SVM Training Phase Reduction Using Dataset Feature Filtering for Malware Detection

O'Kane, P., Sezer, S., McLaughlin, K., & Gyu Im, E. (2013). SVM Training Phase Reduction Using Dataset Feature Filtering for Malware Detection. *IEEE Transactions on Information Forensics and Security*, *8*(3), 500-509. https://doi.org/10.1109/TIFS.2013.2242890

Published in:
IEEE Transactions on Information Forensics and Security

Document Version:
Peer reviewed version

SVM Training Phase Reduction using Dataset Feature Filtering

Philip Okane[1], Sakir Sezer[1], Kieran McLaughlin[1] & Eul Gyu Im[2]
[1]Centre for Secure Information Technologies, Queen's University Belfast, Northern Ireland, UK
[2]Div. of Computer Science & Engineering, Hanyang University, Seoul, 133-791, Korea
[1]{pokane17, sakir.sezer, kieran.mclaughlin}@ecit.qub.ac.uk, [2]imeg@hanyang.ac.kr

Abstract—**Obfuscation is a strategy employed by malware writers to camouflage the telltale signs of malware and thereby undermine anti-malware software and make malware analysis difficult for anti-malware researchers. This paper investigates the use of supervised learning machines to identify malware and investigates the problems of feature identification and feature reduction. We present several methods of filtering features in the temporal domain prior to applying the reduced feature set to the learning machines. The findings have identified several methods of feature reduction and are presented their viability as filters are assessed.**

Keywords-component; Obfuscation, Packers, Polymorphism, Metamorphism Malware, KNN, SVM

I. Introduction

Recent years have seen massive growth in malware, with signature detection and monitoring suspected code for known security vulnerabilities becoming ineffective and intractable.

In response, researchers need to adopt new detection approaches that outmanoeuvre the different attack vectors and obfuscation methods employed by the malware writers. Detection approaches that use the host environment's native op-codes at run-time will circumvent many of the malware writers' attempts to evade detection. One such approach, as proposed in this paper, is the analysis of op-code density features using supervised learning machines performed on features obtained from run-time traces. In further research we intend to expand the detection methods by investigating N-gram size, which will dramatically increase the number of features. With this anticipated explosion of features we have chosen to investigate methods to prune irrelevant features.

While Principle Component Analysis (PCA) is a popular method to reduce features in subspace, this paper aims to identify feature reduction in the temporal (original dataset) space.

Others have carried out research into indentifying malware based on statistical op-code analysis, such as Lakhotia et al [1] that presented a static detection of obfuscated calls relating to *push*, *pop* and *ret* op-codes mapped to stack operations. The approach presented in this paper performs dynamic analysis and evaluates the full spectrum of op-codes, which builds upon the work carried out by Bilar [2].

For large datasets, or costly (computation) distance function, the training process associated with learning machines can become intractable. Thus, the feature explosion that occurs with N-grams for large values of N needs to be addressed. This paper investigates several approaches to filtering out irrelevant features.

The remainder of this paper is laid out as follows: Section II gives an overview of the approach. Section III describes the dataset. Section IV details the test platform and monitoring tools. Section V details the Support Vector Machine configuration and the results obtained. These results are used as a reference to gauge the successfulness of the filtering approaches. Section VI gives a brief overview of competing malware detection strategy. Section VII introduces and presents empirical data that characterise the different feature filtering approaches. The penultimate section, VIII, summarises the results and key characteristics recorded during these experiments. Finally, Section IX concludes by comparing the results with other research and details future work that will be carried out as part of this research.

II. System Overview

The motivation for this research is to reduce the computational overhead required when N-gram analysis is performed on low-level fine grain data. Therefore, developing a lightweight filter that will reduce the number of features to be processed will in turn reduce the computational overhead; thus making the training phase of the SVM approach a viable solution for N-gram analysis where large feature sets are generated. Fig. 1 illustrates an overview of the approach taken in this paper. The programs under investigation are run in a test environment with a debug tool monitoring the runtime op-codes. After completion, the data is parsed into op-code histograms and after some conditioning the dataset is passed to the SVM to perform feature selection of the optimum features that can be used to detect malware. In parallel, the dataset is processed with various filtering algorithms in an attempt to identify the op-codes selected by the SVM. The objective is that a lightweight filter can be identified and used to pre-process the dataset before feeding it to the SVM.

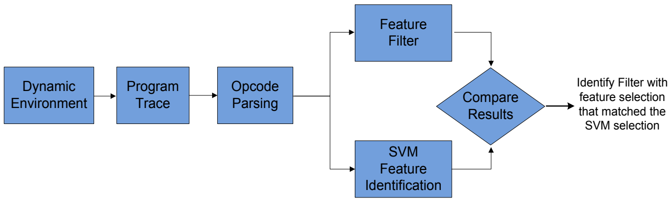
Dynamic Environment
Program Trace
Opcode Parsing
Feature Filter
SVM Feature Identification
Compare Results
Identify Filter with feature selection that matched the SVM selection

Fig. 1 System Overview

III. Test Platform

The main challenge with dynamic analysis is to ensure that the malicious code path is executed during investigation. Three dynamic approaches exist: 1) Native (debugger), 2) Emulation and 3) Virtualization. Each has to address malware evasion techniques that may attempt to fool the dynamic analysis into completing its analysis without running the malicious code [5].

While native environments present the malware with a real platform on which to run, this presents issues with control and ‘clear up’ of the malware infection. The program under investigation needs to be monitored during execution. There are several debugger tools available to monitor and intercept programs - IDA Pro, Ollydbg and WinDb32, which are popular choices for malware analysis.

A virtualization approach is chosen as it provides isolation by decoupling the virtual machine (malware environment) and the OS. The isolation provided by a hypervisor in a virtualised system prevents the malware from infecting the host OS or other applications that are running on adjacent virtual machine on the same physical machine. Prior to performing the analysis, the virtualized environment files (virtual images) are backed-up (snapshot). After the analysis is complete, the infected files are discarded and replaced with the clean snapshot.

The test platform consists of a QEMU-KVM hypervisor with Windows XP (SP3) installed. Ollydbg is chosen because it is open source and supports the StrongOD plug-in to prevent the malware from detecting that it is being monitored.

IV. Dataset

Classification tasks involve separating data into training and test data. Each training-set instance is assigned a target value/label i.e. benign or malicious. The goal of the SVM is to construct a model that predicts the target values of the test data. Table 1, lists the dataset with the benign files being Windows XP executables. The dataset is constructed from runtime traces that are parsed to create histograms of op-code density. The malware samples are restricted to programs that indicated that they are packed or use encryption, which is determined by using the Ollydbg plug-in StrongOD.

