classify an unseen malware file into a set of known families we use a prototype-based classification approach. The prototype based approach is one of the traditional methods for supervised text classification. There are two phases in this approach: training and testing. The training phase involves the construction of one single representative instance, called prototype, for each class or family. Then, in test phase, each given unlabeled file is compared against all prototypes and is assigned the family having the greatest similarity score. There are several ways to build a prototype in the training phase [5]. The assignation of a family or category to the vector representation of a given file f is based on the following criterion:

$$\text{family}(f) = \underset{i}{\text{argmax}}(\text{sim}(f, P_i)) \tag{1}$$

where, P_i is the prototype vector for family i and

$$\text{sim}(f, P_i) = \frac{f \cdot P_i}{||f||\,||P_i||} \tag{2}$$

2.2 Weighting Scheme

Our weighing scheme should assign weights to strings according to their relevance to a malware family such that the string could be useful in discriminating that family from other families. In natural language processing, three different metrics are commonly used to measure the importance of each term according to three observations: a) terms that appear many times in a single file should be more important; b) terms that appear in many files should be less important; and c) a file that contains many terms should be less important.

$$\text{tf}(t, d) = \frac{\text{freq}(t, d)}{\max\{\text{freq}(s, d) : s \in d\}} \tag{3}$$

$$\mathrm{idf}(t, D) = \log \frac{|D|}{|\{d \in D : t \in d\}|} \tag{4}$$

$$\mathrm{tf} \times \mathrm{idf}(t, d, D) = \mathrm{tf}(t, d) \times \mathrm{idf}(t, D) \tag{5}$$

The tf factor as shown in Equation 3 takes into consideration the first observation, the idf from Equation 4 accounts for the second one and the third is considered in the normalization during the computation of tf. The tf-idf function as shown in Equation 5 (as [6] well noted) combines the tf and idf factors and thus incorporates all three criteria.

2.3 Prominent Strings Set (PSS)

We define a prominent string (PS) as a string that appears very frequently in a given malware family but appears rarely in any other family. The set of strings with top k weights for a family is defined as its prominent strings set. We hypothesized that a set of prominent strings PSS for a given family can distinguish one family from another such that its PSS can be seen as the signature of that family. In other words, a PSS should be enough to identify the family of a given unseen malware file.

2.4 Building Prototypes

In order to build prototypes, we first merged all the files labeled as belonging to that family into a single file. We call this new file, $TrainingFile_i$. Then, we computed the tf-idf factor for each unique string in $TrainingFile_i$. We repeated this process for each of the F families in the training dataset. We then used the strings and their tf-idf values to build two different types of prototypes.

To build the first type of prototype, we started out by creating a global vocabulary consisting of all the unique strings in the training dataset, i.e., all the unique strings of each $TrainingFile_i$. Then for each family, we formed a single vector consisting of the corresponding tf-idf values of all the strings in the global vocabulary. This vector is representative of a family and thus is our first kind of prototype.

In the second method, instead of considering the entire vocabulary, we reduced the number of strings that represents a family by only considering the prominent strings. For each $TrainingFile_i$, we ranked the strings in the file according to tf-idf values. Finally, we took the top k strings with highest tf-idf values as the set of prominent strings for that family i. We repeated this process for all F families. After this, we had F different sets of prominent strings along with their tf-idf values, one for each family. This forms our second kind of prototype.

Our prototype is a vector space model representation of a family, where each string in the prototype is a dimension of the vector $\{s_1, s_2, ..., s_d\}$ and the weight of each s_i is given by the tf-idf values computed in the previous phase. For our

first prototype, the dimension d of this vector will be equal to the length of global vocabulary while for the second prototype it will be equal to k.

In the test phase, we will represent an unseen malware file as a vector as well, of equal dimension as the prototype vector. We will then use the Equations 1 and 2 to determine the family for each malware file in the test set. This process is described in the section below.

3 Malware Family Identification

In order to assign a family label to a test malware file, we used prototype based classification as described in Section 2.1. For each test malware file, we created a test vector consisting of the strings in the prototype. This is described in Section 3.1. For both kinds of prototypes, we performed malware family identification for each malware file by using this method. For the prototype based upon prominent strings, we further explored its use in two different ways, which we describe in Sections 3.2 and 3.3.

