Classifying Windows Malware with Static Analysis

Maryann Gong, Uma Girkar, Benjamin Xie
Department of Electrical Engineering and Computer Science
Massachusetts Institute of Technology
Email: {mmgong, umag, bxie}@MIT.edu

***Abstract*—Adversaries are creating new types of malicious software, or *malware*, at an increasing rate. The amount and diversity of malware are making classic security defenses more and more ineffective. Our objective is to analyze multiple dimensions of Windows executable programs. We used machine learning techniques to build and train a classifier to identify malicious software from benign software. We find that the best mixed features classifier we created performs over 10% better than the average performance of 57 commercial anti-virus programs we tested against. In analyzing the performance of the commercial anti-virus programs, we also find bias in the design decisions that sacrifices security in favor of usability.**

I. Introduction

Problem Statement: In the past 2.5 years (since 2013), there have been more new kinds of malware created than the ten years before that combined [1], as depicted in Figure 1. The need to detect previously unseen malware is growing. Of particular concern are the Windows operating systems, which run on over 85% of desktops today [2]. The proliferation of smart mobile devices further increases the attack surface. It is believed that 80% of infected mobile devices have been traced to connections to Windows computers and laptops [3].

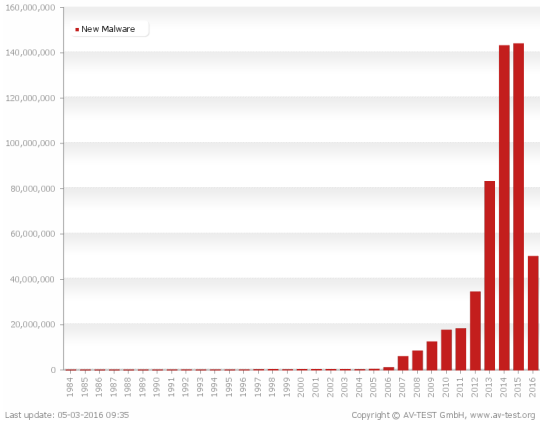

Fig. 1: Number of new malware threats by year [1]

Our objective is to use supervised machine learning techniques to create a classifier that accurately identifies malware, even if the malware is new and previously unseen. We take steps towards this objective by creating and training a classifier that uses static analysis to detect malware in the form of Windows portable executable files (`.exe` files). Using static analysis techniques, we analyze files without running them and extract features used for classification. We find that our results better average results from 57 publicly available anti-virus services and also identify biases in the design of anti-virus services that pose threats to users. We also compared the classifier we built to other common machine learning classifiers and find that we were able to outperform them as well.

A. Security/Attack Model

Attackers write malicious software that is designed to compromise, infiltrate, or damage a computer system without the owner's consent, and, in some cases, knowledge. This software can infect a machine via various methods, such as cross site scripting attacks, through infected removable drives, or through spam emails [4]. We assume that an adversary has *no direct access* to the target computer system and does not have the ability to login to the computer, directly or remotely. Once installed, malware can be used by adversaries to steal sensitive information, perform computations (botnets), and leverage money from the user (ransomware). Ideally, adversaries should have none of these powers to access the computer system itself. In some cases, adversaries may be allowed to observe network traffic to and from the computer system, but they should not be able to observe and modify components within the system.

The goals of static malware detectors are to identify a wide variety of existing malware (if not all), identify newer modifications or variants of existing malicious software, and identify completely new malware, while also identifying benign software as safe all without running the program in question.

Users are assumed to be honest in that they will not attempt to disable or compromise malware detection programs. We assume that they will accidentally attempt to download malware. Upon attempting to install a malicious program, the malware detection software should abort the installation process, quarantine the program, and alert the user the file was suspected to be malicious. Suspected malware can only be run if the informed user understands the risks and chooses to run it anyways.

II. Previous Work

Traditional approaches to malware detection can be categorized into two categories: static analysis and dynamic analysis.

Static analysis involves analyzing a suspected file without running it, as first proposed by Lo 1995 [5]. Static analysis is more secure and efficient because it can be done without running a potentially malicious file. The greatest limitation to static analysis is obfuscation of code.

To prevent detection, malware creators *obfuscate* code by deliberately attempt to write and package it such that the malicious code is difficult to read without being run first. Obfuscation techniques have shown to be successful against common anti-virus programs such as Norton because they focus on matching *syntax*. Recent research has focused on identifying malware by analyzing the *semantics* of programs [6]. Static analysis can better common obfuscation techniques, but Moser 2007 has shown that generic analysis of obfuscated code is NP-hard [7].