Table 1 Dataset

	Dataset	Training	Validation
Benign	260	230	30
Malicious	350	310	40
Total	610	540	70

While there are 344 different Intel op-codes, only 149 different op-codes are recorded during the captured datasets for all programs traced during this experiment. The dataset is normalised by calculating the percentage density of op-codes rather than the absolute op-code count to remove time variance introduced by different run lengths of the various programs. The dataset is sorted into most commonly occurring op-codes as illustrated in Fig. 2.

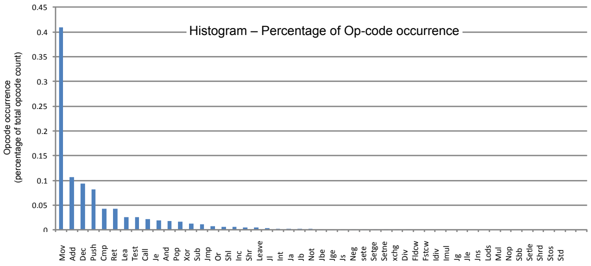

Fig. 2 Histogram: Op-code Percentage

The dataset is marshalled into matrix format so that the data could be manipulated easily by Matlab, as detailed in Table 2. Note – the rank value versus structure size shows that there is a linear dependency between rows i.e. redundant information exists within the dataset.

Table 2 Number of files in datasets

Variable name	Size (r/c)	Rank	Comment
X	623/149	106	Total data
Training	561/149	104	Model training
Test	62/149	61	Validation data
Label	1/149	N/A	Training label

An initial assessment of the data shows two key properties a) The distribution of the various op-codes does not show any consistent distribution shapes; rather op-code distribution changes greatly as illustrated by the difference between *mov* and *ret* op-codes, described later in VII: ‘Area of Intersect’. Therefore, no one data shape could be assumed and hence a non-parametric method should be used. b) The data values are a percentage of the op-codes within a particular program. For example, 0 means that op-code does not occur within that program or 0.25 means that 25% of the program comprises of that op-code. To improve the performance of the SVM the data is linearly scaled (0, +1).

V. Support vector machine

Support Vector Machine (SVM) is a technique used for data classification and was introduced by Boser et al in 1992 [4] and is categorised as a kernel method. The kernel method algorithm depends on dot-products function, which can be replaced by other kernel functions that map the data into a higher dimensional feature space. This has two advantages: Firstly, the ability to generate a non-linear decision plane and secondly, allows the user to apply a classification to data that do not have an intuitive dimensional vector space i.e. SVM training when the data has a non-regular or unknown distribution [16]. The dataset consists of 149 different op-codes, each having their own unique distribution characteristics and therefore a SVM is an appropriate choice. As mentioned earlier, the data is linearly scaled to improve the performance of the SVM. The main advantages of scaling are a) it avoids attributes with greater numeric ranges dominating those with smaller numeric ranges and b) it avoids numerical difficulties during the calculation as kernel values usually depend on the inner products of feature vectors, e.g. in the case of the linear kernel and the polynomial kernel, large attribute values might cause numerical problems [17].

The RBF (Radial Basis Function) kernel is used as it is considered a reasonable first choice in that it provides a non-linear mapping of samples into a higher dimensional space. This caters for instances where the relationship between the class label and attributes is non-linear.

SVM is used to create a reference datum to validate the filter experiments that are presented in the subsequence sections. The SVM is configured to traverse through the dataset searching for op-codes that have a positive impact on the classification of benign and malicious software. The search starts with six op-codes scanning across the complete data sequence for all unique permutations for that number of op-codes. The search is repeated for five and then four op-code sequences. An average of these results is sorted by most occurrences as illustrated in Fig 3, which show the most important op-codes as chosen by the SVM. Only unique op-codes are selected for each SVM classification test and no duplicates of repeated op-code patterns are processed. Key points to note are:

1) The 6 op-codes *ja, adc, sub, inc, add* and *rep*, each having an importance rating of more than 20% of the peak detection rate, are selected as the most important indicators for classifying benign and malicious software.

2) *mov* has a negative impact on the classification and identification of software. i.e. when *mov* is part of the analysis data the output/classification is always incorrect. The *mov* has a high density (30% [2] and 40% in the presented dataset) in both benign and malicious software.

3) Polymorphic and encryption based malware commonly use the *xor* instruction as the transfer/encryption function. Despite the fact that several polymorphic and encryption based malware samples are used in both the training and validation dataset *xor* is not highlighted as an indicator of malware.

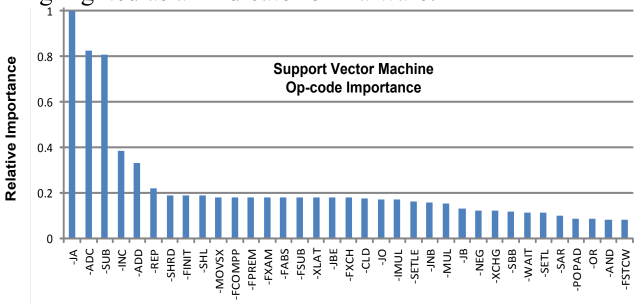

Fig. 3 SVM Op-code Sensitivity

VI. RELATED WORK

There has been extensive research into the detection of malicious code using both static and dynamic analysis. Malware research can be categorised not only in terms of static and dynamic analysis but also in how the information is processed after it is captured. Popular research methods include: Control Flow Graphs (CFG) for both course and fine grain analysis, State machines to model system behaviour, mapping stack operations and mechanisms to detect malicious behaviour for known vulnerabilities.

CFG analysis has played a key role in the research. Bilar [6] compared the statically generated CFG of benign and malicious code. Their findings showed a difference in the basic block count for benign and malicious code. Bilar concluded that malicious code has a lower basic block count, implying a simpler structure: Less interaction, fewer branches and less functionality.

Christodorescu et al [7] presented a semantic aware technique that used CFG model checking to identify components of malware using previously specified malicious code templates. These templates are constructed using a 3-tuple signature that consists of instructions, variables and symbolic constants. They addressed some issues relating to obfuscated code such as 'dead code' insertion, register reassignment and code sequencing.

Vinod et al [8] proposed a method that constructed CFG nodes from blocks of de-obfuscated (Normalised) code of a known malware program. These blocks of instructions are compared for similarities to identify variants of the malware. Zhang et al [9] proposed a similar method that pattern-matched code fragments to determine if two code fragments are similar enough to exhibit functional equivalency.