3.1 Exact Match

In this method, in order to classify a malware file, we first calculated a list of tf-idf values for the unique strings in that file. We then tested this file against the prototype of each family. For every string in the prototype, we took the tf-idf value from the above list in order to construct a test vector of that file. Those strings that are in the prototype but not in the file will have a value of 0. In the case where we use the whole vocabulary as a prototype, this vector will mostly be a sparse vector. For the prominent strings prototype, the vector will usually be sparse when we test against the prototype of a family that the file does not belong to. We then calculated the similarity between this vector and the vector of tf-idf values for the prototype by using cosine similarity between these two vectors (Equation 2). This process corresponds to the general schema of prototype-based approach.

3.2 Nearest Match

Nearest Matches captures not only exact same strings but also strings that are similar to the prominent strings. Two strings are considered similar if their Jaccard Coefficient is higher than certain threshold. Wei et al. (2009) generalised the definition of Jaccard Coefficient to strings [7]. Their definition of Jaccard Coefficient is as follows:

$$\text{Jaccard}(s,t) = \frac{\text{ILD}(s,t)}{|s| + |t| - \text{ILD}(s,t)} \tag{6}$$

where ILD is the Levenshtein distance that computes the minimum number of edits (insertions, deletions and substitutions) required to convert the string s to the string t. It is worth mentioning that ILD is taking into account the order of

the letters in the strings, thus strings like 'silent' and 'listen' are not the exact same strings according to the distance measure.

The prominent strings set with their normalized term frequency values were used as prototype for classification. Each of the prominent strings was compared with strings in the test file in order to find similar strings. The sum of tf values of all these similar strings is taken as the value representing that prominent string in the test query vector. We weighted the strings by only using the tf and not the tf-idf because strings found to be similar to the prominent strings might not appear in the training dataset at all, which means that we do not have their idf.

3.3 Absence of PS

Since prominent strings are representatives of a malware family, lack of these strings in any sample labeled as a certain family raises dubiousness about its labeling. We used this idea to check the labeling of files done by anti-virus vendors. If the intersection between prominent strings sets and the set of all unique words in the test file is low, then the label is said to be incorrect. This is two class classification since we can only classify a file as either being correctly classified or misclassified. Although this method cannot predict the correct class of a given file, it is beneficial in terms of speed because it will need very low number of computations after we have built the prototype.

4 Dataset

We used a dataset consisting of 1504 malware files from our university's malware database. For each malware file, family labels were obtained by using Virustotal [3], which takes the MD5, SHA1 or SHA256 of a malware file and provides the family labels for a sample from different anti-virus vendors. When the data was collected, Virustotal provided results from 47 anti-virus products. For each malware file, n-way vendor agreement was found out, where n denotes the highest number of vendors that agreed upon the same family for the file. For example, given a malware file f, if out of a set of family labels $\{l_1, l_2, ..., l_m\}$, the label l_i was assigned by n malware vendors to f while all the other labels were assigned by less than n malware vendors, then f is said to have n-way vendor agreement. Our dataset contains only those files with at least *5-way vendor agreement*, i.e. at least five vendors assigned that same family label to the malware file. The Table 1 shows the distribution of files with n-way vendor agreement.

Our dataset has malware files belonging to 10 families. The distribution of files across these families is not uniform. The distribution of the files by family is given in Table 2. The data files we used were unpacked by our university's Malware and Forensic Research Lab and contain only printable strings. We extracted the strings from the file by splitting at null character and also performed a preprocessing step to remove all those strings whose length is less than five

[3] https://www.virustotal.com/

Table 1. File Distribution with n-way Vendor Agreement

Vendors (n)	Number of Files
5 - 10	424
11 - 15	235
16 - 20	269
21 - 25	314
26 - 30	206
31 - 35	56

characters. The rationale behind removing small strings is that these are repeated across families, making them less likely to be prominent. They only increase the noise in the data, as well as the processing time.

Table 2. Number of files per family

Family	Files in the family
Bifros	333
Buzus	166
Gamevance	286
Kazy	99
Kbot	107
Medfos	100
Ramnit	115
Sality	92
Virut	93
Zeus	113
Total	**1504**

5 Experiments and Results

We conducted four types of experiments to classify malware files: 1) using the global vocabulary as a prototype to test using exact matches, which we referred to as *Exact Matches: Global vocabulary*; 2) using a prominent strings set as prototype for each family to test using exact matches, which we referred to as *Exact Matches: PS*; 3) relaxation of experiment 2 to allow the consideration of strings that are not a perfect match but are similar, which we referred to as *Nearest Matches*; and 4) using absence of exact matches of prominent strings to find files that are wrongly labeled, which we referred to as *Absence of PS*. There are several parameters being used in our experiments, such as number of prominent

strings (N), absence threshold and string similarity threshold. We performed tuning of these parameters by running experiments on a smaller dataset containing 20 files per family. In addition, we used 10 fold cross validation to test the consistency of our classification model.

