O'kane *et al. Secur Inform (2016) 5:2*
DOI 10.1186/s13388-016-0027-2

Security Informatics

RESEARCH

Open Access

Detecting obfuscated malware using reduced opcode set and optimised runtime trace

Philip O'kane[*], Sakir Sezer and Kieran McLaughlin

Abstract

The research presented, investigates the optimal set of operational codes (opcodes) that create a robust indicator of malicious software (malware) and also determines a program's execution duration for accurate classification of benign and malicious software. The features extracted from the dataset are opcode density histograms, extracted during the program execution. The classifier used is a support vector machine and is configured to select those features to produce the optimal classification of malware over different program run lengths. The findings demonstrate that malware can be detected using dynamic analysis with relatively few opcodes.

Keywords: Component, Packers, Polymorphism, Metamorphism malware, Obfuscation, Dynamic analysis, Machine learning, SVM

Background

The malware industry has evolved into a well-organized $Billion marketplace operated by well-funded, multi-player syndicates that have invested large sums of money into malicious technologies, capable of evading traditional detection systems. To combat these advancements in malware, new detection approaches that mitigate the obfuscation methods employed by malware need to be found. A detection strategy that analyzes malicious activity on the host environment at run-time can foil malware attempts to evade detection. The proposed approach is the detection of malware using a support vector machine (SVM) on the feature (opcode density histograms) extracted during program execution. The experiments use feature filtering and feature selection to investigate all the Intel opcodes recorded during program execution.

While the full spectrum of opcodes is recorded, feature filtering is applied to narrow the search scope of the feature selection algorithm, which is applied across different program run-lengths. This research confirms that malware can be detected during the early phases of execution, possibly prior to any malicious activity.

"System overview" section describes the experimental framework and "Test platform" section details the test platform used to capture the program traces. "Dataset creation" section explains the dataset creation and is followed in "Opcode pre-filter" section with a description of the filtering method used. "Support vector machine" section introduces an SVM and describes the feature selection process. The results and observations are reviewed in "Discussion" section. Finally, "Conclusion" section concludes with a summary of the findings.

Related work

This research is an investigation into malware detection using N-gram analysis and is an extension of the work presented in [1]. However, a summary of the related research is given here to aid the discussion within this paper. Typical analysis approaches involve Control Flow Graphs (CFG), State Machines (modelling behaviour), analysing stack operations, taint analysis, API calls and N-gram analysis.

Code obfuscation is a popular weapon used by malware writers to evade detection [2]. Code obfuscation modifies the program code to produces a new version

*Correspondence: p.okane@qub.ac.uk
Centre for Secure Information Technologies, Queen's University Belfast, Belfast, Northern Ireland, UK

with the same functionality but with different Portable Executable (PE) file contents that are not known by the antivirus scanner. Obfuscation techniques such as packing are used by malware authors as well as legitimate software developers to compress and encrypt the PE. However, a second technique polymorphism [2] is used by malware. Polymorphic malware uses encryption to change the body of the malware which is governed by a decryption key that is changed each time the malware is executed creating a new permutation of the malware on each new infection. Eskandari et al. [3] propose to use program graph mining techniques for detecting polymorphic malware. However, these works employing subgraph matching to classify and detect malware. These API based methods are easily subverted by changing API call sequence or adding extra API calls that have no effect except to disrupt the call-graph.

Sung et al. [4] proposed an anomaly based detection using API call sequences to detect unknown and polymorphic malware using an Euclidian distance measurement between API sequences alignment of different call sequences. This API sequence alignment proposed by Sung approach is effectively a signature based approach since it ignores the frequency of the API calls.

Tian et al. [5] explored a method for classifying Trojan malware and demonstrated that function length plays a significant role in classifying malware and if combined with other features could result in an improvement in malware classification. Unfortunately, these techniques are easily subverted with the addition of innocuous API calls. Sami et al. [6] also propose a method of detecting malware based on mining API calls statically gathered from the Import Address Tables (IAT) of PE files.

Lakhotia et al. [7] investigated stack operations as a means to detect obfuscated function calls. His method modelled stack operation based on push, pop and rets opcodes. However, his approach failed to detect obfuscation when the stack is manipulated using other opcodes.

Bilar [8] demonstrated using static analysis that Windows PE files contain different opcode distributions for obfuscated and non-obfuscated code. Bilar's findings showed that opcodes such as adc, add, inc, ja, and sub could be used to detect malware.

In other research, Bilar [9] used statically generated CFG to show that a difference in program flow control structure exists between benign and malicious programs. Bilar concluded that malware has a simpler program flow structure, less interaction, fewer branches and less functionality than benign software.

More recent, research carried out by Agrawal et al. [10] also demonstrated a difference in the program flow control of malicious and benign software. Agrawal used an abstracted CFG that considered only the external artefacts of the program and used an 'edit distance' to compare the CFGs of programs. His findings show a difference in the flow control structure between benign and malicious programs.

N-gram analysis is the examination of sequences of bytes that can be used to detect malware. Using a machine learning algorithm, Santos et al. [11] demonstrated that N-gram analysis could be used to detect malware.

Santos et al. [12] perform static analysis on PE files to examine the similarity between malware families and the differences between benign and malicious software. Analysis with N-gram ($N = 1$) showed considerable similarity between families of malware, but no significant difference between benign and malicious software could be established. In a later paper, Santos et al. evaluated several machine learning algorithms [13] and showed that malware detection is possible using opcodes. Anderson et al. [14] combine both static and dynamic features in a multiple kernel learning framework to find a weighted combination of the data sources that produced an effective classification.

Shabtai et al. [15] used static analysis to evaluate the influence of N-gram sizes ($N = 1$–6) to detect malware using several classifiers and concluded that $N = 2$ performed best. Moskovitch et al. [16] also used N-gram analysis to investigate malware detection using opcodes and his findings concurred with Shabtai. Song et al. [17] explored the effects of polymorphism and confirmed that signature detection is easily evaded using polymorphism and is potentially on the brink of failure.

Due to the weakness in static analysis and the increase of obfuscated malware, it is difficult to ensure that all the code is thoroughly inspected. With the increasing amount of obfuscated malware being deployed, this research focuses on dynamic analysis (program run-time traces). Other dynamic analysis approaches use API calls to classify malware, which can easily be obfuscated by malware writers. Therefore, these experiments seek to identify run-time features (below the API calls) that can be used to identify malware. For this reason, the research investigates opcode density histograms obtained during program run-time as a means to identify malware.

System overview

The goal of this research, is two-fold (1) find a set of opcodes that are good indicators of malware and (2) determine how long the program needs to run in order to obtain an accurate classification. Figure 1 shows an overview of the experimental approach and to assist understanding, each section is labeled with a corresponding section heading used throughout this paper.