Bonfante et al [10] extended this research by using a reduced CFG to reduce the effects of code reordering as used in obfuscation. Their experiments showed that CFG size affected the false positive error rate i.e. decreasing the CFG size increased the false positive error rates.

Vulnerabilities have been the Achilles heel of software security, which malware writers have continuously targeted, with stack exploits being a major issue. Lakhotia et al [1] presented a method to detect obfuscated calls relating to *push*, *pop* and *ret* that are mapped to stack operations. An abstract stack model is constructed from results obtained from program executions. Their work does not address situations where *push* and *pop* instructions are decomposed into multiple instructions, such as directly manipulating the stack pointer using *mov* commands.

The lack of user input validation has been another security weakness that has lead to many malware attacks. Newsome et al [11] proposed a dynamic taint analysis for automatically detecting malware attacks. Their approach is fine-grained analysis that could detect overwrite attacks that utilised vulnerability and exploits.

Monitoring program behaviour to determine malicious activity has been another avenue of research. Ellis et al [12] proposed a dynamic detection system that used behavioural signatures of worm operations to identify worms. One such behaviour is that a worm acts like a server when infecting a host and after the infection is complete, it changes its behaviour from a server to client as it attempts to infect adjacent hosts.

OS calls analysis has also provided an avenue of research. Okazaki et al [13] proposed an anomaly-based approach to analyse program behaviour, based on profiling OS calls on a Unix platform, and checking whether the system is being used in a different manner. The model is constructed by ranking the OS calls based on their popularity during normal operation. The model is used to compare the behaviour of running programs using a distance algorithm that eliminated system variance. Hofmeyr et al [14] also used anomaly detection based upon the sequence of OS calls. A normal profile obtained from the sequence of OS calls is compared against suspected binaries using hamming distance calculated on the sequences of OS calls.

Sekar et al [15] used Finite State Automata (FSA) to represent OS call sequences. The binary is executed multiple times and recorded the OS calls to create the FSA models. Anomaly detection is achieved by comparing the FSAs with the run-time sequence of OS calls. Ruschitzka et al [16] presented an approach based on the sequence of OS calls that is similar to [15].

Malware has often attempted to hide its presence by injecting itself into other file and performing entry point obfuscation. Rabek et al [18] proposed an anomaly based technique that used both static and dynamic analysis to detect injected, dynamically generated and obfuscated code. During static analysis the location of each OS call is identified within the program and then when the program runs, each OS call is verified against the original location.

N-grams are based on a signature approach that relies on small sequences of strings or byte codes that are used to detect malware. Santos et al [19] demonstrated that n-gram signatures could be used to detect unknown malware. The experiment extracted code and texts fragments from a large database of programs executions to form signatures that are classified using machine learning methods

Sekar et al [20] implemented an n-gram approach and compared it to a FSA approach. They evaluated the two approaches on httpd, ftpd, and nsfd protocols. They found that the FSA method has a lower false-positive rate when compared to the n-gram approach. Li et al [21] describe N-gram analysis, at byte level, to compose models derived from learning the file types the system intends to handle. Li et al found that applying an N-gram analysis at byte level (N=1) on PDF files with embedded malware proved an effective technique of detecting malicious PDF files.

However, Li et al only detected malware embedded at the beginning or end of a file; therefore any malware embedded in the middle of the file will go undetected. Li et al suggested that further investigation needed to be carried out on the effectiveness of N=2, N=3 etc.

In this vein, we have chosen to focus our research on the identification of malware using N-grams obtained from run-time program traces. We started with N=1 and have demonstrated that malware can be identified with a reduced set of features. Initial investigation has shown that for N=1, 149 features are produced in the raw dataset. Increasing N=2 produced 8092 features (no filtering). Therefore, before continuing the research by increasing N, it is prudent to establishing a basis to filter the dataset to prevent feature explosion. To this end, this research focuses on finding a filter to remove redundant features.

The key weakness of static analysis is that the code analysed may not be the code that actually runs, which is particularly true for obfuscated programs that employ polymorphic or metamorphic techniques. Due to the high level of obfuscation employed by malware writers, dynamic analysis is used to find indicators of malware.

VII. Op-Code Pre-filtering

N-gram analysis presents a dimensionality problem in terms of the number of raw features produced and if left unfiltered would result in a SVM training phase with a high computation cost. To reduce this effort and narrow the area of search, this research aims to identify filters that can select the optimum features prior to feeding them to a SVM. The hypothesis is: Malware that employs evasion techniques will exhibit telltale signs in terms of run-time op-codes; such as a higher density of instructions that are commonly used in polymorphic engines within malware. Therefore filtering out irrelevant op-codes and allowing the SVM to focus on a subset will result in a fast training phase.

A. Hypothesis test

Firstly, considering the null hypothesis: Benign and malicious software produce the same op-code distributions; as

$$Ho \rightarrow \mu = \mu o$$

$$Ha \rightarrow \mu \neq \mu o \rightarrow reject$$

Ho – The sample data belong/fits into the distribution of the original dataset.

Ha – If the sample data does not fit into the original dataset, the inference is that the two datasets are different.

As both sets of data are large and the mean and standard deviation can be calculated, the critical Z formula is used as shown;

$$Z = \frac{(\overline{x} - \mu o)}{(\sigma / \sqrt{n})}$$

Where – Z = test statistic;

$\overline{x}$ = test data mean;

μ_0= mean of parent group;

n = Sample size;

σ = standard deviation of population;

Table 2 lists the calculated Z values. A large value of Z indicates that those distributions are likely to belong to different groups and are therefore more likely to indicate a difference between benign and malicious software. On the other hand, a small value of Z would indicate that the two distributions belong to the same group and are therefore unlikely to make a good indicator of malware. The Z values do not present any meaningful correlation to those op-codes selected by the SVM and therefore would not make an appropriate filter, but are listed here for completeness.

B. Area of intersect

Secondly, consider the simplistic characteristics of benign and malicious op-codes with a normal distribution as shown in Fig. 4. The plots are grouped into density curves for benign and malicious software of a single op-code. The horizontal axis relates to the percentage of a given program that is made up of a particular op-code and the vertical axis indicates the number of programs with that percentage of op-code. The key feature to note is the overlapping area of the two density curves. The greater the difference between the mean of the curves and narrower the standard deviation reduces the overlapping area and therefore reduces the interference and corresponding misclassification of the benign and malicious software.