Table 3. Prominent Strings (PS) per family

Family	Prominent Strings
Bifros	!This program cannot be run in DOS mode.\r\r\n, GetProcessHeap, HeapAlloc, kernel32.dll
Buzus	Boolean, TObject, 333333333333333333, comctl32.dll
Gamevance	Are you sure you want to cancel XOBNI Outlook Plugin Installation?, NativeQuad, Rd long, userenv.dll
Kazy	__vbaCyMulI2, MyComputer, __vbaUI1I2, ThreadSafeObjectProvider‘1
Kbot	MSVCRT.dll, .rdata, MSVCRT.dll, uEKKxGLup
Medfos	CoGetCallContext, DeleteTimerQueueTimer, GetProcessPriorityBoost, StgCreatePropStg
Ramnit	.data, @h4a@, ;;;;;;;;;, strrchr
Sality	SAF.ocx, CustomizationManager.SAFDesigner, SAF.SAFFloat, SAF.SAFMaskedText
Virut	__p__commode, _exit, wwwwwwwwwww, _controlfp
Zeus	abcdefghijklmnopqrstuvwxyz, ˆ+*;W, GetLastInputInfo, /2ABj37

Figure 1(a) shows the results for our first experiment *Exact Match: Global Vocabulary*. This method uses prototypes based on global vocabulary and has been described in Section 3.1. As the graphs show, the global accuracy is 91.02% and the accuracy values are almost above 80% for every family except for *Kbot*.

In Table 3 we have listed some of the prominent strings (PS) captured by our model for each family. As the table suggests, the prominent strings are highly disjoint across the families. We created a prototype based upon prominent strings and used it to perform our other experiment *Exact Match: PSS* described in Section 3.1. Each of the prototypes consists of 20 such prominent strings per family. The goal of this experiment was to determine the performance of applying prominent strings in the identification of malware files. Figure 1(b) shows the accuracy obtained for each family as well as the accuracy for the entire test set by using this method. The global accuracy is 80.52%. The families with less accuracy and thus more errors are *Kbot*, *Kazy* and *Sality*. When we calculated the average number of prominent strings per malware sample, *Kbot* had the lowest value of 1.31. This might be the reason that the accuracy for *Kbot* is so low. Similarly, *Kazy* also has the second lowest value of 4.62. The *Exact Match: PSS* experiment was completed within 23 minutes while the *Exact Match: Global Vocabulary* took around 44 minutes to complete. The *Exact Match: PSS* experiment takes only half of the time than the *Exact Match: Global Vocabulary* experiment while still giving reasonable accuracy.

Our *Nearest Match* experiment was designed to analyze change in performance when a more relaxed comparison than exact matches is performed, as we explained in Section 3.2. We used a fixed number (0.8) as the similarity threshold between two strings. Figure 1(c) shows the accuracy obtained for this experiment. The graphs show a decrease in the global accuracy to 59.57%. *Ramnit* and *Kbot* families show an error of nearly 100%, which indicates that this method is not good enough to distinguish among families. Since the worst results were obtained by using *Nearest match* we also tried using just intersections instead of a prototype based approach on *Nearest Match* to check if we can get better results this way. To test a file against a PSS, we took an intersection between the set of all strings in the test file and the PSS. Since we are using the *Nearest Match* method, we also consider similar strings as belonging to the intersection (string similarity is the same as before). The results of this experiment are presented in Figure 2. As we can see the performance was increased to 69.21%, but we still have families with less that 50% accuracy.