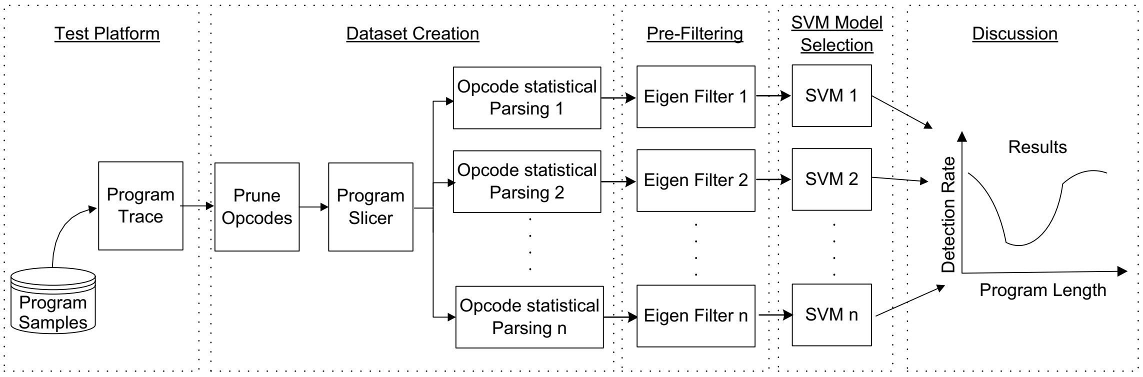


Fig. 1 Experiment overview

- 'Test Platform': The program samples are executed within the controlled environment to create program run-time traces.
- 'Dataset Creation': Each program trace is parsed and sliced into 14 different program run-lengths, creating 14 unique datasets defined by the number of opcodes executed.
- 'Pre-Filtering': A filter is applied to reduce the number of opcodes (features) that the SVM needs to process; thereby reducing the computational overhead during the SVM training phase.
- 'SVM Model Selection': is a process of selecting hyper-parameters (regularisation and kernel parameters) to achieve good out-of-sample generalisation.

Test platform

A native environment would provide the best platform in terms of the least tell-tale signs of a test environment and thereby mitigate any attempts by the malware to detect the test environment and exit early. However, other considerations need to be taken into account, such as ease of running the malware trace analysis.

A virtual platform is selected (QEMU-KVM), as the hypervisor provides isolation of the guest platform (Windows 7 OS test environment) from the underlying host OS and incorporates a backup and recovery tool that simplifies the removal of infected files. In addition to the virtual platform, a debugger is used to record the run-time behaviour of the programs under investigation. A plethora of debugging tools exist, with popular choices for malware analysis being IDA Pro, Ollydbg and WinDbg32 [18].

The Ollydbg debugger is chosen to record the program traces as it utilizes the StrongOD plug-in, which conceals the debugger's presence from the malware. When a debugger loads a program, the environment setting are changed, which enables the debugger to control the loaded program. Malware uses techniques to detect debuggers and avoid being analysed. StrongOD mitigates many of the anti-analysis techniques employed by malware and for an in-depth discussion on these techniques see work by [19, 20].

Dataset creation

Operational codes (Opcodes) are referred to as assembly language or machine language instructions and are CPU operations. They are usually represented by assembly language mnemonics.

Before realising the classifier, the raw data is distilled into a set of meaningful information that is used to train the classifier to predict unknown malicious and benign software samples. As discussed in the related work section, the features are constructed from program trace (p) and is represented as a set of instructions (I) and where n is the number of instructions:

$$p = I1, I2, \ldots In \tag{1}$$

An instruction consists of an opcode and operands. Opcodes, by themselves, are significant [8] and, therefore, only the opcodes are harvested with the operand being reduntant.

The program can, therefore, be defined as a set of ordered opcodes o:

$$p = o1, o2, \ldots on \tag{2}$$

Program slicing is used to investigate the effects of different program run lengths. Therefore, os is defined as a set of ordered opcodes within a program execution:

$$os \subseteq p \tag{3}$$

$$os = o1, o2 \ldots om \tag{4}$$

where m is the length of the program slice, 1k, 2k, 4k ... 8192k opcodes.

The opcode density histograms are constructed using the following steps:

- The program traces are created by recording the runtime opcodes that are executed when a program is run;
- The opcode densities for each program trace are calculated using the parser described below.

The dataset is created by expressing the features as a set of opcodes density, extracted from the runtime traces of Windows PE files. The dataset consists of 300 benign Windows PE files taken from the 'Windows Program Files' directory, and 350 malware files (Windows PE) downloaded from Vxheaven [21]. The datasets are constructed from different program run lengths, creating 14 distinct datasets. This new datasets are created by cropping the trace files into lengths based on the number of opcodes (1k-opcodes, 2k-opcodes etc.) prior to constructing a density histogram for each cropped trace file. The dataset creation starts by cropping the original dataset into 1k opcodes, and a density histogram is created, and is repeated for 2k, 4k, 8k, 16k,… 4096k and 8192k opcodes in length.

Opcode pre-filter

The computational effort associated with N-gram analysis is often referred to as the 'Curse of dimensionality' and was first coined by Bellman in 1961 to describe the exponential increase in computational effort associated with adding extra dimensions to a domain space. Using an SVM to examine all the opcode permutations over the complete opcode range creates a computational problem due to the high number of feature permutations produced.

The increased effort for each additional feature added is calculated using the following Eq. (5)

$$number\ of\ permutations = \frac{n!}{(n-r)!r!} \tag{5}$$

where n = total number of features in the dataset; r = number of features within the group of features under consideration.

To reduce the computational effort, the area of search is restricted to those features that contain the most information. This is achieved by applying a filtering process that ranks features according to the information that they contain and that is likely to be useful to the SVM [22]. Each feature is assigned an importance value using eigenvectors, thereby ranking the feature's usefulness as a means of classification.

Principal Component Analysis (PCA) is a transformation of the covariance matrix, and it is defined in (6) as per [23]:

$$C_{ij} = \frac{1}{n-1}\sum_{m=1}^{n}\left(X_{im}-\bar{X}_i\right)\left(X_{jm}-\bar{X}_j\right) \tag{6}$$

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

PCA compresses the data by mapping it into subspace (feature space) and creating a set of new variables. These new variables (feature space) that define the original data are called principal components (PCs), and retain all of the original information in the data. The new variables (PCs) are ordered by their contribution (usefulness/eigenvalue) to the total information.

The filter consists of two phases: Firstly, PCA is used to determine the most significant PCs, i.e. the number of PCs that contain 99.5 % of the data variance. PCA calculated that 8 PC values embodied 99.5 % of the total variance i.e. Eq. (10) n = 8. Secondly, the ranking value (R) is used to identify those opcodes that contain significant information (variance) and is calculated by multiplying the significant eigenvector column with the respective eigenvalues and then summing each row:

$$R_k = \sum_{k=1}^{n} V \cdot d_k \tag{7}$$

where R = Sum of the matrix variance; V = eigenvector; d = Eigenvalue scalar; n = 8; most significant values that represent 99.5 % of the variance within the data.

Figure 2 shows the ranking of features using (10), with the Y axis showing the ranking of the features, the X axis listing the features (opcodes) and the Z axis showing the different program run lengths. It can be seen that the top 20 ranked features vary with the program run length.

However, high ranking features such as rep, mov, add, etc. remain consistently high over the different program run lengths and the lowest ranking features such as *lea*, *loopd*, etc. remain consistently low over the different program run lengths. Considering the mid-ranking features, it can be seen that significant variations occur with different program run lengths.