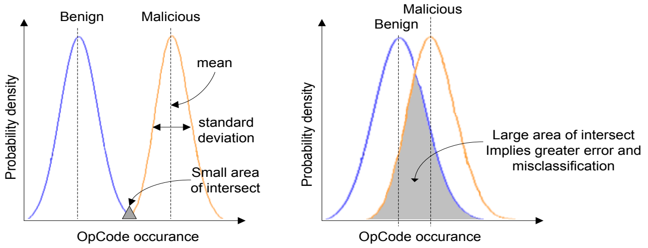

Fig. 4 Ideal characteristics

This implies that a simple analysis of low order statistics, such as calculating the product of the mean and the inverse of the standard deviation to determine the overlapping area would yield the best indicators (op-codes) of benign and malicious software. Hence, calculating the overlapping area for the density curves should provide a numerical value to those indicators. Using-

$$C = \frac{A + B - (|A - B|)}{2}$$

Where – A = area of benign density curve.
B = area of malware density curve.
C = area of intersect of the two density curves.

The above equation produced the results shown in Fig. 5. These results need to be placed in a context that provides meaning in term of relative importance. Those op-codes chosen by the SVM as the optimal indicators are highlighted.

It can be seen that those op-codes chosen by the SVM do correlate in part with those op-codes sorted by the area least intersect. While it is not a direct match, setting a filter criteria that selects op-codes with a area of insect of less than 50% does not eliminate any of the optimal op-codes and removes 75% of those op-codes that provide no values in the classification of benign and malicious software. However, a important op-code (*adc*) has been removed by the 'area of insect filter' and therefore cannot be considered a useful tool for removing irrelevant op-codes.

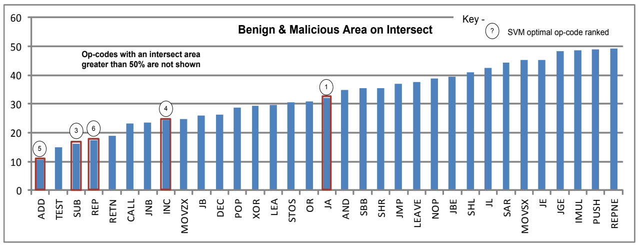

Fig. 5 Benign & Malicious area of intersect

The SVM selected: *ja*, *adc*, *sub, inc, rep and add* as indicators of benign and malicious software. The second most important op-code *adc* is filtered out by the 'area of intersect' filter which contradicts the hypothesis that op-codes with the least area of intersect make the best indicator of benign and malicious software. This is clearly not the case. Two further points need to be considered. Firstly, the overall density of a particular op-code needs to be considered in the context of their area of intersect and its population as it needs to be significantly important to be considered as an indicator of benign and malicious software. Taking *ja* and *rep* op-codes (SVM selected range) as reference points, it can see from the data presented in Table 2 that the other op-codes relating to population and area of intersect fall within the characteristics of *ja* and *rep*. Therefore the area of intersect does not tell the full story as many other op-codes such as *ret, call,* etc have lower area of intersect than *ja* and a population that lays between both *rep* and *ja*. In addition the 'area intersect' filter removes the *adc* op-code. Low dimensional analysis does not consider covariance i.e. the relationship between the distributions of one op-code with that of another op-code.

Table 2 Op-code Statistics

Op-code	% occurrence	% area of intersect	Z-Value
add	6.10	10.76	0.35378
test	3.16	15.07	0.52379
sub	2.67	16.25	0.463953
rep	6.31	17.53	0.455446
ret	1.85	18.82	0.504489
call	1.91	23.30	0.536635
jnb	1.41	23.50	0.419838
inc	5.24	24.59	0.543303
movzx	1.66	24.62	0.9687
jb	1.66	25.83	0.499322
dec	1.67	26.28	0.535424
lea	2.56	29.76	0.896827
stos	0.45	30.44	0.813775
pop	3.51	30.72	0.649154
xor	2.29	32.29	0.821013
ja	1.17	32.20	0.641516
and	1.01	34.72	0.962486
sbb	0.09	35.32	0.784135
shr	0.67	35.44	0.776227
jmp	1.83	36.98	0.677423
leave	0.45	37.46	0.395372
nop	0.34	40.72	0.848149
jbe	0.27	39.57	0.57276
shl	0.55	40.92	0.600475
jl	0.38	42.52	0.896412
sar	0.25	44.27	0.64066
movsx	0.23	45.11	0.647835
je	4.52	45.18	0.851666
jge	0.27	48.26	0.708946
imul	0.30	48.71	0.93168
push	7.99	48.98	0.54069
repne	0.32	49.28	0.732861

As shown in Fig. 5, it is not always the case that op-codes with a low area of intersect produce the best indicators of benign and malicious software. This requires a closer inspection of the op-code distribution curve to understand the characteristics that make the best indicators chosen by the SVM over the other op-codes that have similar area of intersect and population.

C. Linear Programming

The previous explanation (Fig. 4) considered a normal probability distribution. Therefore, further investigation is required to understand how the area under each curve is interpreted when a decision plane is applied. Linear Programming (LP) [23] is a technique that is applied to optimise a linear function when subject to linear equality and inequality constraints. LP can be applied to the classification of benign and malicious software. The components of LP are-

Constraints - The data is in the form of a probability density curve. The horizontal axis represents the makeup of a program i.e. the op-code percentage that makes up a program and the vertical axis, representing the number of programs that have that percentage of op-codes. The probability density is based on a percentage of op-code counts obtained from traces during the execution of a program. The minimum value is 0 and the maximum is the percentage of the most occurring op-code within the captured dataset (*mov*). Thus the maximum value is 0.4 (40%).

Decision Variable - this is the value found during the search for the maximum or minimum point. It is the percentage of a particular op-code that yields the greatest area of benign and malicious density that lies either side of the decision plane.

Objective function - is the numerical expression used to define the goal of the task. The mathematical input to the LP is the cumulative probability as the decision variable is incremented across the range (illustrated in Fig. 6). Therefore the maximum classification would be achieved when the two density curves do not intersect and their entire area lies on their respective side of the decision plane, as-

$$A = 1; B = 1; when A \not\subset B$$

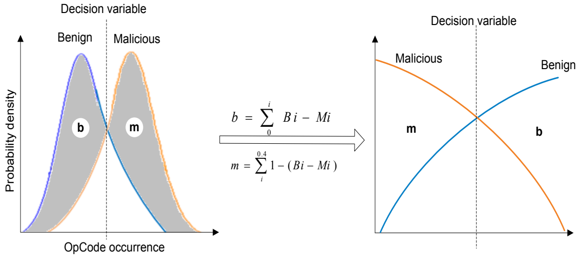

Fig. 6 Linear Programming Optimisation

Listed below is the Matlab script used to calculate the optimum decision point.

```
stepSize = 0.001;
endofData = 0.4;
range = endofData/stepSize;
result1 = [];
result2 = [];

for i = 1:1: range
    result1 = [result1; ((sum(f(i:range))/sum(f)))];
    result2 = [result2; (1- (sum(f2(i:range))/sum(f2)))];
end;

result3 = abs(result1 - result2);

% minimum point that represents the maximum density either side of the
decision plane.
min(result3)
```