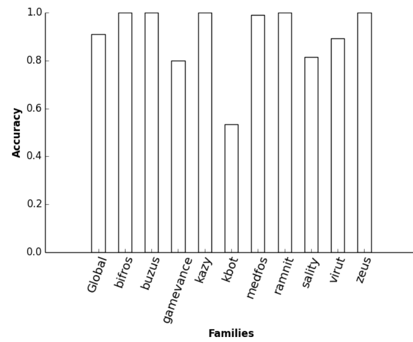
5.1 Absence of PS

We designed this experiment to investigate if the absence of PS in a given file indicates that a wrong family label was given to that particular file, as we described in Section 3.3. But since we do not have the ground truth to compare against and we only have the vendor agreement data, we cannot really report the accuracy for this case because any instance found as mislabeled will be counted as an error. We can only compare with the vendor agreement data. Figure 1(d) shows a graph of the percentage of files that were found by this method to be correctly labeled by the vendors across different malware families. The total agreement is around 74.1%. For families like *Bifros*, *Buzus* and *Ramnit*, the label given by vendors seems to be correct as this method also agrees with the vendors 100% for these families. However, for certain families like *kbot* and *kazy*, this method does not really agree with the vendor labels. These are the same families for which other methods too have lower accuracy. As shown by this method, the lower accuracy may have been the result of the files actually being mislabeled by the vendors.

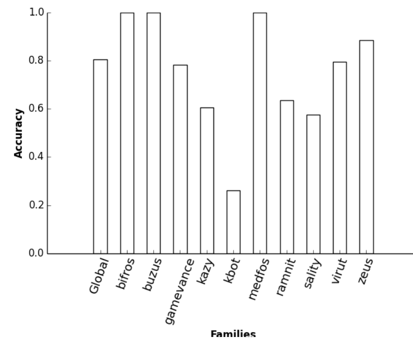
5.2 Correlation with Vendor Agreement

We tried to find the correlation between n-way vendor agreement and the accuracy that our method achieves. Since we use vendor agreement as the gold standard to find the accuracy, this relation is important. We observed that for higher vendor agreement, our method also has higher accuracy.

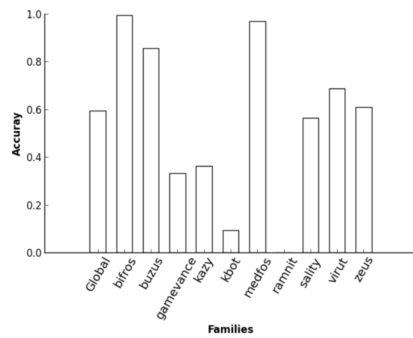
Figure 3 shows the accuracy of our system per n-way vendor agreement for Experiments *Exact Match: Global Vocabulary* and *Exact Match: PSS*. Among the 47 vendors, if 30-35 of them agree upon the family of a malware file, the average accuracy for such a file is mostly between 80% to 100% in both experiments. On the other hand, if only 5-10 vendors agree upon the family, the average accuracy drops to 30% for Experiment *Exact Match: PSS* and to 56% for Experiment



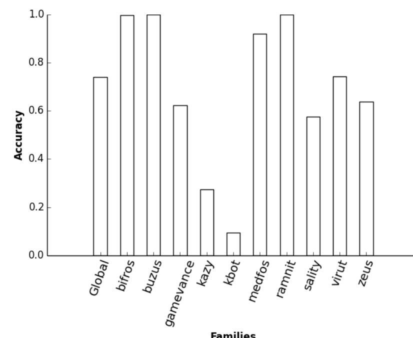
(a) Exact Match: Global Vocabulary



(b) Exact Match: PSS



(c) Nearest Match



(d) Absence of PS

Fig. 1. Accuracy for four different methods: Exact Match: PSS, Nearest Match, Absence of PS and Exact Match: Global Vocabulary. (PS = Prominent Strings, PSS = Prominent Strings Sets)

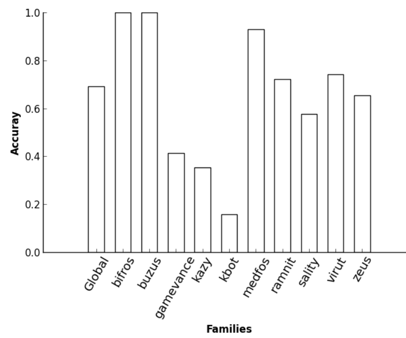
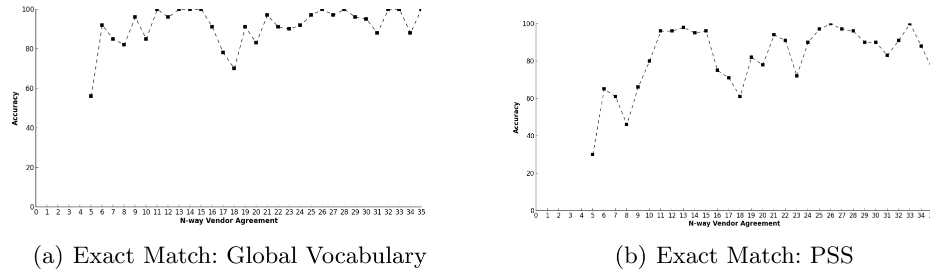