Dynamic analysis runs programs in isolated or limited settings to determine if programs are malicious. More specifically, dynamic analysis typically executes malware binaries and decides during runtime when the program has been unpacked if it is malicious or not. At this point, most obfuscation has been removed [8]. Examples of dynamic analysis techniques include monitoring threads and processes and simulating common Internet protocols (INetSim). The main limitation to dynamic analysis is that limited (often single) paths of execution are examined, leading to an incomplete picture of malware activity.

A. Machine Learning for Malware Detection

Machine learning for malware analysis can be roughly categorized into two categories: classification and clustering.

Reserachers have come up with a number of machine learning-based malware detection algorithms in recent years. One such framework for the automatic analysis of malware behavior with machine learning [8] uses clustering to automatically identify novel classes of malware with similar behavior. Another algorithm targeted towards malware detection in Android devices extracts features from packaged Android applications. It then uses these extracted features to train an One-Class Support Vector Machine [9]. The main idea behind this algorithm is to generate a classifier that will classify the majority of the training data as positive and classify testing or training data as negative only if it is significantly different from the training data. The algorithm is constructed this way because known, benign applications are far more accessible than malicious ones.

Clustering yields a faster runtime performance in practice, but it requires processing files in batches. Recent work by Reick 2011 has used both classification and clustering techniques for more incremental analysis [8]. The features analyzed include analyzing strings in binary executables as well as the binaries themselves.

III. Methods

A. Dataset

Our dataset consists of a total of 3,294 Windows Portable Executable (PE) files. This dataset is split between 2,382 known, verified malware programs and 912 known, benign software programs. This dataset was collected and provided by the company Cyphort [10], a computer and network security company focused on fighting advanced persitant threats (APTs). We split this dataset 80% and 20% into a training and test set respectively. In our test set, we maintained a similar class balance as in our entire dataset, meaning the test set should be approximately 70% malware and 30% benign files.

B. Features

1) Dynamic-Link Libraries: One of the key sets of features we used is a list of dynamic link libraries used by each Windows executable. A Dynamic Link Library, or DLL, is a shared library that contains code, resources, data, or any combination of the aforementioned. They share the file extension .dll. These DLLs are used by Windows programs to share functionality and resources between themselves. Much of the functionality of Microsoft Windows Operating Systems is provided through use of DLLs [11].

Commonly used resources, code, functions, or data are packaged in DLLs, which can be bound or linked to some Windows program. USER32 and WSOCK32 our examples of DLLs. USER32 is a user interface library and WSOCK32 is a Windows socket application programming interface [12]. The use of these shared libraries is motivated by resource reduction and modularity. By dynamically linking these shared modules at runtime, programs can avoid code duplication, improving overall program performance. In addition, these separately imported, shared libraries encourage modularity, allowing for DLLs to be individually fixed and updated without disrupting the applications that use them [13].

We can infer certain characteristics about a program through the set of DLLs it uses. We used these DLLs to help distinguish between benign and malware executables.

We used GNUs objdump command to extract the list of DLLs used by a Windows executable [14]. The program headers in the program executable files contain information such as the file size and also the list of DLLs used by the program. We were able to parse the DLLs from the output generated by the objdump command.

We use a bag-of-words approach when representing the DLLs for each file. That is, we look at frequency and existence of DLLs without considering the order. Once we extracted the list of DLLs for each file, we constructed a feature vector for each file where each index corresponded to a specific DLL whose value was either 1 or 0, indicating whether that DLL was present in the file or not. The overall dimensionality, or number of DLLs, found for our data set was 414.

2) Strings: Portable Executable format files also contain printable strings within their headers and the bodies of the files. These printable strings could reveal information regarding the nature of the program, whether it is benign or

TABLE I: Most common strings from benign and malware files

Benign	Malware
Sleep	GetProcAddress
GetCurrentProcess	ExitProccess
ExitProcess	GetLastError
CloseHandle	SVW3
LoadLibraryA	CloseHandle

```
1f0e 0eba b400 cd09 b821 4c01 21cd 6854
7369 7020 6f72 7267 6d61 7220 7165 6975
6572 2073 694d 7263 736f 666f 2074 6957
646e 776f 2e73 0a0d 0024 0000 0000 0000
454e 3c05 026c 0009 0000 0000 0302 0004
0400 2800 3924 0001 0000 0004 0004 0006
000c 0040 0060 021e 0238 0244 02f5 0000
0001 0004 0000 0802 0032 1304 0000 030a
```

Fig. 2: Example of hexdump command output.

malicious. These strings could be part of the file name, file author signatures, bits of recycled code, or information on system resources used [12]. We were able to extract these printable strings from file using the GNU strings command [14].