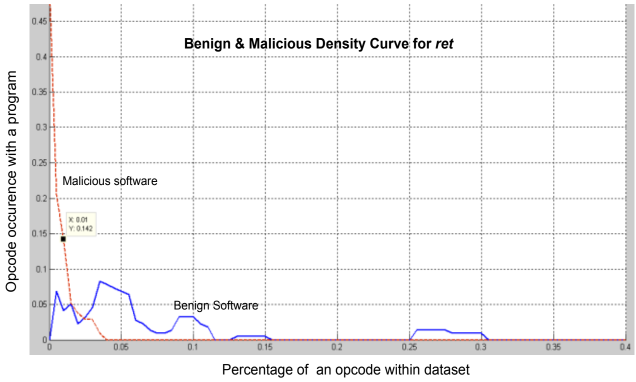

Fig. 7 *ret* op-code probability density curve

Fig. 7 shows the density distribution of the *ret* op-code for both benign and malicious software. It can be seen that a software program with a *ret* density of 0.001 is much more likely to be malicious that benign software. To distinguish between benign and malicious software and to determine the likelihood of a correct classification, the optimum value has to be obtained and the respective areas that lie either side of the optimum decision plane have to be assessed. The cumulative density curves are calculated, as the decision variable is incremented across the density curves, as shown in Fig. 8.

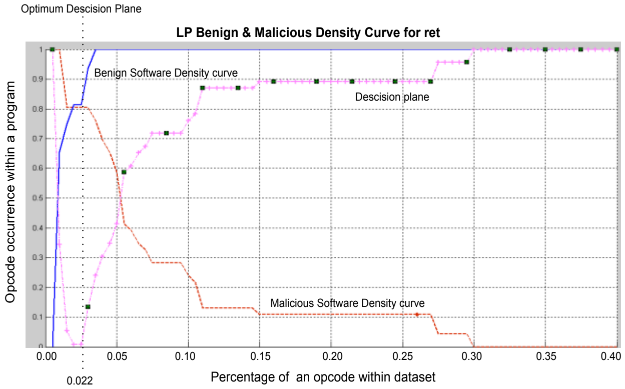

Fig. 8 Cumulative Probability Density with Cost function

In Fig. 8, a cost function calculates the area under each curve and identifies the optimum decision point (0.022). This point marks the optimum value that should be used to maximise the correct classification of benign and malicious software. Fig. 8 shows a cumulative probability density of 84% lying either side of a decision plane at 0.022. Therefore, a software program with a value greater than 0.022 is highly likely (84%) to be benign software; and programs with values less than 0.022 are highly likely to be malicious.

Considering ‘false friends’: Those op-codes that appear at first glance to have the potential to detect malicious software but fail to do so. Fig. 9 shows the cumulative probability density for the *je* op-code with the decision plane optimised at 0.045. This example is purposely chosen as a worst op-code to demonstrate that the area of intersect and population are not the only contributing factors. The optimum value yields a very poor predictor of malware as the area either side of the decision plane is 57% making is slightly better than guessing. Here we have seen that the *je* op-code had initially promising characteristics of a low area of intersect and a high population, yet the LP analysis of the *je* op-code showed that it only has a density of 57% either side of the decision plane. The process is repeated for all the op-codes within this discussion and the results are listed in Table 3.

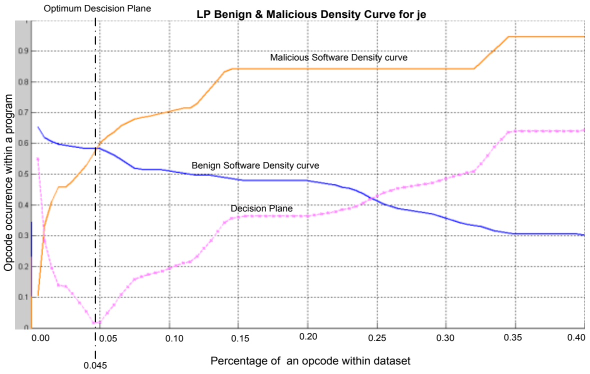

Fig. 9 Cumulative Probability Density with Cost function

Table 3 LP Decision Plane

Op-code	Occurrence %	area of intersect %	Optimised Decision plane %
pop	2.60	28.70	70
push	8.87	48.98	64
je	6.53	45.18	57
add	4.35	10.75	56
test	3.16	15.07	56
sub	2.67	16.25	60
rep	6.31	17.53	78
ret	4.54	18.82	84
call	4.60	23.30	78
jnb	1.41	23.50	54
inc	5.24	24.59	66
mov	40.76	24.62/(49.7)	72 (unstable)
jb	1.66	25.83	54
dec	1.67	26.28	54
lea	1.74	29.29	72
stos	0.45	30.44	56
pop	3.51	30.72	56
xor	2.29	29.23	58
ja	1.17	32.20	57

While there are no clear boundaries of what makes a good or bad indicator of malicious software, the LP analysis has placed a numerical value upon those op-codes discussed in this section (Table 3). Clearly op-codes with 50% of their area either side of the decision plane make very poor indicators of malware, whereas op-codes with a high area either side of the decision plane make better indicators. Therefore, using this criteria LP has demonstrated that *cmp, push, je, add* and *pop* are less effective that *ret* and *lea.* However, two op-codes remain of interest, *call* and *mov*. The *call* op-code has a high area either side of decision place (78%) which merits further investigation. While the *mov* op-code has an area of 72% either side of the decision plane, its behaviour when analysed by Pearson’s correlation, SVM and LP merits further examination as to why *mov* has a negative impact on malware identification.

Fig. 10 shows the density distribution of the *mov* op-code for both benign and malicious software. It can be seen that the distributions are substantially interleaved as compared to the *ret* op-code as shown in Fig. 7. In addition, during analysis, it was noted that small changes in the calculation step size produced dramatic changes in the results. In an attempt to stabilise the results, the data is processed using a KSdensity function, which is a Kernel density estimation function that performs data smoothing by projecting data population based on a normal kernel profile [25] and is shown in Fig. 11.