Splitting these features into their opcode categories: arithmetic (*sub*, *dec*); logic (*xor*); and flow control (*je*, *jb*, *jmp*, *pop*, *nop* and *call*) infers that the program structure (flow control) changes with different program run lengths. Therefore, in the following experiment, the filter is run for each program run length to ensure the optimum feature selection.

Support vector machine

SVMs are classifiers that rely heavily on the optimal selection of hyper-parameters. A poor choice of values

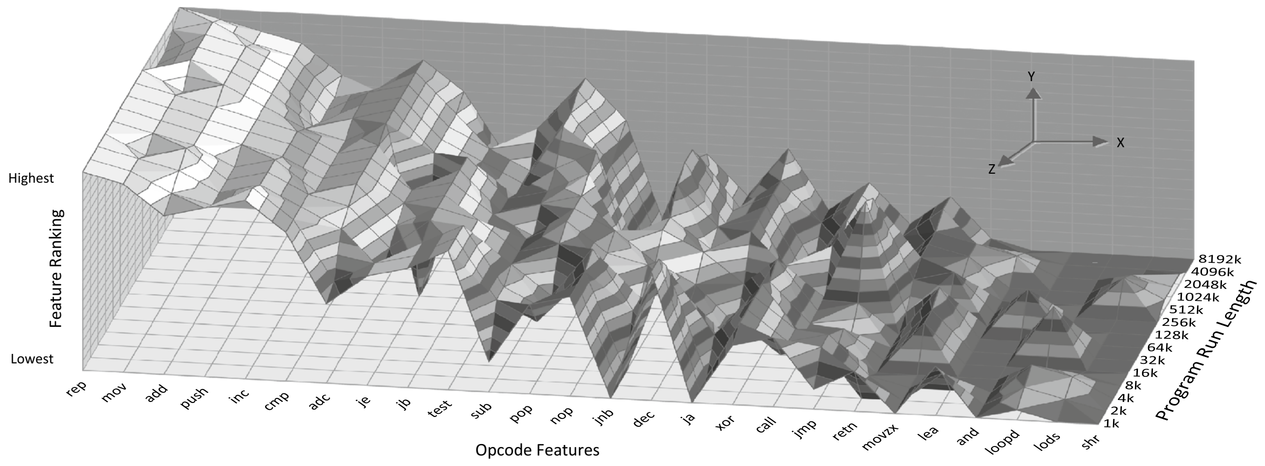


Fig. 2 Features ranked by eigenvalues

for a hyper-parameter can lead to poor performance in terms of overly complex hypothesis that leads to poor out-of-sample generalisation. The task of searching for optimal hyper-parameters, with respect to the performance measures (validation), is the called 'SVM Model Selection'.

The model selection process is categorised into:

- Kernel selection;
- Parameter grid search;
- Feature selection.

Herbrich et al. [24] demonstrated that, without normalisation, large values can lead to over-fitting and thereby reducing the out-of-sample generalisation. Normalisation can be performed in either the 'input space' or the 'feature space'.

Input Space normalisation is carried out on the input features (x) and is defined as:

$$\bar{x} = \frac{x}{\|x\|} \in R \tag{8}$$

Feature space normalisation is applied to the kernel rather than to the input vectors. Consider a kernel function K(x, y) which represents a dot-product in the feature space. Normalisation in the feature space requires a new kernel function definition [25]:

$$\bar{k}(x,y) = \frac{k(x,y)}{\sqrt{k(x,x)k(y,y)}} \in R \tag{9}$$

where R is a unit hypersphere.

Input space normalisation, as defined in (11), is implemented in the experiments presented in this paper.

An SVM maximizes the precision of the model by transposing the data into a feature space (high dimensional) where a hyper-plane separates the new features into their respective classes. This increases the class separation and is illustrated by way of an example, two opcodes *pop* and *ret* are used as they demonstrate the characteristics of kernel mapping. Figure 3 shows a plot of *pop* and *ret* features and how there mapping into feature space increases class separation.

The selection of an appropriate Kernel is key to the success of any machine learning algorithm. A linear kernel generally performs better at generalising the training phase into good test results where the data can be linearly separated. However, as shown in Fig. 5, the data is not linearly separated. Therefore, an RBF kernel (a non-linear decision plane) is used as it yields a greater accuracy than a linear kernel, as illustrated in Figs. 5 and 6.

The correct adjustment of the RBF kernel parameters significantly affects the performance of the SVM's ability to classify correctly, and poorly adjusted parameters can lead to either overfitting or underfitting. There are two parameters—C and λ. C is used to adjust the trade-off between bias and variance errors and λ determines the width of the decision boundary in feature space.

Two grid searches are performed to find the values of λ and C that produce an optimal SVM configuration. The first search is a coarse grain search, ranging from $\lambda = 1\ \mathrm{e}^{-5}$ to $1\ \mathrm{e}^{5}$ and $C = 0$–10. This is followed by a fine grain search (increments of 0.1) over a reduced range ($\lambda = \pm 10$, $C = 0$–3). The optimal performance was established with $\lambda = 1$ and $C = 0.8$.

Before continuing with the experiments, the results need to be placed in context. The measure of malware detection is based on:

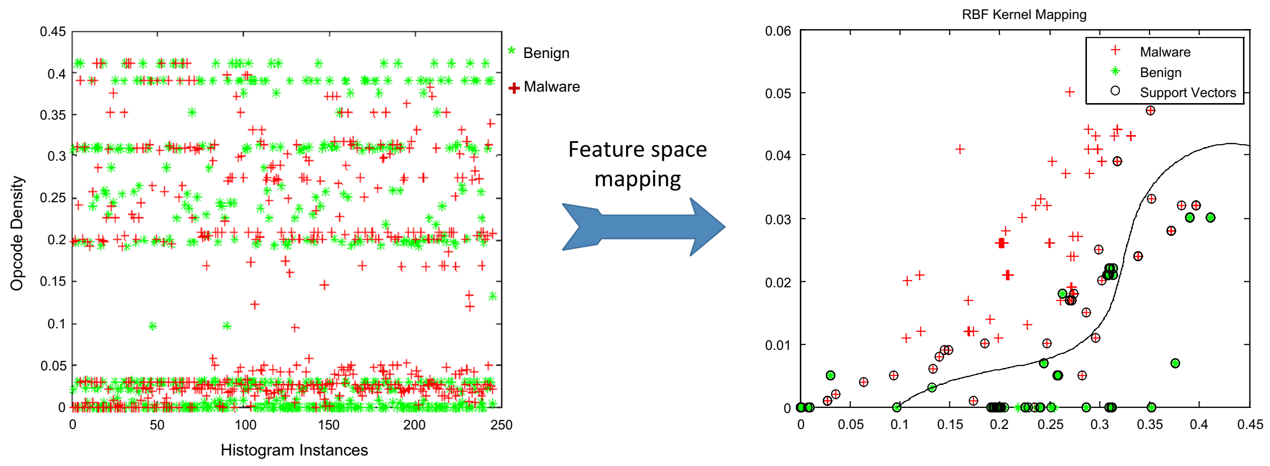


Fig. 3 Opcode density for pop and ret mapped into feature space

Detection accuracy is defined in (10) and is the correct classification of True Positive (*TP*) and True Negative (*TN*).

$$\text{Detection Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{10}$$

False positive (FP) is when a benign file is mistakenly classified as a malicious file and is defined in (11).

$$\text{False Positive} = \frac{FP}{TP + FP} \tag{11}$$

This is also known as a false alarm and can have a significant impact on malware detection systems. For example, if an antivirus program is configured to delete or quarantine infected files, a false positive can render a system or application unusable.