Fig. 2. Accuracy for Nearest Matches using intersections

Exact Match: Global Vocabulary. In this case, since the vendor agreement is so low, the family label assigned by these vendors might not even be correct for that malware file. So, even when our results do not agree with the vendor label, our method could be correctly classifying these malware files. But we have no way of knowing this right now. In Figure 3(a), the relative change in accuracy is not that pronounced across vendor agreement since the accuracy of this method is very high in itself and thus, most of the accuracy values are in a small range.



(a) Exact Match: Global Vocabulary

(b) Exact Match: PSS

Fig. 3. Accuracy per n-way vendor agreement

6 Conclusion and Future Work

The accuracy for all our methods is above 80% except for the nearest match method. Our *Exact Match: Global Vocabulary* method that considered all the unique strings in the training files was the best with an accuracy of 91.02%. The low accuracy just for the nearest match method could be due to the fact that we allowed a high degree of freedom for detecting near similar strings. Also, as evidenced by our experiments, by using exact matches of prominent strings, we are getting good accuracy. So, there is no need to perform extra computations to find nearest matches and use the *Nearest Match* method. Even though *Exact Match: PSS* cannot achieve the accuracy as obtained by *Exact Match: Global Vocabulary*, PSS highly reduces the dimension of the prototype vector. So, *Exact Match: PSS* will help to decrease the computational time while still being highly accurate.

The malware dataset was collected by using the 5-way vendor agreement method. For the time being, we chose to believe that we are taking a good gold standard as each malware sample's labeling is agreed upon by at least five anti-virus vendors. But we do acknowledge that there might be some labeling errors in this dataset, especially when n is low. During prototype creation, this might have instilled some errors into our models. Also, we could be correctly classifying the malware even when our label does not agree with the vendor

label. In the future, we need to explore a more rigorous approach to gather a good gold standard dataset. Although our family labels were sourced from the 5-way vendor agreement method, our system can very well be used when fewer than 5 vendors are in agreement. Also, it could be applied to newly arrived malwares, before most of the vendors have given their verdict.

Our results clearly demonstrate that despite of these shortcomings in the dataset, we can still obtain a very good accuracy by making use of string information. Hence we can safely say that string information can be used in the classification of malware.

7 Acknowledgments

We thank Kevin R Mitchem for providing us with the malware dataset. This research was partially funded by The Office of Naval Research under grant N00014-12-1-0217.

References

1. Park, Y., Reeves, D., Mulukutla, V., Sundaravel, B.: Fast malware classification by automated behavioral graph matching. In: Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research. CSIIRW '10, New York, NY, USA, ACM (2010) 45:1–45:4
2. Bailey, M., Oberheide, J., Andersen, J., Mao, Z., Jahanian, F., Nazario, J.: Automated classification and analysis of internet malware. In Kruegel, C., Lippmann, R., Clark, A., eds.: Recent Advances in Intrusion Detection. Volume 4637 of Lecture Notes in Computer Science. Springer Berlin Heidelberg (2007) 178–197
3. Tian, R., Batten, L., Islam, M., Versteeg, S.: An automated classification system based on the strings of trojan and virus families. In: Malicious and Unwanted Software (MALWARE), 2009 4th International Conference on. (2009) 23–30
4. Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C.: Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. Information Security Technical Report **14** (2009) 16 – 29
5. Han, E.H., Karypis, G.: Centroid-based document classification: Analysis and experimental results. In: Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery. PKDD '00, London, UK, UK, Springer-Verlag (2000) 424–431
6. Debole, F., Sebastiani, F.: Supervised term weighting for automated text categorization. In: Proceedings of the 2003 ACM symposium on Applied computing. SAC '03, New York, NY, USA, ACM (2003) 784–788
7. Wei, C., Sprague, A., Warner, G.: Clustering malware-generated spam emails with a novel fuzzy string matching algorithm. In: Proceedings of the 2009 ACM symposium on Applied Computing, ACM (2009) 889–890