A Dynamic Heuristic Method for Detecting Packed Malware Using Naive Bayes

Ehab M. Alkhateeb
Dubai, United Arab Emirates
ehabalkh@gmail.com

Mark Stamp
Department of Computer Science
San Jose State University
California, United States
mark.stamp@sjsu.edu

***Abstract*—In this paper we consider a heuristic malware detection method based on dynamic analysis of API calls. We utilize a naïve Bayes classifier to distinguish between benign and malware samples, and Levenshtein distance is shown to increase the effectiveness of the technique. For comparison, we consider commercial anti-malware products. We provide experimental evidence of the strength of our technique based on a substantial malware dataset. We show that our approach achieves particularly impressive results in detecting packed malware samples.**

***Keywords*—Malware, dynamic analysis, naïve Bayes classifier, Levenshtein distance, packed malware.**

I. Introduction

Malicious software, or malware, is an umbrella term used to refer to different forms of intrusive software. There are many varieties of malware, including Trojans, spyware, viruses, and rootkits. Today, malware is more powerful and difficult to detect than ever—without question, malware detection is one of the most important topics in computer security [1].

Commercial antivirus (AV) or anti-malware products depend primarily on signature-based detection, which is ineffective against many advanced forms or malware [2]. According to AV-Test there are more than 390,000 new malware samples created every day [3]. Heuristic detection offers an alternative to signature scanning. The motivation of the research presented here is to improve on common heuristic detection techniques by developing a strategy that offers a strong detection rate, and is efficient in terms of both computation and storage requirements. In this paper, we empirically detect malware by an "intelligent" heuristic method. Our experiments are designed to highlight the effectiveness and practicality of the proposed method. Our approach performs well in general, and is particularly impressive in the case of packed malware samples.

This paper is divided into five sections. Section II covers related work and relevant background topics. In Section III, we outline our proposed heuristic detection method, while Section IV presents our experimental results. Finally, in Section V, we give our conclusions and briefly consider future work.

II. Background

In this section we outline common malware detection approaches and techniques. Here, we also discuss naïve Bayes, heuristic analysis, and present the measures that we use to quantify the success of our experiments.

A. Malware Detection Techniques

Malware detection approaches can be divided into the two broad categories of static and dynamic, depending on how the relevant features are extracted. In static analysis the malware is investigated without executing (or emulating) the code, while in dynamic analysis the tested file is executed and monitored (typically, in a virtual environment) to extract feature that can then be examined for indications of suspicious behavior.

Malware detection techniques can be further categorized as signature-based, anomaly based, heuristic, or behavioral based [4]. Signature-based detection is the most elementary, since we simply scan for a fixed pattern. A disadvantage of signature detection is that it cannot succeed against previously unseen malware. Anomaly-based detection attempts to discern malware-like patterns, and thereby classify samples. Behavioral detection is generally based on monitoring the interaction of the software with its environment. The emphasis here is on heuristic detection.

Heuristic analysis is a method employed by many computer antivirus programs. One potential advantage of heuristic detection is that it can be used to detect previously unknown computer viruses, as well as new or metamorphic variants of existing viruses [5].

B. Code Obfuscation

Code obfuscation consists of making a program obscure, and can operate on both program data and control flow [6]. There are many different ways that malware writers can use to obfuscate their code. Packed cide is a well known method to obfuscated malware, and packed malware can be challenging to detect [7].

C. Application Programming Interfaces

An application programming interface (API) is a method or protocol used for communication between different software components. APIs can serve as a feature to help determine whether a program is malicious or benign. A major advantage of API based analysis is that APIs characterize important aspects of the code, and are relatively difficult to obfuscate.

D. Levenshtein Distance

In [8], Vladmir Levenshtein proposed the well-known distance measure that bears his name. Levenshtein (or edit) distance is based on the minimum number of insertions, deletions, and substitutions required to transform one string into another.

Let L be the Levenshtein distance function. Then, for example, we have

$$L(\texttt{sea}, \texttt{set}) = 1,$$

since we can simply substitute `t` for `a` to transform `sea` to `set`. As another example,

$$L(\texttt{trail}, \texttt{fails}) = 3,$$

since we could delete the `t`, substitute `f` for `r` and insert `s` to transform `trail` into `fails`, and no fewer number of operations will succeed. The work in [9] gives more details about Levenshtein distance.