In Table I, we display the most common printable strings found in both our benign training dataset and our malware training dataset.

We assume a bag-of-words approach when constructing our strings representation of our dataset. Using the set of extracted strings from our dataset, we constructed binary attribute vectors where each string represents a binary feature. For each file, we indicated whether the specific string was present or not in the file with either 1 or 0.

3) Byte Sequences: The portable executable files contain binary representations of the machine instructions. For our byte sequences set of features, we converted the binary code file into a hexadecimal file. In this representation, each machine instruction is a 2-byte word composed of two, 2-digit hexadecimal numbers. Whereas the DLL features set revealed resource information for a program, byte sequences represent which specific machine instructions are used and called in a program [12]. We reasoned that benign and malware programs would display distinct patterns of machine instruction calls.

To extract the byte sequences, we utilized the hexdump linux command to convert the binary code into lines of 4-digit hexadecimal words [15]. To simplify computations, maintain program efficiency, and reduce feature dimensionality, we consider a random subset of the machine instruction words from the program file. First we select a random subset of 100 lines from the program file. Then, from each selected line, we randomly choose a subset of 10% of the words from the line. We again assume a bag-of-words representation for our features. The feature vector for each file is a binary attribute vector where each machine instruction word from the dataset represents a particular attribute in the vector. For each attribute for a file, we indicate the presence of the specified word with 1 and its absence with 0.

C. Classification

1) Naive Bayes: We used the Naive Bayes classifier on each of our feature sets individually to distinguish between malicious files and benign files. The classifier is based on the Bayes rule and relies on the assumption that the predictors are not dependent on each other within each class. In other words, in order to determine $P(X|Y)$ where $X = < X_1, X_2, ...X_n >$, the algorithm assumes that all the X_is are conditionally independent of each other given Y. Each X_i is also independent of all subsets of the other X_js given Y. However, this classifier has been found to work well even when the conditional independence assumptions do not hold.

The Naive Bayes method uses the training data to estimate the parameters of a probability distribution. For the prediction step, the Naive Bayes classifier then computes the posterior probability of the testing set for each class. The test data is classified as the class with the highest posterior probability.The formula for computing the posterior probability by the Naive Bayes algorithm is derived as follows:

$$P(Y = Y_k|X_1X_2...X_n) = \frac{P(Y = Y_k)P(X_1X_2...X_n|Y = Y_k)}{\sum_j P(Y = Y_j)P(X_1X_2...X_n|Y = Y_j)}$$

This equation comes directly from the Bayes rule. Now using the conditional independence assumptions made, we obtain the following equations.

$$P(Y = Y_k|X_1X_2...X_n) = \frac{P(Y = Y_k)\prod P(X_i|Y = Y_k)}{\sum_j P(Y = Y_j)\ \prod P(X_i|Y = Y_j)}$$

We can manipulate this equation to find the class with the highest posterior probability to get:

$$Y \leftarrow argmax_{y_k} P(Y = Y_k)\prod P(X_i|Y = Y_k)$$

We can ignore the denominator because it is not dependent on y_k.

2) Combining Feature Set Classifiers: We utilized three different methods of combining our main feature sets of DLLs, strings, and byte sequences. The first method simply merges the individual feature vectors into one large vector. The second method trains a classifier on each individual feature set, equally weights the classifiers, and takes the majority vote of their output classifications. The final method again trains a classifier on each individual feature set, then averages the outputted posterior probabilities of each class, and selects the class that maximizes the average posterior probability.

a) Merging Feature Vectors: In this method, we merged the three separate feature vectors into one large feature vector. We appended the binary attribute features for DLLs, strings, and byte sequences into one large, combined feature vector. The dimensionality of the individual DLLs, strings, and byte sequence feature vectors were 414, 1502, and 445 respectively. The combined feature vector has dimensionality of 2361. We then ran our same Naive Bayes classifier on this new combined feature vector to determine the final class predictions. This combined classifier does not associate attribute features with their original feature sets and does not leverage the information this relationship might yield.