False negative (*FN*) is when a malicious file is mistakenly classified as benign and is defined in (12).

$$\text{False Negative} = \frac{FN}{TN + FN} \tag{12}$$

This occurs when an anti-virus security product fails to detect an instance of malware. This can be due to a zero-day attack or malware using obfuscation techniques to evade detection [2]. The impact of this security threat depends on whether the detection method is the last line of defence in the overall malware detection system.

False positives present a major problem, in that networks and host machines, can be taken out of service by the protective actions, as a consequent of alarms, such as quarantining or deleting a critical file. However, this paper focuses on end-point detection where false negatives present a security threat. Therefore, this research focuses on the minimisation of FN rate along with the detection accuracy.

In order to address the problem of FN rates, the optimisation function considers the FN rates by measuring the distance between the detection accuracy and the FN rate as described in (13). Steers the search by selecting those features that maximise $\text{OPT}_{\text{value}}$.

$$\text{OPT}_{\text{value}} = \text{Detection Accuracy} - D \times \text{FN Rate} \tag{13}$$

where *D* is a scalar used to adjust the sensitivity of the FN rate.

The challenge here is to choose a value of *D* that guides the SVM to select features that lead to the desired behaviour i.e. maximise the detection accuracy while minimising the FN rate. Setting $D = 1$ will direct the SVM to maximise the distance between detection accuracy and the FN rate. However, this may not yield the lowest FN rate. Therefore, *D* has to be greater than 1 to penalise the SVM for selecting non-minimal FN rates. A pilot study is carried out to find the value of *D* that will produce the maximum detection accuracy that has a low FN rate. It is not practical to investigate all values of *D* for all the combinations of opcodes studied in this experiment. Therefore, the cost function (13) is evaluated for $D = 1$, 1.5, 2 and 4. The results are shown in Fig. 4, where the upper part of the graph shows the detection accuracies for $D = 1$, 1.5, 2 and 4 against the program run lengths and the lower part of the graph shows the FN rates for $D = 1$, 1.5, 2 and 4 against the program run lengths. The following observations can be made:

- $\boldsymbol{D = 1}$ produces a detection accuracy ranging from 72.3 to 90.8 % (average 85.1 %) and a FN rate ranging from 0 to 10.79 % (average 5.4 %);
- $\boldsymbol{D = 1.5}$ produces a detection accuracy ranging from 70.8 to 90.8 % (average 84.4 %) and a FN rate ranging from 0 to 9.25 % (average 4.96 %);

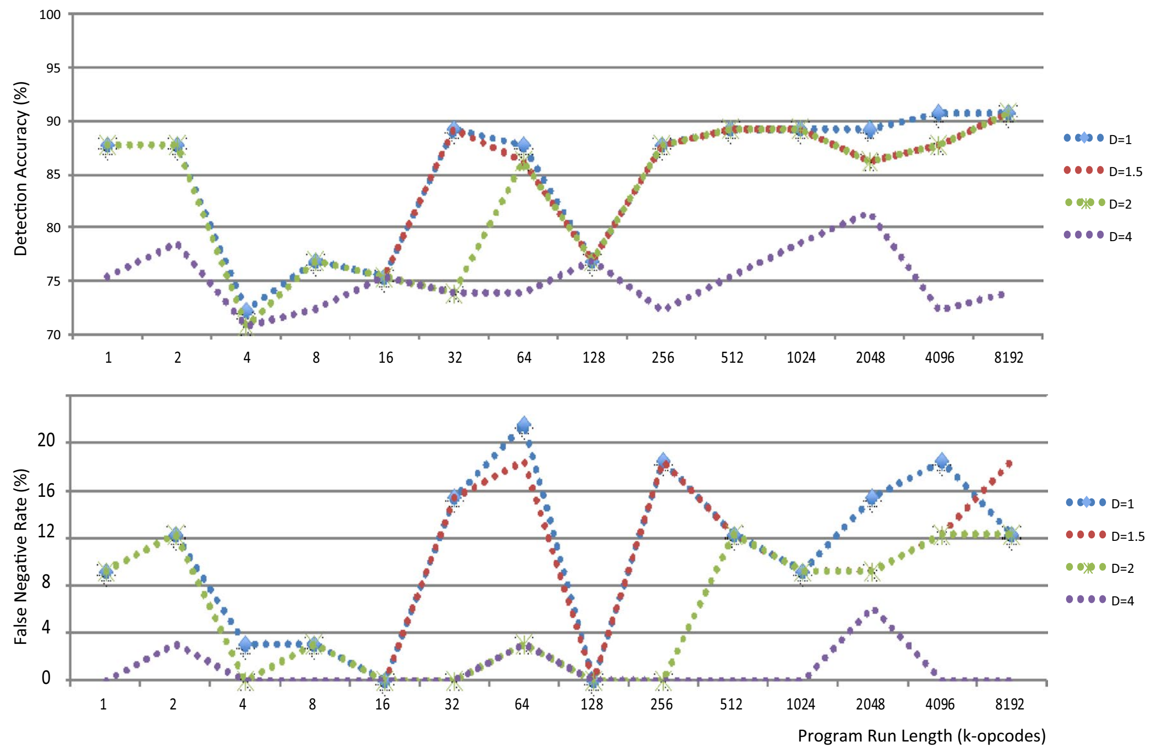


Fig. 4 Evaluation of the scalar *D* used in the cost function

- $\boldsymbol{D = 2}$ produces a detection accuracy ranging from 70.8 to 90.8 % (average 84.4 %) and a FN rate ranging from 0 to 6.18 % (average 2.98 %);
- $\boldsymbol{D = 4}$ produces a detection accuracy ranging from 70.8 to 81.5 % (average 75.1 %) and a FN rate ranging from 0 to 3.1 % (average 0.44 %).

Considering the average results; $D = 1$ and $D = 1.5$ yield very similar results with good detection accuracy of 85.1 and 84.4 % respectively but $D = 1$ and $D = 1.5$ produce a high FN rate of 5 % approximately. $D = 4$, produces an excellent FN rate of 0.44 %; however the corresponding detection accuracy is low at 75.1 %. $D = 2$ yields a compromise between $D = 1.5$ and $D = 4$ with a detection accuracy of 84.4 % and a FN rate of 2.98 %.