In this research, we use Levenshtein distance to measure the distance between feature vectors.

E. Naïve Bayes Classifier

Naïve Bayes is a supervised learning algorithm that is based on conditional probabilities. This method has been used with success in a wide variety applications, ranging from spam filtering to text categorization to medical diagnosis.

In general, to determine the most likely class of the input or feature vector $\mathbf{x} = (x_1, \ldots, x_n)$, we first compute

$$\begin{aligned} y &= \operatorname*{argmax}_{k\in\{1,\ldots,|C|\}} P(C_k|x_1,\ldots,x_n) \\ &= \operatorname*{argmax}_{k\in\{1,\ldots,|C|\}} P(C_k)\prod_{i=1}^{n} P(x_i|C_k) \end{aligned} \tag{1}$$

The desired classification is then given by

$$\begin{aligned} y &= \operatorname*{argmax}_{k\in\{1,\ldots,|C|\}} \log\bigl(P(C_k|x_1,\ldots,x_n)\bigr) \\ &= \operatorname*{argmax}_{k\in\{1,\ldots,|C|\}} \log\left(P(C_k)\prod_{i=1}^{n} P(x_i|C_k)\right) \\ &= \operatorname*{argmax}_{k\in\{1,\ldots,|C|\}} \left(\log\bigl(P(C_k)\bigr) + \sum_{i=1}^{n}\log\bigl(P(x_i|C_k)\bigr)\right) \end{aligned} \tag{2}$$

The direct estimation of $P(x_i\,|\,C_k)$ from a given set of training examples may be difficult, particularly when the feature space is high-dimensional. Therefore, approximations are widely used, such as the simplifying assumption that features are independent of the class [10]. This assumption yields the naïve Bayes classifier, which can be defined as

$$y = P(C_k)\prod_{i=1}^{n} P(x_i\,|\,C_k) \tag{3}$$

We note in passing that a hidden Markov model (HMM) can be viewed as a sequential version of naïve Bayes [11].

F. ROC Analysis and AUC

To construct a receiver operating characteristic (ROC) curve from a scatterplot, we plot the true positive rate versus the false positive rate, as the threshold varies through the range of values. ROC curves can be used to compare the performance of binary classifiers. In particular, the area under the ROC curve (AUC) gives us a statistic that can be interpreted as the probability that a randomly selected positive instance scores higher than a randomly selected negative instance [12]. In this paper we use the AUC as our primary measure of success.

G. Related Work

Machine learning techniques have been used in malware classification based on static or dynamic features. Examples of popular features include opcodes, API calls, network packets, call graphs, and so on. In this section we discuss previous works related to malware classifiers based on API analysis and machine learning algorithms, based on static or dynamic features.

The work in [13] suggests that dynamic heuristic analysis is better than static analysis for detecting large scale attacks and sophisticated malware.

A dynamic analysis method is presented in [14], where the authors employ DNA sequence alignment algorithms to compare API call patterns. While the method shows good detection results, it relies on an extensive behavioral analysis and does not appear to be applicable to packed malware. Also this technique has a high cost in terms of local machine storage, computation, and memory.

In the paper [15], the authors propose a heuristic method to detect malware in a Windows environment based on dynamic API call sequences. In this work, the sequences are arranged according to the frequencies of the individual API calls within a sample. The classifier used is a multi-dimensional naïve Bayes. The authors of [15] claim strong performance and efficient detection. However, their approach is resource intensive. In addition, a program is traced for a maximum of two minutes in a virtual environment, which opens the door to relatively simply evasive maneuvers by an attacker. Also, the authors do not consider obfuscated malware.

The authors of [16] present an approach based on dynamic API call frequencies. The main disadvantage of this work is that significant computation resources are required, as large hash tables are constructed for use in classification.

Our goal is to improve on some of the techniques discussed above. In particular, our proposed heuristic method is able to effectively detect packed malware, while being highly efficient.

III. Proposed Method

In this section we discuss our proposed heuristic detection method, which is based on dynamic analysis of API call frequencies, with naïve Bayes and Levenshtein distance used for classification. Our method can be viewed as a refinement process, where we group closely related API calls in the samples under consideration. During the training phase, we create a table to store all unique API calls found in the superset of malware and benign samples. The subset of API calls that are most strongly associated with malicious behavior is extracted.