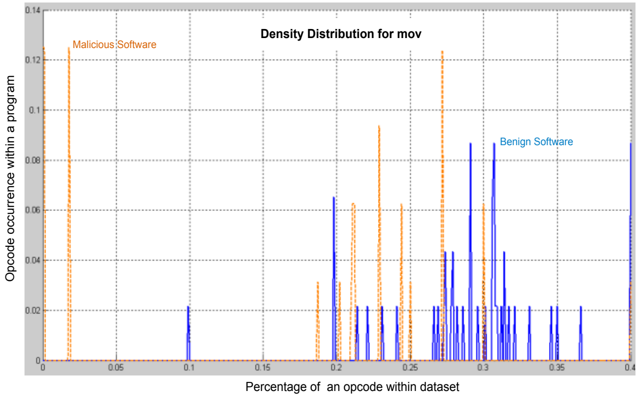

Fig. 10 *mov* Density Distribution

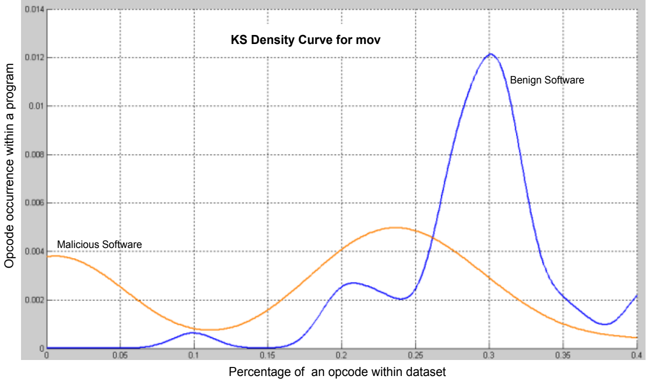

Fig. 11 KS Density Curve for *mov*

Calculating the area of intersect for the KS density curve fit on the mov op-code resulted in an intersect area increasing from 15.55% to 49.70%. Clearly, the granularity of the analysis of the *mov* op-code greatly affects the results and implies an unstable indicator.

Fig. 12 shows both the raw density curve and its KS density curve fit for the *call* op-code. The raw data shows interleaving and some clustering of the two density curves (benign and malicious) as highlighted by the KS density curves fit. While this is an improvement over the *mov* op-code, comparing it to the *ret* op-code as shown in Fig. 12, it can be seen that the *ret* op-code presents better separation between the two density curves (benign and malicious). Calculating the area of intersect for KS density fit on the *call* op-code increases the area slightly from 24.81% to 25.6%, implying stability as opposed to the unstable results seen with *mov*.

Performing the same comparison (raw data versus KS density) for *ret* and *lea* gave 13.69% dropping to 6.81% and 24.81% increasing to 25.63 respectively.

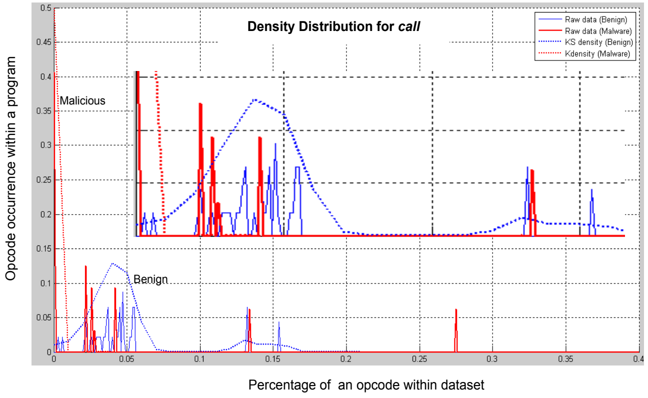

Fig. 12 Distribution of *call* op-code

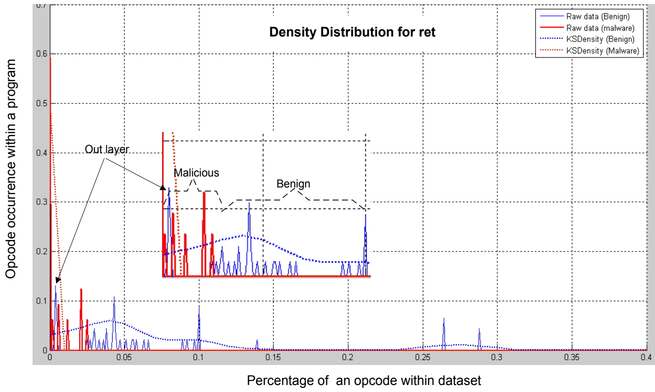

Fig. 13 Distribution of *ret* op-code

While the statistical analysis presented in this section aids the understanding and identification of those op-codes that make the best indications, their comparison to SVM ability is somewhat compromised as the SVM is more than the sum of the individual op-codes; it is the covariance matrix i.e. comparison between different variables. The remapping performed by the SVM on the input dataset into feature space improves the data separation and thus improves both the accuracy and robustness of the classifier. In addition, the SVM has the cumulative effect of considering a set of op-codes rather than individual op-codes. But, the relative importance of individual op-codes is valid.

D. Subspace

An alternative approach to determine the importance of the individual op-codes, thereby ranking their usefulness as classification features, is to investigate the eigenvalues and eigenvectors in subspace. Principal Component Analysis (PCA) is a transformation of the covariance matrix and it is defined as [24]:

$$C_{ij} = \frac{1}{n-1} \sum_{m=1}^{n} (X_{im} - \bar{X}_i)(X_{jm} - \bar{X}_j)$$

Where – C =.Covariance matrix of PCA transformation;
X=.dataset value;
$\overline{X}$ = .dataset mean;
n and *m* = data length;

This is a technique used to compress data by mapping the data into a subspace while retaining most of the information/variation in the data. It reduces the dimensionality by mapping the data into a subspace and finding a new set of variables (fewer variables) that represent the original data. These new variables are called principal components (PCs) and are uncorrelated and are ordered by their contribution (usefulness/eigenvalue) to the total information that each contain.

Firstly, to determine the number of PCs that correlate to greater than 95% of the data variance, PCA is used. The results show that eight values accounted for 99.5% of the variance; therefore the eight largest (most significant) eigenvalues are used to locate the most significant eigenvectors (meaningfully data).

As PCA is an algorithm that operates on variance of data i.e. a covariance matrix of the training dataset, which is calculated in Matlab as follows:

$$C = cov(trainingData)$$
$$[V, \lambda] = eig(C)$$
$$d = diag(\lambda)$$

Calculating the significant values by multiplying the significant eigenvector Column by the respective eigenvalues and then summing each row

$$R_k = \sum_{k=1}^{8} V.d_k$$

Where – R = Sum of the matrix variance;
C = Covariance;
V = eigenvector;
λ= EigenValue matrix;
d= EigenValue scalar;

The results are illustrated in Table 3, note $\lambda_{3\text{->}7}$ values are not shown due to layout considerations.

