ResearchGate

See discussions, stats, and author profiles for this publication at: https://www.researchgate.net/publication/274008692

N-gram density based malware detection

Conference Paper · January 2014

DOI: 10.1109/WSCAR.2014.6916806

CITATIONS
2

READS
86

3 authors:

Philip Okane
Queen's University Belfast
4 PUBLICATIONS **63** CITATIONS
SEE PROFILE

Sakir Sezer
Queen's University Belfast
178 PUBLICATIONS **1,050** CITATIONS
SEE PROFILE

Kieran Mclaughlin
Queen's University Belfast
54 PUBLICATIONS **254** CITATIONS
SEE PROFILE

Some of the authors of this publication are also working on these related projects:

Project Vulnerability and Exploits Analysis View project

Project SOC Conference View project

All content following this page was uploaded by Philip Okane on 18 August 2015.

The user has requested enhancement of the downloaded file. All in-text references underlined in blue are added to the original document and are linked to publications on ResearchGate, letting you access and read them immediately.

www.ietdl.org

Published in IET Software
Received on 18th February 2013
Revised on 24th June 2013
Accepted on 24th July 2013
doi: 10.1049/iet-sen.2013.0020

ISSN 1751-8806

Malware detection: program run length against detection rate

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]Division of Computer Science and Engineering, Hanyang University, Seoul 133-791, Korea
E-mail: pokane17@qub.ac.uk

Abstract: N-gram analysis is an approach that investigates the structure of a program using bytes, characters or text strings. This research uses dynamic analysis to investigate malware detection using a classification approach based on N-gram analysis. A key issue with dynamic analysis is the length of time a program has to be run to ensure a correct classification. The motivation for this research is to find the optimum subset of operational codes (opcodes) that make the best indicators of malware and to determine how long a program has to be monitored to ensure an accurate support vector machine (SVM) classification of benign and malicious software. The experiments within this study represent programs as opcode density histograms gained through dynamic analysis for different program run periods. A SVM is used as the program classifier to determine the ability of different program run lengths to correctly determine the presence of malicious software. The findings show that malware can be detected with different program run lengths using a small number of opcodes.

1 Introduction

Recent years have seen a large growth in malware; as a result, previously successful methods such as signature detection and monitoring suspected code for known security vulnerabilities are 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 opcodes 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 opcode density features using supervised learning machines performed on features obtained from run-time traces. A support vector machine (SVM) is one form of supervised learning machine, used in this research as the basis for malware detection. The approach taken in this paper investigates the full spectrum of opcodes, through dynamic analysis, to identify a subset of opcodes and to determine the optimum run-time for a program to maximise the probability of malware detection. This is achieved using a novel combination of feature pre-filtering and feature selection to investigate the ability to detect malware during different program run lengths. This research shows that it is possible to detect malware in the early stages of its execution, potentially before it can do any harm.

Section 2 discusses related research and in Section 3, the experiments are placed into context with an overview of the experimental approach.

Section 4 specifies the environment used to capture the dataset and introduces anti-analysis approaches taken by malware writers. This is followed, in Section 5, with an explanation of how the dataset is created and the filtering approach used. The SVM is introduced in Section 6 and describes the search sequence used to find the optimum opcodes to be used for malware detection. Section 7 summarises the results and key characteristics recorded during these experiments. Finally, Sections 8 and 9 conclude the paper by summarising the findings.

2 Related work

Extensive research has been undertaken 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 coarse and fine grain analysis, state machines to model system behaviour, the mapping of stack operations and N-gram analysis.

Lakhotia *et al.* [1] presented a state machine method to detect obfuscated calls relating to *push*, *pop* and *ret* opcodes 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.

Bilar [2] used static analysis to obtain opcode distributions from windows portable executable (PE) files that could be used to identify polymorphic and metaphoric malware. Bilar's findings show that many prevalent opcodes (*mov, push, call* etc.) did not make good indicators of malware.