We adopt a supervised learning model based on a naïve Bayes classifier. Note that in the training phase we construct the set of API calls that are associated with malicious behavior and we determine their corresponding probabilities using the relevant naïve Bayes parameters. In the testing phase, data is also processed using naïve Bayes, with the classifier used to assign a score to each tested sample (i.e., executable file). We then flag each sample as either malicious or benign based on a threshold. Both the training and testing phases are discussed in more detail in below.

A. Training Phase

In the training phase, we create a list of API calls that are associated with malicious behavior. This list, which we denote as LMB-API, is derived from a set of characteristics assigned to API calls. These characteristics are discussed in detail below.

In the training phase, we begin by examining the binary file in a virtual environment, using an API hooking tool to trace the API calls. While tracing the API calls, a log file is created to store the API calls. We refines the extracted API calls, extracting API names and eliminating repetition among the API call names. This refinement process is essential for accurate data collection.

The refined log file is scanned in order to update the LMB-API with any previously unseen API calls. When completed, the LMB-API is a detailed table that contains all unique API calls that have been collected during the training phase, and their characteristics. Specifically, the LMB-API includes the following information for each distinct API call.

- API Frequency — The frequency of each API call, over all training samples is, naturally, the API frequency. For a given API call, RECBEN is the frequency relative to benign samples and RECMAL is the observed frequency among malicious samples. For example, from Table I we see that `DhcpFreeLeaseInfo` appeared in 228 of the 1,000 malware samples, while in the 1,000 benign samples, it appeared only 39 times, thus its RECBEN is 0.039 while its RECMAL value is 0.228.
- Pattern — To identify the behavior of an API, Each API call is categorized into a pattern according to its malicious behavior. We assign these patterns according to the common behave of popular malware cateogries, this includes (Spyware, Trojan, Rootkit, Advertise, etc.) This helps to better categorize the type of any new threat. A pattern may include one or more sub category. For instance: Trojan.Zbot : This means that the heuristic analyzer identified a Trojan Zbot behave.
- X — A probability is calculated for each API call represented as X_i that appear in the naïve Bayes equation (2). That is each API call has a probability X_i of the class Malware related to a particular pattern.
- M — The probability of the malware class M in the naïve Bayes equation (2).
- Description — Descriptive information for an API call related to the malicious behavior found in malware samples where the specific API call appears.

Examples of API calls and their corresponding characteristics are summarized in Table I.

After the API call refinement step, the scanning process attempts to match each API call with those found in LMB-API. If an API call already exist in LMB-API, then we do the following.

1) The array variable RECMAL is incremented by 1 if the scanned file is malware sample.
2) The array variable RECBEN is incremented by 1 if the scanned file is benign sample.

On the other hand, if an API call is not in the LMB-API table, we create a new record in the table and assign its recurrence value and other details. Once all training samples have been scanned, the LMB-API table is sorted based on the most distinguishing malware features, and the probabilities X_i and M are calculated.

We select the best features based on the Levenshtein distances within the LMB-API table classed by their patterns, that is, we determine similar features using the Levenshtein equation with some restriction. We utilized Levenshtein distance to select best features by reading each API call sorted in ascending order by its probability within its malware category. The algorithm initially reads each API call alone and compares it with all other API calls within same pattern—if a similar API is found we group it under the API call. A similar API is identified if it achieves a specific Levenshtein distance and string length. This length is calculated when the length of the API call is “reasonable” to

Algorithm III.1: Training phase

```
Data: File
Result: log_file; new_record; update record
$i \leftarrow 1$
$j \leftarrow 1$
Create ($log\_file$)
Refinement ($log\_file$)
while EOF($log\_file$) == false do
    if log_file_API[$i$] exists then
        UpdateRecord(RECMAL, RECBEN)
    else
        CreateRecord(log_file_API[$i$], RECMAL
        , RECBEN, $PatternFamily$, $Description$)
    $i \leftarrow i + 1$
/* Rearrange LMB-API */
$k \leftarrow 5$ /* constant ratio value */
while RECMAL / $k \geq$ RECBEN do
    TakeValues
for 1 to NumberOfITems($X$) do
    FindProb($X_j$)
    UpdateRecord($X_j$)
    $j \leftarrow j + 1$
FindProb($M$)
UpdateRecord($M$)
Function CompetitiveFeaturesLD($API_n$):
    /* Select most prevalent features */
    $Q \epsilon EmptyArray$
    $u \leftarrow True$
    $c \leftarrow n$
    $h \leftarrow 3$
    while (c $\geq$ 0) do
        $l \leftarrow c$
        $Y \leftarrow API$
        while ($l \geq 0$) do
            $w \leftarrow$ LevenshteinDistance($API_c$,$Y_l$)
            if (Len($Y_l$) - Len($API_c$)) $\leq h$ And
             (Len($Y_l$) - Len($API_c$)) $\geq -h$ then
                if $w \geq$ Len($Y_l$)/$h$ then
                    for 1 to NumberOfITems($Q$) do
                        if $Q_f = Y_c$ then
                            $u \leftarrow False$
                        else
                            $u \leftarrow True$
                        $f \leftarrow f + 1$
                    if $u = True$ then
                        $z \leftarrow API_c$
                        $Q \leftarrow Q + z$
            $l \leftarrow l - 1$
        $c \leftarrow c - 1$
    for 1 to $n$ do
        RemoveItem($API_n$) if found in $Q$
    Return $API_n$
```