Table 3 Ranking by Eigenvector

Op-code	$\lambda_1 x_{[1..149]}$ (10^{-3})	$\lambda_2 x_{[1..149]}$ (10^{-3})	$\lambda_{..} x_{[1..149]}$ (10^{-3})	$\lambda_8 x_{[1..149]}$ (10^{-3})	$\sum_{k=1}^{8}$ (10^{-3})
rep	4.79808	5.7494	..	0.009	11.81529
mov	6.78336	6.78336	..	0.03066	9.05347

add	2.8656	2.8656	..	0.03066	8.06337
push	2.23008	2.08845	..	0.0693	6.69625
adc	1.46304	1.80965	..	0.0051	3.91844
sub	0.74688	1.9006	..	0.01953	3.60979
inc	0.20928	0.20928	..	0.03669	3.05887
je	0.81024	0.6698	..	0.0231	2.80649
cmp	1.70304	0.1207	..	0.06282	2.68178
pop	0.96672	0.63155	..	0.03018	2.44806
test	0.79872	0.8789	..	0.05628	2.25302
ja	0.41568	0.95965	..	0.1449	1.97699
jnb	0.31392	0.7616	..	0.11751	1.66773
jb	0.25824	0.8279	..	0.08028	1.54664
call	0.6096	0.55845	..	0.02712	1.54664

Table 3 has been sorted based on the sum (largest first) of the eigenvectors and are displayed in Fig. 3 and is overlayed with those op-codes chosen by SVM as the optimal features for detecting malware. The 'eigenvector' filter has not only correctly chosen those op-codes selected by the SVM but has grouped them in to the most significant range (highest eigenvector/eigenvalues).

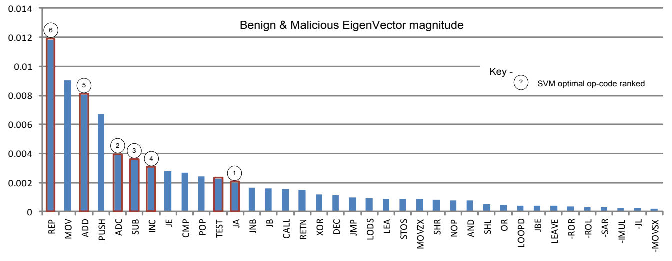

Fig. 4 Eigenvector magnitude

Given that the six SVM chosen op-codes have been grouped into the top twelve op-codes i.e. top 8% thereby removing the 92% irrelevant op-codes makes this an effective filtering mechanism to reduce features prior to the SVM training phase.

VIII. Discussion

Malware has a long history of evolutionary development as the war between the anti-malware researchers and the malware writers has progressed. This study presents an argument for the analysis of run-time op-code trace to detect malware.

Firstly, Op-code traces are captured for both benign and malicious software in the form of dynamic traces of each program. This data is marshalled and used as both the training and validation datasets.

Secondly, a SVM performs analysis on all the available op-codes and identifies the best indicators of malware, which is used to create a reference datum for the filters.

Finally, several filter criteria are tested against the reference datum generated by the SVM.

While this research is still ongoing, evidence exists to show that a subset of op-codes can be used to detect malware; and applying a filter to the features can reduce the SVM training phase. This has established an efficient approach to investigate the variables of N-gram N=2, N=3 etc. It is proposed that the findings provide a basis for further research in order to identify key op-codes or groups of op-codes (N-grams) that can be used to detect malware, negating evading techniques employed by the malware writers.

Bilar [6] demonstrated a difference in structure between benign and malicious software, which has been borne out by the finding produced by the SVM.

Lakhotia et al [1] presented a method to statically detect obfuscated calls relating to *push*, *pop* and *ret* op-codes that are mapped to stack operations. However, their approach did not model situations where the *push* and *pop* instructions are decomposed into multiple instructions, such as directly manipulating the stack pointer using *mov* commands. The method proposed in this paper investigates the full spectrum of op-codes to identify key op-code characteristics that will yield valuable indicators for malware detection.

While Lakhotia et al [1] and Bilar [2] methods use static analysis, the approach presented in this paper uses dynamic analysis and therefore evaluate actual execution paths as opposed to evaluating all possible paths through the program that is normally done when static analysis is used.

The results obtained from this experiment shown that high-density op-codes such as *mov* are not good indicators of malware. However, less frequent op-codes such as *ja, adc, sub, inc* and *add* make better indicators of malware, which confirms Bilar [2] claim that the 14 most occurring op-codes do not provide a good indication of malware with the exception of the *add* op-code, as listed in Table 4 (shadded area indicate op-codes selected by the SVM). While there is no numerical relationship between Bilar and the SVM results, LP and eigenvector analysis are introduced to identify a pre-classification filter and append a meaningful numerical value in terms of their ability to indentify malware. While the 'area of intersect' filter identifies five of the op-codes chosen by the SVM it missed an important op-code *adc*. The 'eigenvector' filter correctly indentifies all the op-codes chosen by the SVM. A point to note is that eigenvector filter ranked the *mov* op-code as the second best indicator of malware which contradicts the other analysis. Therefore the 'eigenvector' filter adequately removes irrelevant op-codes but does not guarantee that all the selected op-codes make the best indicators of malware.

Table 4 Op-code Importance

Op-code	SVM	Area	Subspace	Bilar Population
mov	×	×	2	25%
push	×	32	4	19%
call	×	6	15	9%
pop	×	×	×	6%
cmp	×	×	×	5%
jz	×	×	×	4%
lea	×	×	×	4%
test	×	2	×	3%
jmp	×	×	×	3%
add	5	1	3	3%
jnz	×	×	×	3%

ret	×	5	16	2%
xor	×	13	17	2%
and	×	×	×	1%
rep	6	4	1	×
ja	1	17	12	×
inc	4	8	7	×
adc	2	×	5	×
sub	3	3	6	×

Note - × indicates that the op-code was not ranked high enough to be considered

Further investigation is required to determine the interrelationship between op-codes, N-gram size and their ability to act as good indicators of malicious software. The investigation will need to identify and exclude misleading data as is exhibited by the *mov* op-code, which has an immense negative impact on the correct classification of benign and malicious software.

IX. Conclusion

This paper, proposes the use of SVM as a means of identifying malware. It shows that malware, that is packet/encrypted, can be detected using SVMs and by using the op-codes chosen by the SVM as a benchmark, identified a pre-filter stage using eigenvectors that can reduce the feature set and therefore reduce the training effort. The results presented in this paper exposed three key points.