However, lesser frequent opcodes such as *ja, adc, sub, inc* and *add* proved to be better indicators of malware.

In another research, Bilar [3] compared the statically generated CFG of benign and malicious code. His 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.

Ye *et al.* [4] developed an intelligent malware detection system (IMDS) that analyses Windows API sequences called by PE files. The analysis uses objective-oriented association (OOA) rule-based classifier to detect malware. They present data that shows their system outperforms other classifiers such as Naive Bayes, SVM and Decision Tree (J4.8).

N-gram analysis is based on a signature approach that relies on small sequences of strings or byte codes that are used to detect malware. Santos *et al.* [5] demonstrated that N-gram signatures could be used to detect unknown malware. The experiment extracted code and text fragments from a large database of program executions to form signatures that are classified using machine learning methods.

Sekar *et al.* [6] implemented an N-gram approach and compared it to a finite state automaton (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.* [7] describe N-gram analysis, at byte level, to compose models derived from learning the file types that 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 for 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-gram analysis with greater number of bytes, that is, $N = 2$, $N = 3$ etc.

Santos *et al.* [8] examined the similarity between families of malware and the dissimilarity between malware and benign software using opcode-sequence gained through static analysis of PE files. Santos' findings showed that using $N = 1$ (N-gram) weighted opcode frequency, a high degree of similarity existed between families of malware, but the similarity rating between malicious and benign software was too high to be an effective classifier. However, using $N = 2$ produced a greater difference between malicious and benign software. In a later paper, Santos *et al.* examined a SVM with a single-class learning approach that used the frequency of opcodes obtained from static analysis [9]. Santos reduced the effort of labelling that is required for the training phase and highlights the issue of unpacking malware. Santos *et al.* [10] evaluated several learning methods (K-nearest neighbour, Bayesian network, SVM, etc.) and showed that malware can be detected with a high degree of accuracy using opcode-sequence.

Shabtai *et al.* [11] used static analysis to examine the effectiveness of malware detection when using different N-gram size ($N = 1$–6) with various classifiers. Shabtai's findings showed that $N = 2$ performed best. Moskovitch *et al.* [12] also investigated the detection of malware using N-gram opcode analysis. While Moskovitch used different classification methods to Shabtai, their findings also showed that $N = 2$ performed best.

Song *et al.* [13] investigated the effect of polymorphic engines used in malware to evade detection by network content matching. Song's findings demonstrated that signature detection on the contents of a file is at the brink of failure and with little effort, the malware writers could make signature detection redundant.

This demonstrates the key weaknesses in static analysis; that is, looking at the contents of a malicious file that is likely to be obfuscated and therefore making it difficult to ensure that all areas of the code are correctly investigated. Owing to the high level of obfuscation found in many malicious programs, this research is focusing on behavioural analysis using program run-time analysis.

In this vein, we have chosen to focus our research on the identification of malware using opcodes. However, we have chosen to obtain the opcodes from run-time program traces. Bilar [2] demonstrated that opcode N-gram ($N = 1$) could be used to detect malware. Therefore, we started with $N = 1$ and have demonstrated that malware can be identified with a reduced set of features. Although the main thrust of this research is N-gram analysis based on run-time monitoring of programs, there is no literature available relating to the optimal time that malware needs to be run in order that it can be identified; with many researchers suggesting that the malware be run to completion or until a nefarious activity is carried out. Therefore this research not only identifies a subset of opcodes but also determines program run length to achieve optimal detection rates. To this end, this research concentrates on finding an optimal run-length for dynamic analysis with a suitable subset of features.