measure, specifically, the compared API call distance should not exceed a three character change with more or less and the string size should be at least more than half of the length of the API call under consideration. For example, the API call `IsBadWritePtr` would be similar to the features `IsBadStringPtrA` and `IsBadStringPtrW`, based on Levensthein distance. Using this information, we reduce the size of the feature set by grouping `IsBadWritePtr`,

TABLE I: LMB-API repository samples

API	RECMAL	RECBEN	X	Pattern	Description
`DhcpFreeLeaseInfo`	228	39	0.0123	Spyware	IPv4 address
`RtlIpv4AddressToStringA`	223	26	0.0120	Spyware	IPv4 address
`CreateToolhelp32Snapshot`	165	32	0.0109	Trojan	Process injection
`Process32First`	162	22	0.0087	Trojan	Process injection

`IsBadStringPtrA`, and `IsBadStringPtrW` under the single heading of `IsBadWritePtr`. Algorithm III.1 summarizes the training phase as discussed here.

At the end of the training phase, we have selected features as shown in Table I, above. From a practical point of view, the training phase is conducted in a different environment than the user local machine environment. The user local machine will inherit this table to later be used for classifying new samples by the heuristic analyzer.

B. Testing Phase

In the testing phase, we first execute the sample binary file in a virtual environment. The testing phase is conducted on the user local machine. A log file is created and the API calls are traced using a hooking tool. The log file is then processed using the same refinement process as in the training phase. In the testing phase we will look for API call names that match with the LMB-API list—if one or more is found, the API call will be recorded along with its probability. Hence, one API might contain one or more pattern, and the testing phase will select the probabilities corresponds to the highest probability pattern. Thus, A vector of match probabilities V is generated by comparing each API call in the refined log file to the LMB-API list—for each API call that matches we collect its probability X_i from the LMB-API table and insert it as V_i into the vector V. Recall that the LMB-API in Table I was generated in the training phase. The vector V is used to compute a score from Bayes formula as given in equation (2). The scoring process is summarized in Algorithm III.2, where the score is computed as

$$\text{score} = -\log\Bigl(P(M)\prod_{i=1}^{n} P(V_i|M)\Bigr) \tag{4}$$

Finally, if the score is less than a specified threshold, the file is flagged as malicious; else it is classified as benign.

IV. Experiments and Results

In this section we discuss our experiment and the results obtained using the proposed malware detection method as outlined in Section III. We first present additional specifics on our experimental design, then we give malware detection results, and finally, we consider obfuscation in the form or code packing. Our technique stands out from previous work in the obfuscation experiments.

A. Setup

Our experiments are based on 22,000 labeled samples, split between malware and benign as follows. We have 7,000 malware samples that were obtained from VirusTotal and the Malicia dataset. We packed each of these malware samples with two packers, giving us an additional 14,000 packed malware samples. These malware samples are summarized in Table II.

The 1,000 samples in our benign set were obtained from online sources across many different software categories (e.g., multimedia, clipboard, systems, and so on). These benign samples are summarized in Table III.