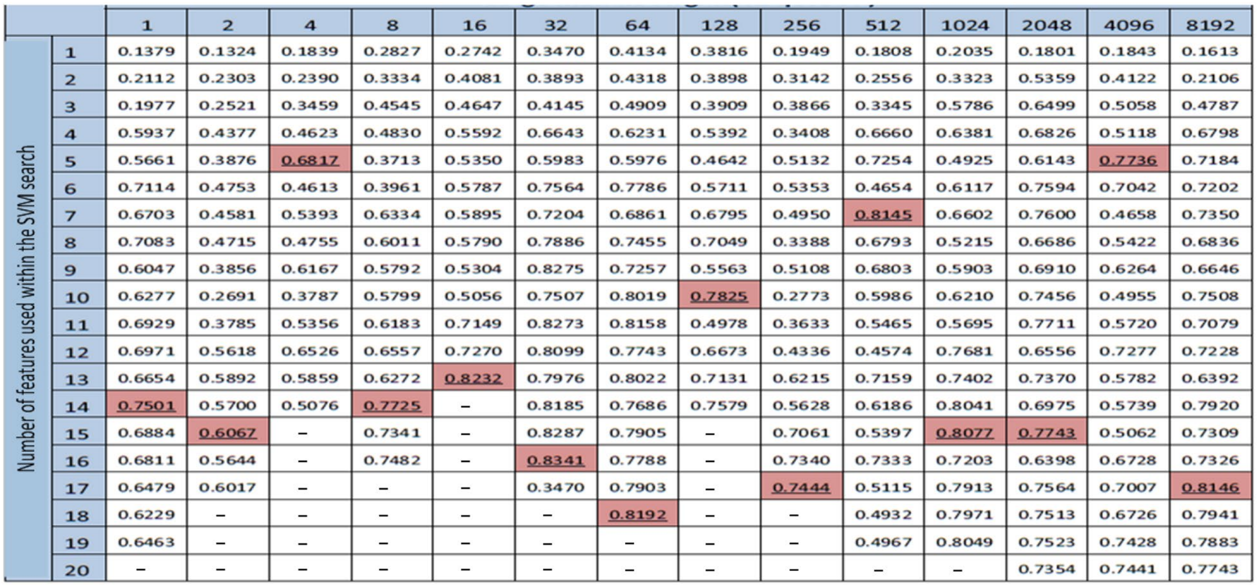
The results show that a lower value of D achieves a higher detection rate at the expense of the FN rate. A greater value of D results in lower FN rate at the cost of the detection rate. $D = 2$ delivers a low FN rate without overly penalising the detection accuracy and is therefore chosen as the steering function (13) for the remainder of the experiments carried out in this paper.

The SVM feature search uses Eq. (13) with $D = 2$ and scans all the combinations of opcodes. The search starts with one opcode and examines each of the filtered opcodes, testing for the largest value of (13). Next, the search is repeated, examining all unique combinations of two features and so forth, until all 20 opcode features are used. Table 1 shows the results, with the maximum optimisation value shaded.

Note, the columns '1 to 20' represents the number of opcodes in each test, with the rows '1, 2, 4, 8,..., 8192' represent the program run lengths in k-opcodes. The optimisation value is shown against that number of opcodes and program run length. I.e. the first row shows the cost function value (measure of performance) for a single opcode feature, with the maximum optimisation value for each program, run length and the second row shows the cost function values for two opcode features, with the maximum optimisation value for each program run length and so on. In Table 1, the maximum values are identified with an underscore. It can be seen that a point is reached, when adding more features results in a reduction of the maximum value; the assumption made is that over-fitting is occurring. As already mentioned, the grid search is guided by the performance metric in Eq. (13) and is measured using tenfold cross-validation.

While an optimal detection rate is a vital characteristic of any detection system, FP and FN rates need to be considered. These experiments are aimed at end host detection, and it can be argued that FN rates outweigh the importance of FP rates. Therefore, the aim of our approach is to convict all suspicious files and let further malware analysis determine their true status.

Table 1 Program run length versus %optimisation value



Number of features used within the SVM search	1	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192
1	0.1379	0.1324	0.1839	0.2827	0.2742	0.3470	0.4134	0.3816	0.1949	0.1808	0.2035	0.1801	0.1843	0.1613
2	0.2112	0.2303	0.2390	0.3334	0.4081	0.3893	0.4318	0.3898	0.3142	0.2556	0.3323	0.5359	0.4122	0.2106
3	0.1977	0.2521	0.3459	0.4545	0.4647	0.4145	0.4909	0.3909	0.3866	0.3345	0.5786	0.6499	0.5058	0.4787
4	0.5937	0.4377	0.4623	0.4830	0.5592	0.6643	0.6231	0.5392	0.3408	0.6660	0.6381	0.6826	0.5118	0.6798
5	0.5661	0.3876	0.6817	0.3713	0.5350	0.5983	0.5976	0.4642	0.5132	0.7254	0.4925	0.6143	0.7736	0.7184
6	0.7114	0.4753	0.4613	0.3961	0.5787	0.7564	0.7786	0.5711	0.5353	0.4654	0.6117	0.7594	0.7042	0.7202
7	0.6703	0.4581	0.5393	0.6334	0.5895	0.7204	0.6861	0.6795	0.4950	0.8145	0.6602	0.7600	0.4658	0.7350
8	0.7083	0.4715	0.4755	0.6011	0.5790	0.7886	0.7455	0.7049	0.3388	0.6793	0.5215	0.6686	0.5422	0.6836
9	0.6047	0.3856	0.6167	0.5792	0.5304	0.8275	0.7257	0.5563	0.5108	0.6803	0.5903	0.6910	0.6264	0.6646
10	0.6277	0.2691	0.3787	0.5799	0.5056	0.7507	0.8019	0.7825	0.2773	0.5986	0.6210	0.7456	0.4955	0.7508
11	0.6929	0.3785	0.5356	0.6183	0.7149	0.8273	0.8158	0.4978	0.3633	0.5465	0.5695	0.7711	0.5720	0.7079
12	0.6971	0.5618	0.6526	0.6557	0.7270	0.8099	0.7743	0.6673	0.4336	0.4574	0.7681	0.6556	0.7277	0.7228
13	0.6654	0.5892	0.5859	0.6272	0.8232	0.7976	0.8022	0.7131	0.6215	0.7159	0.7402	0.7370	0.5782	0.6392
14	0.7501	0.5700	0.5076	0.7725	–	0.8185	0.7686	0.7579	0.5628	0.6186	0.8041	0.6975	0.5739	0.7920
15	0.6884	0.6067	–	0.7341	–	0.8287	0.7905	–	0.7061	0.5397	0.8077	0.7743	0.5062	0.7309
16	0.6811	0.5644	–	0.7482	–	0.8341	0.7788	–	0.7340	0.7333	0.7203	0.6398	0.6728	0.7326
17	0.6479	0.6017	–	–	–	0.3470	0.7903	–	0.7444	0.5115	0.7913	0.7564	0.7007	0.8146
18	0.6229	–	–	–	–	–	0.8192	–	–	0.4932	0.7971	0.7513	0.6726	0.7941
19	0.6463	–	–	–	–	–	–	–	–	0.4967	0.8049	0.7523	0.7428	0.7883
20	–	–	–	–	–	–	–	–	–	–	–	0.7354	0.7441	0.7743

In a final testing phase, bootstrapping is introduced to ensure a robust measure of out-of-sample generalisation performance. The concern is that sample clustering may result, as many of the malware samples belong to the same malware family and often have similar file names. The Parser used, reads files from the directory (in alphabetic order) and creates the density histograms, which may result in clustering of malware samples that belong to the same family. Therefore, randomly selecting test samples prior to the SVM processing will ensure that the validation data is random.