3 System overview

The motivation for this research is to find an optimal program run-length that can be used for benign and malicious software classification. This research is two-fold: (1) identify a subset of opcodes that produce the best classification of malicious and benign software and (2) determine the duration that a program needs to run to ensure accurate classification of malicious and benign software. Fig. 1 is an illustration of the system and consists of several stages. To aid understanding, each section of the figure is given a heading, which corresponds to a section of this paper. 'Test Platform': the programs under investigation are run in a test environment using a debug tool to monitor the runtime opcodes and produce program traces. 'Dataset Creation': the program traces are then parsed and sliced into various duration lengths, defined as the number of opcodes executed. Performing analysis on short increments of program slices over the complete duration of the program would be ideal. However, this is not practical due to the large amount of additional computing effort required. Therefore, an x^2 scale is used so that the early stages of the program can be scrutinised while still having the opportunity to examine the program at a later stage. All slices are taken from the beginning of the program and are 1, 2, 4, 8, 16, …,4 096 and 8192 k in length. Next, the slices of the original dataset are parsed into opcode density histograms. Fourteen histograms are created; one for each slice and form the feature set used by the pre-filters and SVMs. 'Pre-Filter': is a ranking algorithm that assigns a ranking to each feature based on the variance of the data when mapped into subspace. The pre-filter reduces the number of opcodes that are processed by the SVM and therefore reduces the computational overhead; thus making

© The Institution of Engineering and Technology 2014

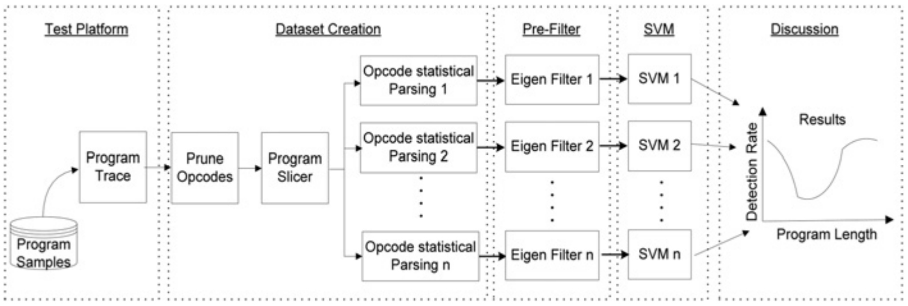

Fig. 1 *Experiment overview*

the training phase of the SVM a viable approach. 'SVM': is used to further refine and identify the optimal feature set used to detect malware. Although the system caters for all Intel opcodes, the program traces consist only of 150 different opcodes.

4 Test platform

The main challenge with dynamic analysis is to ensure that the malicious code path is executed during investigation. Three dynamic approaches exist: (I) Native (debugger), (II) Emulation and (III) Virtualisation. Each has to address malware evasion techniques that may attempt to fool the dynamic analysis into completing its analysis without running the malicious code [14].

While native environments provide 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 WinDbg, [15] which are popular choices for malware analysis.

Emulation and virtualisation both provide a decoupling layer that enables the OS that hosts the malware to operate in a safe isolated environment. The key difference between emulators and virtualisation is that emulators perform all the remapping of the OS platforms in software whereas in the case of virtualisation many of the hosting activities are performed in hardware. QEMU supports both emulation and virtualisation. When used in the virtualised mode, special hardware functionality is required (Intel VT-x) and normally performance is faster than emulators [16]. A virtualisation approach is chosen as it provides isolation by decoupling the virtual machine (malware environment) from the OS and outperforms emulators. 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 machines on the same physical machine. Prior to performing the analysis, the virtualised 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 [17], which helps to prevent the malware from detecting the presence of a monitoring tool (Ollydbg). While StrongOD does not claim to mitigate all anti-analysis techniques, it is useful in fooling the malware samples contained within this dataset. When a debugger loads a program under investigation (debuggee), it changes the environment in which the program runs to allow the debugger to interact and control the debuggee. Malware programs use techniques to detect when a debugger is running and enter into 'evade mode' preventing the debugger from correctly analysing the malware. For a comprehensive list of techniques, see work by Ferrie and Chen [18, 19].