Both the training and testing algorithms, which appear here as Algorithms III.1 and III.2, respectively, were coded using the VB.Net language. In addition, an Oracle VirtualBox was used to execute the binaries and Drltrace was used to trace API calls under a Windows 7 environment. The relevant hardware

Algorithm III.2: Testing phase

```
Data: File
Result: log_file; 0 for benign, 1 for malicious
$score \leftarrow 1$
$i \leftarrow 1$ /* Index */
Create ($log\_file$)
Refinement ($log\_file$)
/* Loop until end of file */
while EOF($log\_file$) == false do
    if API-exist then
        CheckScoreValue (log_file_API[i])
    $i \leftarrow i + 1$
$score \leftarrow score * M$
if $score \leq threshold$ then
    Return 1 /* malicious */
else
    Return 0 /* benign */
Function CheckScoreValue(log_file_API[i]):
    /* Function for finding the score */
    $X \leftarrow$ log_file_API[$i$].$X$
    $score \leftarrow score * X$
    Return score
```

TABLE II: Malware dataset

Family	Files
VirusTotal	1,000
Zbot	2,000
Winwebsec	4,000
VirusTotal Packed	2,000
Zbot Packed	4,000
Winwebsec Packed	8,000
Total	21,000

TABLE III: Benign samples

Bookmark managers	29	Browsers	34
Clipboard apps	14	IP scanners	16
System information	18	Maps/GPS	22
Networks miscellaneous	11	Network and bandwidth tools	29
System benchmarks	22	Camera apps	16
Multimedia audio players	43	Office tools text editors	85
Programming file editors	86	Password managers	30
Secure cleaning	55	System hard disk utilities	22
System launchers	16	Shutting down tools Setups	16
File managers	45	Database utilities	10
Desktop enhancements	23	CD DVD Blu-ray	20
Tools and DVD burning	19	Desktop enhancements	11
Clocks and time management	12	Internet remote utilities	19
Packers and protectors	43	System tweak	66
Software education	30	Video recording	36
Fax and telephony	32	Unit conversion	27
Video encoders	43	Total	1,000

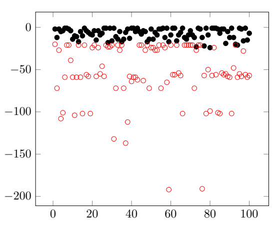
consisted of an Intel Core i5 with 4 GB RAM.

Note that Drltrace is a dynamic API call tracer for Windows and Linux applications, and is designed primarily for malware analysis [17]. In our experiments, we encountered many malware samples that attempted to hide their malicious calls by using a timer delay, pausing program execution, and similar techniques. Such behavior was clearly designed to evade dynamic analysis. It was necessary to take these issues into account so that we could perform accurate API tracing. The temporal criteria for tracing each executable ranged between 60 and 240 seconds. Also, we set a range of 70 MB to 150 MB for the size of the log file. In total, approximately 0.7 TB of (unfiltered) log data was collected across our set of malware and benign samples. After refinement of this data, as discussed in Section III-A, the size was reduced to a much more reasonable 15.4 MB for the 1,000 benign samples and 185 MB for the 21,000 malware samples. Hence, non-filtering the log file can cause incorrect occurrences for either benign or malware samples. Thus, this will cause incorrect scores and worsen incorrect detection.

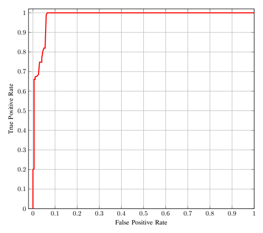
B. Detection Experiments

We employ five-fold cross-validation in all of our experiments. That is, the malware dataset was partitioned into five equal-sized subsets, say, S_1, S_2, S_3, S_4, and S_5. In the first fold, we train using S_1, S_2, S_3, and S_4, and we use the resulting model to score S_5 and the benign set. This process is repeated five times, with a different subset reserved for testing in each fold. The results of each experiment is plotted in the form of an ROC curve and we calculate the area under the curve (AUC).

Examples of scatterplots of scores and ROC for each of the Zbot samples versus our benign set are given in Fig. 1 (a), (b), respectively. These results show good separation between the classes, which we next make precise in terms of the AUC statistic for ROC curves.



(a) Zbot Scores



(b) Zbot ROC

Fig. 1: Score scatterplots

Our naïve Bayes classifier attained an AUC of 0.87 for VirusTotal dataset. Including the Levenshtein distance calculation into our naïve Bayes classifier increased the AUC to 0.91. From these results, we see that the proposed heuristic method using a naïve Bayes classifier and Levenshtein distance performs best.