Bootstrapping is implemented in Matlab using the built-in function 'randperm' to randomly split the dataset into training and testing data. As shown in the script below, the labels are first overwritten with ones to indicate benign samples for training and zeros for malicious training samples. The script then randomly overwrites 10 % of the benign and malicious files to test as shown in the script below.

```
% Matlab script – Randomised Cross-validation
% Randomly select test data
% first set all the data samples to training data
inputDataBenignType(:) = ones; # Benign training samples
inputDataMalwareType(:) = zeros; # Malicious training samples
    % randomly selects benign samples for testing
      n = size(inputDataBenignType);
      k=n(2)/10;
     index = randperm(n(2),int8(k));
     inputDataBenignType(index) = 3; #Benign testing samples
    % Randomly select malware sample for testing
     clear index;
     n = size(inputDataMalwareType);
     k=n(2)/10;
     index = randperm(n(2),int8(k));
     inputDataMalwareType(index) = 2;#Malicious testing samples
```

The premise of Bootstrapping is that, in the absence of the true distribution, a conclusion about the distribution can be drawn from the samples obtained. Parke et al. [26] suggest that 200 iterations are sufficient to obtain a mean and standard deviation value of statistical importance.

As previously mentioned, the optimisation value is used to find a set of features that yield the optimum combination of detection accuracy and FN rate (as shown in Table 1). Figure 5 shows the detection accuracy and the FN rates for the different program run lengths derived from the maximum optimisation values (Table 1). The results shown in Fig. 5 are validated using 200 iterations of the Bootstrapping method. Figure 5 shows that medium program run lengths produce the best detection accuracy coupled with the lowest FN rates. However, good detection rates are achieved for short program run lengths but detection rates need to be considered in conjunction with the corresponding FN rate.

While there is no universally defined valued that specifies a 'good detection' system; the values obtained in these experiments need to be placed in context. Curtsinger et al. [27], defined 0.003 % FN as an 'extremely low false negative system' and Dahl [28] classified a system with < 5 % FN as a 'reasonably low' false negative rate. Ye et al. [29] examined several detection methods and found that FN rates varied significantly with different classifiers such as Naive Nayes with 10.4 % FN; SVM with 1.8 % FN; Decision Tree (J48) with 2.2 % FN; Intelligent Malware Detection System (IMDS) with 1.6 % FN.

While our approach fails to satisfy the criteria of 'extremely low' FN, it does meet the criteria for a 'reasonably low' FN rate for the program run lengths of 1k and above 8k.

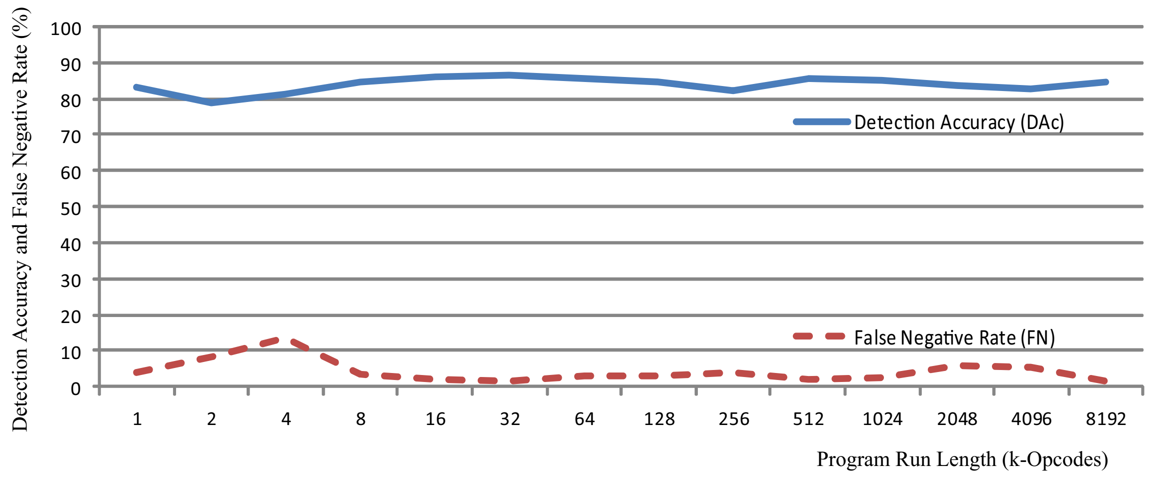


Fig. 5 Detection accuracy and FN rate versus program run length

Figure 6 shows the detection accuracies (DR) and the false negative rates (FN) plotted against the number of features used for classification. Figure 6 is constructed by taking an average of the detection accuracies and false negative rates across the program run lengths (as indicated by the maximum optimisation values shown in Table 1) for feature groups (1–20). This shows the relationship between the number of features and the detection accuracy and false negative rates. It can be seen that both the detection accuracy and false negative rate improves with an increasing number of features (up to 13 features), and degrades and becomes more inconsistent (greater variance) thereafter.

It can be seen (Fig. 6), that adding more features does not always improve the results. The performance of both the detection accuracy and the FN rate peaks at 13 features (average), above which the performance degrades. This degradation is pervasive in all the program run lengths. It is believed that this is likely due to over-fitting caused by too much variance being introduced by the additional features. Again, the smallest variance occurs with 13 features (average).

Discussion

The research presented, investigated the use of run-time opcode traces to discriminate between malicious and

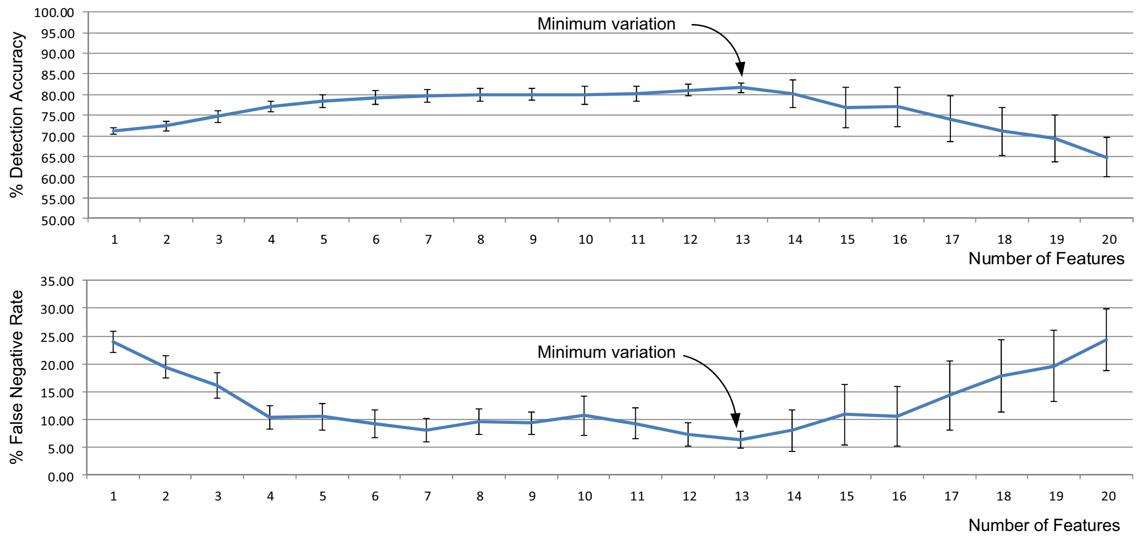


Fig. 6 Detection accuracy and FN versus number of features

Table 2 Optimum features for malware detection at selected run lengths (K-opcodes)