b) Equally-Weighted Voting: For the equally-weighted voting scheme, we first trained and tested our Naive Bayes classifier on each of the individual feature sets to get three separate sets of class predictions. Then to combine these classifiers we took the majority for each test sample. For example, if test sample x was labeled as benign by two of the feature set classifiers and malware by the remaining classifier, then our equally-weighted voting classifier would classify test sample x as benign. This method naively gives each set of features equal importance.

c) Maximum Averaged Posterior Probabilities: Finally, for our last combined classifier method, we utilized the average of the posterior probabilities on each feature set. Similarly to the equally weighted voting scheme described above, we first applied our Naive Bayes classifier individually on each of the feature sets. Recall from the previous section where we described the algorithm, Naive Bayes calculates the posterior probability of a sample belonging to each class (benign or malware) given the set of features or observations for that test sample, and then outputs the class with the maximum posterior probability. For this combined classifier method, we average the calculated posterior probabilities from each feature set and then select the class with the maximum average posterior probability [16]. Although each feature set technically has an equal weight, the level of confidence of the classifiers of each feature set does affect the overall weighting of the output classification. If a classifier on a particular feature set is very confident (has a high posterior probability for a class), then this will be reflected in the combined classifiers posterior probabilities.

IV. Results

We tested and evaluated our Naive Bayes classifier on all three individual feature sets of dynamic link libraries, strings, and byte code sequences. The results from the three individual feature sets are displayed in table II. Next, we tested three different classifier combination methods (as described in the previous section) on the combined data set consisting of all the feature sets. The results from these three methods are shown in table III. For each experiment, we count the number of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN). True positives and true negatives refer to malicious and benign samples respectively that were correctly classified. False positives refer to benign samples that were misclassified as malware. Similarly, false negatives are malicious samples that were misclassified as benign.

TABLE II: Performance on Individual Feature Sets

Feature Set	TP	TN	FP	FN	Accuracy
DLLs	456	66	116	20	79.33%
Strings	176	182	0	300	54.41%
Byte Sequences	469	98	84	7	86.17%

The highest performing feature set consisted of the byte sequence features, while the lowest scoring features were the strings. We anticipated that the byte code sequences may have the highest accuracy due to the nature of the information it brings. Specifically, byte sequences reveal information on each individual machine instruction called by the program. Furthermore, this particular feature set had the largest dimensionality at 1,502, which was more than three times the dimensionality of the other two feature sets. We discuss a possible explanation for the low performance of our classifier on our strings feature set due to code obfuscation later in our section on limitations.

Since the eventual goal of our tool is to prevent user's machines from becoming compromised by malware, we wish to prevent malicious software from being misclassified as benign. Therefore, the first priority is to minimize false negatives. Our byte sequence features have by far the lowest false negative rate at approximately 1.06%, whereas our strings features has a very high false negative rate at 45.59%. The DLLs feature set also yielded a relatively low false negative rate at 3.04%. However, it is significant to note that the strings feature set had zero false positives, meaning no benign programs were misclassified as malware. This could also indicate that the classifier on the strings feature set was slightly biased towards negative, or benign, classification.

TABLE III: Performance on Combined Feature Set Classifiers

Combination Method	TP	TN	FP	FN	Accuracy
Merged Feature Vect.	316	167	15	160	73.40%
Eq. Weighted Voting	455	66	116	21	79.18%
Maximum Avg. Posterior	454	128	54	22	88.45%

We combined our separate feature sets by merging the feature vectors themselves, combining the classifiers in an equally weighted voting scheme, and by averaging the posterior probabilities outputted by each classifier. These three methods are described in greater detail in the Methods section. The performance of the three different combination methods align with their levels of sophistication.

The simplest method of combining the feature sets was by explicitly merging the actual feature vectors themselves. This resulted in a 73.40% accuracy rate, which is lower than the individual performance of both DLLs and byte sequence features. The high dimensionality of the combined feature set may cause overtraining, negatively impacting performance. Also, similar to the strings features, there is a relatively low false positive rate and high false negative rate. This indicates

that the strings feature set had a negative influence on the overall classifier using this method.

The next method was slightly more complex. The equally weighted voting scheme outputted the majority decision of the three separate feature sets. Accuracy improved over the first method to 79.18%. However, this is still lower than the individual performance of the byte sequence feature set and comparable to the DLL feature set. This can be attributed to the fact that despite strings' relatively poor performance, it still has the same weight vote as the other classifiers.