As mentioned above, from the Malica dataset we consider two malware families, namely, Zbot (2,000 samples) and Winwebsec (4,000 samples). Using the proposed method for Zbot samples we attained an average AUC of 0.98, while for Winwebsec is 1.00. These results are summarized in the form of a bar graph in Fig. 2.

Fig. 3 give the frequencies of the most prevalent API calls in the Winwebsec malware samples. From the figure, we can see that the most significant features can be viewed as representative of potentially malicious acts, such as hidden encryption, code injection, extracting OS details, memory manipulation, and so on.

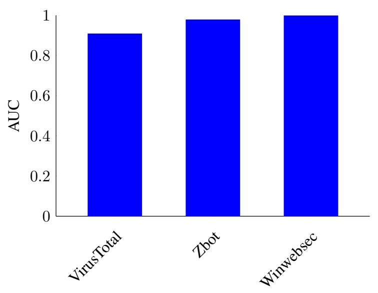


Fig. 2: Average AUC for each malware categories

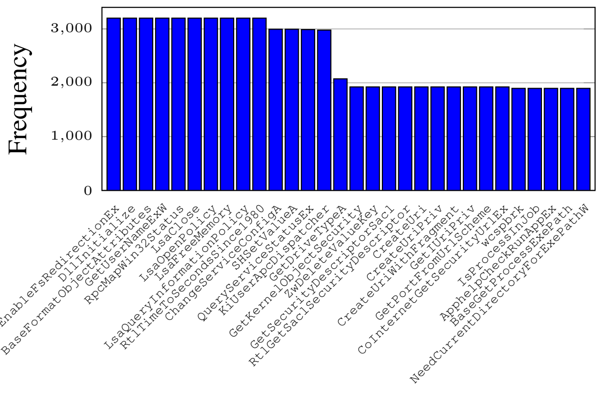


Fig. 3: Winwebsec most prevalent features

The experiments discussed in this section show that by applying naïve Bayes and a Levenshtein distance calculation to API call data, we can obtain results that are competitive with previous work in this field. As far as we are aware, our technique is more efficient—both in terms of computational cost and storage requirements—than any comparable research that relies on API call analysis.

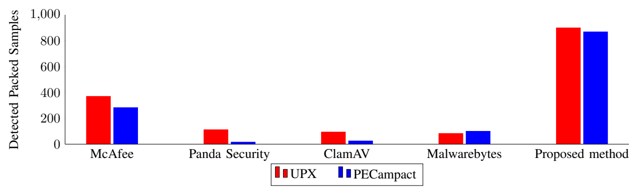
C. Obfuscated Samples

In this section we consider malware that has been obfuscated using two popular packers, namely, UPX and PECompact. We used these packers to compress the same VirusTotal and Malicia samples that were used in the previous experiments. Interestingly, we found that some of these samples were already packed using UPX or PECompact. For each such case, we added a sample from the appropriate family. Thus, we have the same number of samples as in our previous experiments, and each was packed by us. This give us a total of 2,000 packed VirusTotal malware samples, while for the Malicia dataset, we have 4,000 packed Zbot and 8,000 packed Winwebsec samples.

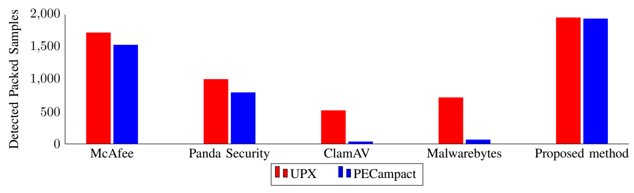
For VirusTotal packed samples, and both UPX and PECompact, our method outperform commercial Malware detectors by a large margin. For UPX, our method detected with 90% accuracy, while all other malware detectors achieved less than 40%. For PECompact, our method was equally impressive, detecting at 87% accuracy, while other malware detectors achieved less than 30%. These results are summarized in the form of a bar graph in Fig. 4 (a).

For Zbot packed samples, the proposed method also outperform other malware detectors. For UPX packed samples, our method achieved 97% accuracy, while other malware detectors achieved less than 87%. For PECompact, our method also achieved 97% accuracy, while all other malware detectors achieved less than 76%. These results are summarized in Fig. 4 (b).