Length (k-opcodes)	Logical & arithmetic							Address manipulation						Flow control												Detection rate	False negative rate	False positive rate
1		*add*		*dec*	*inc*	*sub*	*xor*			*mov*		*pop*		*call*	*jmp*			*rep*	*retn*			*jb*	*je*		*test*	0.8341	0.042	0.301
2	*adc*	*add*		*dec*	*inc*	*sub*	*xor*					*pop*	*push*	*call*	*jmp*	*loopd*	*nop*					*jb*			*test*	0.7874	0.0847	0.36
4		*add*			*inc*									*call*	*jmp*								*je*			0.8127	0.1349	0.247
8	*adc*	*add*				*sub*	*xor*		*lods*			*pop*	*push*				*nop*	*rep*		*cmp*		*jb*	*je*	*jnb*	*test*	0.8439	0.0357	0.297
16	*adc*	*add*		*dec*			*xor*			*mov*		*pop*			*jmp*		*nop*	*rep*		*cmp*	*ja*	*jb*	*je*			0.861	0.0189	0.277
32	*adc*	*add*			*inc*		*xor*				*movzx*			*call*	*jmp*	*loopd*	*nop*	*rep*		*cmp*		*jb*	*je*	*jnb*	*test*	0.8631	0.0158	0.277
64	*adc*	*add*		*dec*	*inc*		*xor*			*mov*	*movzx*	*pop*	*push*	*call*	*jmp*		*nop*	*rep*		*cmp*		*jb*	*je*	*jnb*	*test*	0.8582	0.0297	0.273
128	*adc*	*add*	*and*	*dec*			*xor*					*pop*		*call*				*rep*		*cmp*				*jnb*		0.8475	0.0325	0.293
256	*adc*	*add*	*and*	*dec*	*inc*	*sub*				*mov*	*movzx*	*pop*	*push*	*call*				*rep*	*retn*		*ja*	*jb*	*je*	*jnb*		0.8237	0.0376	0.343
512		*add*						*lea*					*push*	*call*	*jmp*				*retn*						*test*	0.8559	0.0207	0.287
1024	*adc*	*add*			*inc*	*sub*		*lea*		*mov*	*movzx*	*pop*	*push*					*rep*	*retn*		*ja*	*jb*	*je*		*test*	0.8511	0.0234	0.297
2048	*adc*	*add*		*dec*		*sub*		*lea*		*mov*	*movzx*	*pop*	*push*	*call*					*retn*		*ja*		*je*	*jnb*	*test*	0.8377	0.0587	0.28
4096	*adc*					*sub*	*xor*					*pop*								*cmp*						0.8258	0.0525	0.313
8192	*adc*	*add*		*dec*	*inc*					*mov*	*movzx*	*pop*	*push*	*call*	*jmp*			*rep*	*retn*	*cmp*	*ja*		*je*	*jnb*	*test*	0.8472	0.0136	0.313
Occur	11	13	2	8	8	7	8	3	1	7	6	11	8	10	8	2	5	9	6	7	5	8	10	7	9			

benign software. Table 2 summarizes the results in terms of performance (Detection, false negative and false positive rates) versus program run lengths with the corresponding opcode features.

The performance rates are listed in the right-hand column (taken from Table 1) and correspond to different program run lengths as indicated in the left-most columns i.e. 1k-opcodes, 2k-opcodes, 4k-opcodes, 8k-opcodes, etc. The central columns list the opcodes used to achieve these results.

Encryption-based malware often use the *xor* (opcode) to perform their encryption and decryption. Table 2 shows that *xor* frequently appears in the shorter program run lengths. This frequent appearance of *xor* is expected as the unpacking/decrypting occurs at the start of a program. An exception is that the 4k-opcodes length program does not use *xor* to classify benign and malicious software.

Figure 7 presents opcode categories in terms of their ability to detect malware, which is constructed from the information presented in Table 2. Figure 7 is calculated for each category and then normalised using the total area of all the categories. The results show that the flow control category is the most effective at 59 % followed by Logic and Arithmetic at 31 %. This implies that a program structure (Flow Control) is the most significant indicator of benign and malicious software followed by the logic and arithmetic components of the program, which concurs with Bilar [8, 9] findings.

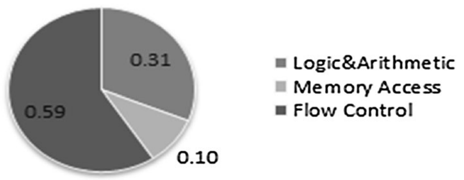


Fig. 7 Breakdown of malware detection by opcode category

In summary, several observations can be made:

1. More is not always best; the optimum number of features varies with the program run length, but typically (average) 13 opcodes yield the best results. As an example, the maximum detection accuracy (83.4 %) for the 1k-opcode program run length is achieved with 14 features. However, adding more features decreases the detection accuracy, which is typical of all the program run lengths.
2. Table 2 shows that *xor* is used as an indicator of malware for shorter program run lengths i,e 1k-opcdes to 126k-opcodes (excluding 4k-opcode). This is expected behaviour as encrypted malware frequently uses *xor* to perform its decryption and is normally exercised in the early stages of the program execution.
   An exception, is the absence of *xor* in the 4k-opcode length, which is not clearly understood beyond the fact that the machine learning algorithm did not chose it as an optimal feature for this program run length i.e. other features performed better for this particular program run length.
3. While FN is not ideal, many of the program run lengths (excluding 2 and 4K-opcodes), are be considered to be a 'reasonably low' FN rate (FN < 5 %). The relative short program run lengths of 2 and

4k-opcodes have high FN rates of 8.47 and 13.49 % respectively. The other program lengths present good detection rates of 81–89 %, the FN rates between 1.58 and 5.87 %.

4. The maximum detection accuracy of 86.3 % with the lowest FN rate (1.58 %) is obtained for a program run length of 32k-opcodes. However, a program run length of 1K-opcodes produces a good detection accuracy of 83.4 %, with a respectable FN rate of 4.2 %.
5. The bottom row (Occur) of Table 2 shows the number of times a particular opcode was selected by the classifier (SVM) as an indicator of malware. For example, opcode *add* was chosen 13 times out of 14 program run lengths, whereas, opcode *lods* was only chosen once for the 8k-opcode run length. What is clear, is that the opcodes chosen (by the SVM) change relative to different program run lengths. Our observations show that shorter program run lengths rely on 'logic and arithmetic' and 'flow control', whereas the longer program run lengths rely more on 'flow control' opcodes. This infers that the detection of longer program run length relies on the complexity of the call structure of a program. This is consistent with Bilar [9] finding that showed malware having a less complex call structure than non-malicious software.