Firstly, the identification of a high population op-code: *mov* that is not only is a poor indicator of benign/malicious software, but inhibits the ability to correctly classify software when used with other op-codes such as *ja, adc, sub, inc, add* and *rep*.

Secondly, a subset of op-codes can be used to detect malware. However, the SVM analysis demonstrates that *ja*, *adc* and *sub* are strong indicators of malware as they are four times more likely to be used in the correct classification of malware than the next most significant op-codes (*inc*). Several op-codes have been identified as potential indicators of malware, which provides the basis for an improvement in detection techniques beyond current state of the art [3]. Finally, using the 'eigenvector' pre-filter, the dataset can safely remove irrelevant features.

References

[1] Arun Lakhotia, Eric Uday Kumar, and Michael Venable "A Method for Detecting Obfuscated Calls in Malicious Binaries" Software Engineering, IEEE Transactions on, Nov. 2005, Volume: 31, Issue:11, On page(s): 955 – 968

[2] Bilar, D "Opcodes as predictor for malware", Int. J. Electronic Security and Digital Forensics (2007) Vol. 1, No. 2, pp.156–168.

[3] Ulrich Bayer, Imam Habibi, Davide Balzarotti, Engin Kirda, and Christopher Kruegel, "A View on Current Malware Behaviors" Proceedings of the 2nd USENIX conference on Large-scale exploits and emergent threats: botnets, spyware, worms, and more. USENIX Association Berkeley, CA, USAYear of Publication: 2009.

[4] Boser, Bernhard E.; Guyon, Isabelle M.; and Vapnik, Vladimir N.; "A training algorithm for optimal margin classifiers", In Haussler, David (editor); 5th Annual ACM Workshop on COLT, pages 144–152, Pittsburgh, PA, 1992. ACM Press

[5] Clemens Kolbitsch, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, Xiaoyong Zhou and XiaoFeng Wang,"Effective and Efficient Malware Detection at the End Host", USENIX Association, 18th USENIX Security Symposium, pages 351-366

[6] Bilar, D. "Callgraph properties of executables and generative mechanisms", Journal AI Communications - Network Analysis in Natural Sciences and Engineering archive Volume 20 Issue 4, December 2007

[7] M. Christodorescu, S. Jha, S. Seshia D. Song, and R. Bryant, "Semantics-Aware Malware Detection.", IEEE Symposium on Security and Privacy, 2005 pp. 32–46.

[8] Vinod P., V. Laxmi and M. S. Gaur, "Static CFG Analyzer for Metamorphic Malware Code", Proceedings of the 2nd international conference on Security of information and networks ACM New York, NY, USA Pub 2009.

[9] Qinghua Zhang, Douglas S. Reeves, "MetaAware: Identifying Metamorphic Malware", Cyber Defense Laboratory, Computer Science Department North Carolina State University, Raleigh, NC 27695-8207

[10] Guillaume Bonfante, Matthieu Kaczmarek and Jean-Yves Marion "Control Flow Graphs as Malware Signatures ", Nancy-Universit´e - Loria - INPL - Ecole Natioanale Sup´erieure des Mines de Nancy B.P. 239, 54506 Vandoeuvre-l`es-Nancy C´edex, France

[11] James Newsome, Dawn Song, "Dynamic Taint Analysis for Automatic Detection, Analysis, and Signature Generation of Exploits on Commodity Software

[12] D. Ellis, J. Aiken, K. Attwood, and S. Tenaglia. "A Behavioral Approach to Worm Detection." ACM Workshop on Rapid Malcode, 2004, pp. 43–53.

[13] Yoshinori Okazaki, Izuru Sato and Shigeki Goto, "A New Intrusion Detection Method based on Process Profiling" Proceedings of the 2002 Symposium on Applications and the Internet

[14] S. Hofmeyr, S. Forrest, and A. Somayaji."Intrusion detection using sequences of system calls." Journal of Computer Security, pp. 151–180, 1998

[15] R. Sekar, M. Bendre, P. Bollineni, and D. Dhurjati. "A Fast Automaton-Based Approach for Detecting Anomalous Program Behaviors." IEEE Symposium on Security and Privacy, 2001.

[16] C. Ko, M. Ruschitzka, and K. Levitt. "Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach." IEEE Symposium on Security and Privacy, 1997.

[17] Chih-Wei Hsu, Chih-Chung Chang, and Chih-Jen Lin, "A Practical Guide to Support Vector Classification", Department of Computer Science National Taiwan University, Taipei 106, Taiwan, http://www.csie.ntu.edu.tw/~cjlin Initial version: 2003 Last updated: April 15, 2010

[18] Jesse C. Rabek, Roger I. Khazan, Scott M. Lewandowski, and Robert K.Cunningham."Detection of Injected, Dynamically Generated, and Obfuscated Malicious Code." ACM Workshop on Rapid Malcode, 2003 , pp. 76–82, 2003.

[19] Igor Santos, Yoseba K. Penya, Jaime Devesa and Pablo G. Bringas. "N-GRAMS-BASED FILE SIGNATURES FOR MALWARE DETECTION", S3Lab, Deusto Technological Foundation, Bilbao, Basque Country fisantos, ypenya, jdevesa, pgbg@tecnologico.deusto.es

[20] Sekar, R., Bendre, M., Dhurjati, D., & Bollineni, P. (2001). "A fast automaton-based method for detecting anomalous program behaviors", In R. Needham & M. Abadi (Eds), Proceedings of 2001 IEEE symposium on security and privacy (pp. 144--155), IEEE Computer Socitey, Los Alamitos, CA.

[21] W. Li, K.Wang, S. Stolfo, and B. Herzog. "Fileprints: Identifying file types by n-gram analysis", 6th IEEE Information Assurance Workshop, June 2005

[22] Cristianini N, Shawe-Taylor, J "An introduction to support vector machines: and other kernel-based learning", Pub 2000, ISBN 052178019

[23] Vanderbei R, "Linear Programming: Foundations and Extensions", Pub 2000, ISBN 0792373421

[24] Bernhard Schölkopf, Alexander Smola and Klaus-Robert Müller , "Kernel principal component analysis", Artificial Neural Networks — ICANN'97 Lecture Notes in Computer Science, 1997, Volume 1327/1997, 583-588, DOI: 10.1007/BFb0020217

[25] "Matlab Statistics Toolbox", http://www.mathworks.co.uk/help/toolbox/stats/, Last access 27 October 2011