5 Dataset creation

Operational codes (Opcodes) are machine language instructions that perform CPU operations on operands such as arithmetic, memory/data manipulation, logical operations and program flow control. There are 15 opcodes directly referred to in this paper, which are grouped as follows:

Arithmetic operations – *add*, *adc* (add with carry flag), *inc*, *sub*;
Memory manipulation – *lea* (load effective address), *mov*, *pop*, *push* (retrieve and place data onto a stack);
Logical operations – *xor* (exclusive OR);
Program flow control – *call* (jump to a function), *ret* (return from function) *cmp* (compare data), *ja*, *je* (jump if a condition is met), *rep* (a prefix that repeats the specified operation).

Prior to implementing a classifier, attributes are extracted from the raw data to create meaningful information in the form of features that can be used to train the classifier so that it can successfully predict the classifications of unknown samples. Bilar [3] have shown that a difference in structure between benign and malicious software exists. Therefore, the hypothesis is that this structural difference is present in the opcode population density. The features that are used for these experiments are opcode density histograms gained by executing a program.

Therefore, representing a program as a set of instructions; where a program trace p can be defined as a sequence of instructions $\boldsymbol{I}$ and $\boldsymbol{n}$ is the number of instructions

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

A program instruction is a 2-tuple composed of an opcode and one or more operands. Opcodes are significant in themselves [3] and therefore the operand component of the instruction is discarded.

Therefore the program is redefined as a set of ordered opcodes o

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

An opcode sequence, *os* is defined as an ordered subgroup of opcodes within the program execution, where

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

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

and m is the length of os.

The program run-length slicing requires that the program order remains unchanged by processes used to capture and record the program instructions.

The raw data is parsed to discard data that is not relevant to the processing experiments. A bespoke parser was implemented to extract opcode density histograms from the program traces. The key components of the parser are

Pre-scan: Performs a scan on all the trace files to construct a complete list of opcodes present within the trace files. This list is sorted into alphabetic order and is used by the statistical scan function to ensure that all the histograms created are of the same size and order.

Statistical scan: The trace files are scanned for opcodes and an opcode density histogram is constructed for each file. The previously defined list of opcodes is used so that any opcodes that were not encountered within a particular file can be padded out with zeros (no occurrence of that opcode within this file) to ensure that all the opcodes are present and in the same order in each histogram, making it easier for histogram manipulations. The histogram is written directly to two Excel files (one for benign opcode densities and another for malicious opcode densities). Several options exist, which are:

- norm: The output histogram can be formatted as absolute value or normalised opcode densities.
- *r*: A restriction can be applied to the opcodes to focus on selected opcodes only.
- n-gram: This flag is used to group opcodes together to form opcode sequences for the purpose of N-gram analysis, that is, $N = 2,3,4\ldots$. When the flag is not specified, the default is $N = 1$, which is the case for the experiments presented in this paper.
- slice *n*: This enables a section of the trace file to be cropped for analysis. The size is expressed in *K* opcodes. Here the statistics are analysed from the start of the trace to a number *n* of opcodes encountered.

The –norm (normalise) parameter is selected for this work as it reduces the effect of different program run-lengths.

The Excel files contain a header row that is reserved for classification labels. This can be manually edited or rewritten when read in by the MATLAB (R2011b) scripts.

The label definitions are: 0 and 1 indicating benign and malicious files, respectively, which are used for training and labels 2 and 3 representing benign and malicious files, respectively, which are used for testing.

The dataset is constructed by representing each executable file as a set of opcode density histograms obtained from runtime traces. Recall that the operands associated with each opcode are omitted and that only the opcodes are recorded.

Classification tasks involve separating data into training and test data. Each training-set instance is assigned a target value/label, that is, benign or malicious. The goal of the SVM is to identify a subset of opcodes that best predicts the target values of the test data. There are 260 benign Windows XP files taken from the 'Program Files' directory (230 training file, 30 validation files). There are 350 malware files (310 training files, 40 validation files) which are malicious windows executable files downloaded from Vxheaven website (this website is no longer available) and consists of a range of malicious activities such as: backdoor downloaders, system attack, fake alert/warnings, Ad-Aware and information stealer.