The last method, taking the maximum of the averaged posterior probabilities, was the most sophisticated and used the output of the individual feature set classifiers at a finer granularity past the hard, class assignment to the actual posterior probabilities. The performance of this method was the highest out of all combination methods and all individual feature set classifiers at 88.45%. It had relatively low false negative and false positive rates at 3.3% and 8.2% respectively.

V. Discussion

A. Comparing to the State of the Art

We compare our performance to that of 57 publicly available Anti-Virus software using VirusTotal [17]. We analyze a sample of 1,484 files from our dataset (814 benign files, 670 malware files) and report our results in Table IV. We find the average accuracy of the 57 Anti-Virus programs to be 77.49%. We note that Anti-Virus programs tend to bias towards classifying files as benign, as evidenced by the strong performance in identifying benign files (true negative) and tendency to classify malware as benign (false negative). A likely explanation is the balance between usability and security. While false positives are safer for a system, they present a strain on the user as they become frustrated that their anti-virus program constantly warns them of files that are often trusted. False negatives are more dangerous to the system though, so the underlying justification for Anti-Virus companies is likely that some form of anti-virus software running on a system is better than none at all or a disabled anti-virus program. We argue that users should be informed of this design decision which places users at greater risk and perhaps have a choice of deciding "security levels" of their anti-virus to balance usability and security.

We did not compare our algorithm with other malware detection algorithms because our binary and malicious files did not match the format of the files required to run these algorithms. Additionally, it did not make sense to compare the accuracy between algorithms tested on different datasets. As a result, we compared our algorithm to other state of the art machine learning classifiers using the data obtained by extracting certain features from the original binary and malicious files. We tested the data with the support vector machine classifier [18], binary decision tree classifier [19], and the discriminant analysis classifier [20]. The resulting accuracies are displayed for the DLL features (Table V), strings features (Table VI), and byte sequence features (Table VII).

TABLE IV: Average Performance of 57 Commercial Anti-Virus Services (from VirusTotal)

TP	TN	FP	FN	Accuracy
0.56	0.95	0.05	0.44	77.49%

TABLE V: Other Classifier Performance on DLLs

ML Classifier	TP	TN	FP	FN	Accuracy
SVM	466	124	58	10	79.64%
Binary Decision Tree	444	100	82	32	79.94%
Discriminant Analysis	360	113	33	21	74.57%

TABLE VI: Other Classifier Performance on Strings

ML Classifier	TP	TN	FP	FN	Accuracy
SVM	440	130	52	36	86.63%
Binary Decision Tree	442	125	57	34	86.17%
Discriminant Analysis	438	116	66	38	84.19%

TABLE VII: Other Classifier Performance on Byte Sequences

ML Classifier	TP	TN	FP	FN	Accuracy
SVM	471	90	92	5	85.26%
Binary Decision Tree	460	76	106	16	81.46%
Discriminant Analysis	362	85	61	19	80.27%

The accuracy produced by the other machine learning classifiers significantly varied across the individual feature sets. For the DLL feature sets, the other machine classifiers performed roughly around the same as our classifier. Our classifier performed better than the support vector machine, binary decision tree, and discriminant analysis classifiers for the byte sequences feature set. Finally, all three of the other machine learning classifiers performed much better than our classifier on the strings feature set. However, our combined classifier still yields the highest accuracy overall.

B. Limitations

Program obfuscation may have provided a barrier to the extraction of strings from all our files (see II). Designers of malware programs sometimes attempt to avoid detection by obfuscating their code. When code is obfuscated, the program is packed in a manner that makes fields of the file hard to read. This can inhibit static analysis methods, such as reading printable strings from an executable. This code packing does not hinder the actual execution of the program because at runtime wrapper program unpacks the program. However, prior to runtime it is often difficult to determine whether a program is in fact packed or unpacked. Possible program obfuscation could adversely impact the efficacy of the strings features set [21].

C. Future Work

We only used the Naive Bayes classifier on the individual features sets to distinguish between malicious and benign files. Although the Naive Bayes classifier requires only a small amount of training data to estimate the parameters of a probability distribution and is easy to implement, it relies on the conditional independence of variables within each class. This likely resulted in a loss of accuracy as dependencies almost always exist within variables. Other more sophisticated boosting classifiers such as Adaboost or RobustBoost that can model the dependencies between these variables could be tested on the individual feature sets. It would also be interesting to test some novel malware detection algorithms such as those cited in the previous works section on the benign and malicious files dataset directly.