Conclusion

The experimental work carried out in this research investigated the use of an SVM to detect malware. The features used by the SVM were derived from program traces obtained from program execution. The findings indicate that encrypted malware can be detected using opcodes obtained during program execution. The investigation continued to establish an optimal program run-length for malware detection. The dataset was constructed from run-time opcodes and compiled into density histograms and then filtered prior to SVM analysis. A feature selection cost function was identified and used to steer the SVM for optimal performance. The full spectrum of opcodes were examined for information, and the search for the optimal opcodes was quickly narrowed using an Eigenvector filter.

The findings show that malware detection is possible for very short program run lengths of 1k-opcodes that produce a detection rate of 83.41 % and a FN rate of 4.2 %. Using mid-range program run lengths also yields a sound detection rate. However, their corresponding FN rates deteriorate. The 1k-opcode characteristics provide a basis to detect malware during run-time, potentially before the program can complete its malicious activity, i.e. during their unpacking and deciphering phase.

The research presented, provides an alternative malware detection approach that is capable of detecting obfuscated malware and possible Zero-day attacks. With a small group of features and short program run length, a real world application could be implemented that detects malware with minimal computation, enabling a practical real world solution to detect obfuscated malware.

Authors' contributions

We have read the ICMJE guidelines and can confirm that the authors PO, SS, KM contributed intellectually to the material presented in this manuscript. All authors read and approved the final manuscript.

Competing interests

We the authors of this paper confirm that we do not have any competing financial, professional or personal interests that would influence the performance or presentation of the work described in this manuscript.

Received: 14 July 2015 Accepted: 20 April 2016
Published online: 04 May 2016

References

1. Okane P, Sakir S, McLaughlin K, Im EG (2014) Malware detection: program run length against detection rate. IET Softw 8(1):42–51
2. O'Kane P, Sezer S, McLaughlin K (2011) Obfuscation: the hidden malware. IEEE Secur Privacy 9(5):41–47
3. Eskandari M, Hashemi S (2012) A graph mining approach for detecting unknown malwares. J Vis Lang Comput 23(3):154–162
4. Sung A, Xu J, Chavez P, Mukkamala S, et al (2004) Static analyzer of vicious executables (save). In: Proceedings of the 20th annual computer security applications conference, 2004
5. Tian R, Batten L, Islam R, et al (2009) An automated classification system based on the strings of trojan and virus families. In: Proceedings of the 4rd international conference on malicious and unwanted software: MALWARE, 2009, pp 23–30
6. Sami A, Yadegari B, Rahimi H, et al (2010) Malware detection based on mining API calls. In: Proceedings of the 2010 ACM symposium on applied computing, 2010, pp 1020–1025
7. Lakhotia A, Kumar EU, Venable M (2005) A method for detecting obfuscated calls in malicious binaries. IEEE Trans Softw Eng 31(11):955–968
8. Bilar D (2007) Opcodes as predictor for malware. Int J Electron Secur Digit Forensics 1(2):156–168
9. Bilar D (2007) Callgraph properties of executables and generative mechanisms. AI Communications, special issue on Network Analysis in Natural Sciences and Engineering 20(4): 231–243
10. Agrawal H (2011) Detection of global metamorphic malware variants using control and data flow analysis. WIPO Patent No. 2011119940, 30 September 2011
11. I Santos, YK Penya, J Devesa, PG Garcia (2009) N-grams-based file signatures for malware detection. S3Lab, Deusto Technological Foundation
12. Santos I, Brezo F, Nieves J, Penya YK, Sanz B, Laorden C, Bringas PG (2010) Opcode-sequence-based malware detection. In: Proceedings of the 2nd international symposium on engineering secure software and systems (ESSoS), Pisa (Italy), 3–4th February 2010, LNCS 5965, pp 35–43
13. Santos I, Brezo F, Ugarte-Pedrero X, Bringas PG (2013) Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf Sci 231:64–82
14. Anderson B, Storlie C, Lane T (2012, October) Improving malware classification: bridging the static/dynamic gap. In: Proceedings of the 5th ACM workshop on Security and artificial intelligence, pp 3–14. ACM
15. Shabtai A, Moskovitch R, Feher C, Dolev S, Elovici Y (2012) Detecting unknown malicious code by applying classification techniques on opcode patterns. Secur Inf 1(1):1–22
16. Moskovitch R, Feher C, Tzachar N, Berger E, Gitelman M, Dolev S, Elovici Y (2008) Unknown malcode detection using opcode representation. In: Proceedings of the 1st European conference on intelligence and security informatics (EuroISI08), 2008, pp 204–215

17. Song Y, Locasto M, Stavro A (2007) On the infeasibility of modeling polymorphic shellcode. In: ACM CCS, 2007, pp 541–551
18. Eilam E (2011) Reversing: secrets of reverse engineering. Wiley, New York
19. Ferrie P (2011) The ultimate anti debugge reference. http://pferrie.host22.com/papers/antidebug.pdf. Written May 2011, last accessed 11 October 2012
20. Chen X, Andersen J, Mao ZM, Bailey M, Nazario J (2008) Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In: ICDSN proceedings, 2008, pp 177–186
21. Heaven VX (2013) Malware collection. http://vxheaven.org/vl.php. Last accessed Oct 2013
22. O'Kane P, Sezer S, McLaughlin K, Im EG (2013) SVM training phase reduction using dataset feature filtering for malware detection. IEEE Trans Inf Forensics Secur 8(3):500–509
23. Kantardzic M (2011) Data mining: concepts, models, methods, and algorithms. Wiley, London. ISBN 0-471-22852-4
24. Herbrich R, Graepel T (2002) A PAC-Bayesian margin bound for linear classifiers. IEEE Trans Inf Theory 48(12):3140–3150
25. Graf ABA, Borer S (2001) Normalization in support vector machines., Pattern RecognitionSpringer, Berlin, Heidelberg, pp 277–282
26. Parke J, Holford NHG, Charles BG (1999) A procedure for generating bootstrap samples for the validation of nonlinear mixed-effects population models. Comput Methods Programs Biomed 59(1):19–29
27. Curtsinger C, Livshits B, Zorn B, Seifert C (2011) Zozzle: low-overhead mostly static javascript malware detection. In: Proceedings of the usenix security symposium, Aug 2011
28. Dahl G, Stokes JW, Deng L, Yu D (2013) Large-scale malware classification using random projections and neural networks. Poster (MLSP-P5.4), May ICASSP 2013, Vancouver Canada, IEEE Signal Processing Society, 2013
29. Ye Y, Wang D, Li T, Ye D (2007) IMDS: intelligent malware detection system. In: Proceedings of the 13th ACM SIGKDD international conference on knowledge discovery and data mining. ACM, 2007