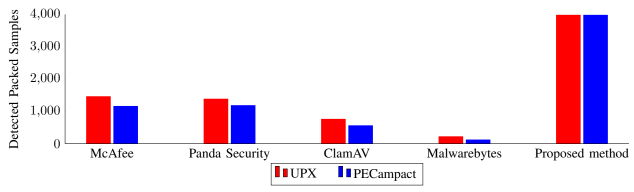
For Winwebsec packed samples, our proposed method once again outperforms other malware detectors. For UPX packed samples, our method achieved 99% accuracy, while all other malware detectors achieved less than 37%. For PECompact compressed samples, our method also achieved 99% detection accuracy, while other malware detectors achieved less than 30%. These results are summarized in Fig. 4 (c).



(a) VirusTotal



(b) Zbot



(c) Winwebsec

Fig. 4: Packed sample detection

V. Conclusion

Sophisticated malware is widespread and, due to various obfuscation techniques, such malware can be difficult to detect using standard techniques. In this paper, we proposed and analyzed a state-of-the-art dynamic heuristic method based on API calls. Our classifier uses naïve Bayes and a Levenshtein distance computation for increased robustness. We achieved good accuracy and our approach performed significantly better than commercial anti-malware products, particularly when confronted with packed malware samples.

Our work demonstrates that dynamically traced API features based on frequencies are feasible to collect and analyze. Furthermore, by our technique, we are able to determine the relative importance of the various features, since each feature has a defined probability. Consequently, our use of Levenshtein distance provides a simple and effective means of feature reduction. Overall, the proposed method achieves strong detection results with minimal computation and a low storage requirement.

For future work, we plan to apply this technique to the detection of additional challenging classes of malware, including polymorphic and metamorphic samples. We also plan to compare our dynamic analysis to an analogous static approach. We generally expect that dynamic features will perform better than static features, especially in the case of obfuscated code. However, static feature can typically be collected more efficiently, and hence the tradeoff between efficiency and effectiveness is worthy of exploration.

Acknowledgments

The authors thank VirusTotal for providing access to the malware samples used in this research.

References

[1] E. M. S. Alkhateeb, "Dynamic malware detection using API similarity," in *2017 IEEE International Conference on Computer and Information Technology*, CIT 2017, pp. 297–301, 2017.

[2] J. Aycock, *Computer Viruses and Malware*. Advances in Information Security, Springer, 2006.

[3] B. Jones, "PSafe Blog: Why you should only buy AV-test certified antivirus software." https://www.psafe.com/en/blog/why-you-should-only-buy-av-test-certified-antivirus-software/, 2017.

[4] N. Idika and A. P. Mathur, "A survey of malware detection techniques. Purdue University," 2007.

[5] W. Wong and M. Stamp, "Hunting for metamorphic engines," *Journal in Computer Virology*, vol. 2, no. 3, pp. 211–229, 2006.

[6] J.-M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology*, vol. 4, pp. 211–220, 2008.

[7] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. San Francisco, CA, USA: No Starch Press, 1st ed., 2012.

[8] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Physics Doklady*, vol. 10, no. 8, pp. 707–710, 1966.

[9] E. S. Ristad and P. N. Yianilos, "Learning string-edit distance," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, no. 5, pp. 522–532, 1998.

[10] I. Rish, "An empirical study of the naïve Bayes classifier," in *Workshop on Empirical Methods in Artificial Intelligence (IJCAI 2001)*, vol. 3, pp. 41–46, IBM, 2001.

[11] M. Stamp, *Introduction to Machine Learning with Applications in Information Security*. Boca Raton: Chapman and Hall/CRC, 2017.

[12] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006.

[13] W. Yan, Z. Zhang, and N. Ansari, "Revealing packed malware," *IEEE Security & Privacy*, vol. 6, no. 5, pp. 65–69, 2008.

[14] Y. Ki, E. Kim, and H. K. Kim, "A novel approach to detect malware based on API call sequence analysis," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 659101, 2015.

[15] M. A. Jerlin and K. Marimuthu, "A new malware detection system using machine learning techniques for API call sequences," *Journal of Applied Security Research*, vol. 13, no. 1, pp. 45–62, 2018.

[16] R. Tian, R. Islam, L. Batten, and S. Versteeg, "Differentiating malware from cleanware using behavioural analysis," in *5th International Conference on Malicious and Unwanted Software (MALWARE)*, pp. 23–30, IEEE, 2010.

[17] M. Shudrak, "GitHub: Drltrace is a library calls tracer for windows and linux applications.." https://github.com/mxmssh/drltrace/, 2018.