Additionally, since false negatives are much more harmful to the user than false positives, further work could involve modifying the algorithm to focus on minimizing false negatives. The potential drawback to doing this however is that if the chance of a false negative is small enough, further tweaking the algorithm to minimize the number of false negatives could significantly worsen the performance of the algorithm. To top it off, minimizing the number of false negatives could lead to an increase in the number of false positives. Although false positives do not compromise the security of the system, they can discredit the integrity of the classifier and hinder usability.

VI. Conclusion

We investigated the application of static analysis and machine learning techniques to identify and distinguish malicious Windows software from benign software. The static nature of our methods allows our tools to analyze software portable executables without actually running the program and potentially compromising a user's machine.

Our main contributions include applying various machine learning techniques to the problem of malware detection, comparing our results to current state-of-the-art detection tools, and identifying a potential security vulnerability in the design decisions of current anti-virus software. We used data mining techniques to extract meaningful, static features from PEs and subsequently applied Naive Bayes on our extracted features to classify executables as malicious or benign. Furthermore, we compared our results to the anti-virus program Virus Total and other machine learning classifiers like support vector machines, binary decision trees, and discriminant analysis. Finally, we identified a security vulnerability in the design tradeoffs of current anti-virus software. Prioritizing the minimization of false positives over minimizing false negatives can allow malware to infiltrate a user's machine undetected.

As machine learning techniques progress and improve, we anticipate further improvement of machine learning performance in the detection of malware. We believe that further research in the application of these methods to security problems could lead to safer, more secure systems.

Acknowledgment

We thank Evangelos Taratoris (MIT EECS) for his guidance and support through this project. We also thank Dr. Fengmin Gong (Cyphort Inc.) for providing us with the data to make this research possible.

While we cannot explicitly share the dataset used in our analysis, this paper can be made publicly available immediately.

REFERENCES

[1] "AV-Test," "https://www.av-test.org/en/statistics/malware/", accessed May 1, 2016.

[2] Net marketshare. Accessed May 6, 2016. [Online]. Available: https://www.netmarketshare.com/operating-system-market-share.aspx?qprid=10&qpcustomd=0

[3] Alcatel-Lucent, *Alcatel-Lucent malware report shows significant rise in mobile infections via PCs and adware in first six months of 2015*. Alcatel-Lucent, 2015.

[4] Microsoft. How does malware infect your pc? [Online]. Available: https://www.microsoft.com/security/portal/mmpc/help/infection.aspx

[5] R. W. Lo, K. N. Levitt, and R. A. Olsson, "Mcf: a malicious code filter," *Computers & Security*, 1995.

[6] M. Christodorescu and S. Jha, "Static analysis of executables to detect malicious patterns," in *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, 2003.

[7] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*, 2007.

[8] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," *Journal of Computer Security*, 2011.

[9] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Intelligence and Security Informatics Conference (EISIC), 2012 European*, 2012.

[10] Cyphort. [Online]. Available: http://www.cyphort.com/

[11] M. Corporation. Dynamic-link libraries. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/ms682589(v=vs.85).aspx

[12] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," *IEEE Symposium on Security and Privacy*, 2001.

[13] M. Corporation. (2007) What is a dll? [Online]. Available: https://support.microsoft.com/en-us/kb/815065

[14] I. Free Software Foundation. (2014) Gnu binutils. [Online]. Available: https://www.gnu.org/software/binutils/

[15] I. O'Reilly Media. linux devcenter hexdump. [Online]. Available: http://www.linuxdevcenter.com/cmd/cmd.csp?path=h/hexdump

[16] D. M. Tax, M. van Breukelen, R. P. Duin, and J. Kittler, "Combining multiple classifiers by averaging or by multiplying?" *Pattern Recognition*, 200.

[17] Virustotal. [Online]. Available: https://www.virustotal.com/en/documentation/public-api/

[18] J. Sukens and V. J., "Least squares support vector machine classifiers," 1999.

[19] S. Shlien, "Multiple binary decision tree classifiers," 1989.

[20] M. Sebastian, R. Gunnar, and J. Weston, "Fisher discriminant analysis with kernels," 1999.

[21] L. Mehta. (2016) Static malware analysis. [Online]. Available: http://resources.infosecinstitute.com/malware-analysis-basics-static-analysis/#article