To ensure that Ollydbg correctly unpacked and executed the malware, samples were restricted to programs that Ollydbg correctly identified as packed or encrypted.

The malware samples were run for 3 min ensuring that not only the loading and unpacking phases were recorded but also that malicious activity occurred, that is, pop-up, writing to the disk or registry files.

Although there are 344 Intel opcodes, only 150 different opcodes are recorded during the captured datasets for all programs traced during this experiment. As previously stated, the dataset is normalised by calculating the percentage density of opcodes (rather than the absolute opcode count) in order to remove time variance introduced by different run lengths of the various programs. The dataset is sorted into most commonly occurring opcodes as illustrated in Fig. 2.

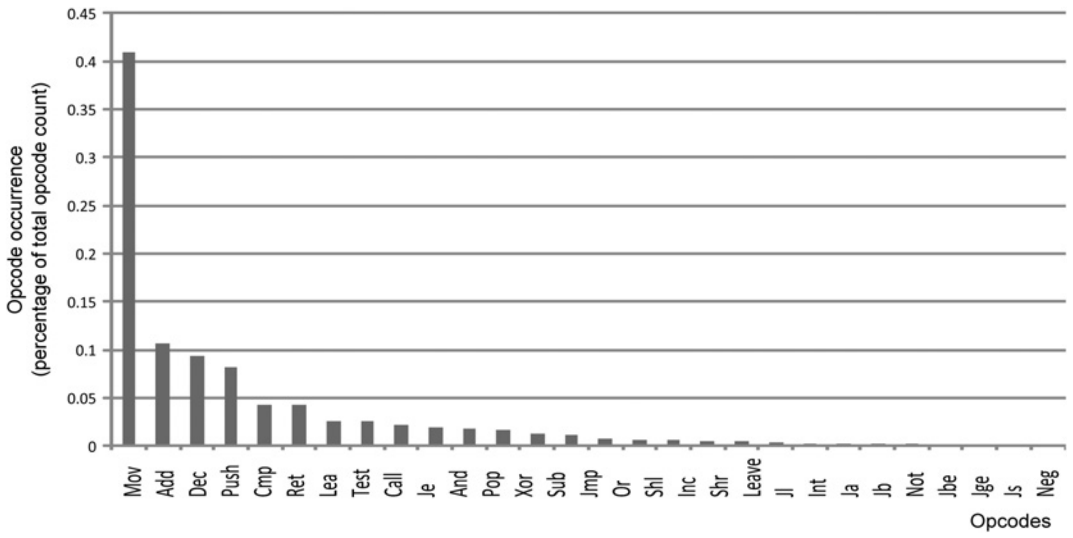

Fig. 2 *Histogram: opcode percentage*

The data values are a percentage of the opcodes within a particular program. For example, 0 means that the opcode does not occur within that program trace or 0.25 means that 25% of the program trace comprises of that opcode. To improve the performance of the SVM, the data is linearly scaled (0, +1).

The dataset is manipulated by using different program lengths to create 14 distinct datasets using different program run lengths. The first dataset is created, by cropping the original dataset to 1 k opcodes and a density histogram is created, reflecting the percentage of each opcode found as illustrated in Fig 2. This is repeated for 2, 4, 8, 16, …, 4096 and 8192 k opcodes in length.

6 Pre-filter

Using a SVM to perform a full spectrum N-gram analysis presents a dimensionality problem, in terms of the number of raw feature permutations produced, which would result in a high computation cost during the SVM training phase. To reduce this effort and narrow the area of search, this research uses a pre-filter that selects those features that are likely to provide the maximum information to the SVM. This section details a filter approach derived from subspace analysis using principal component analysis (PCA) and eigenvectors. This assigns a value of importance to each opcode, thereby ranking its usefulness as a classification feature. PCA is a transformation of the covariance matrix and it is defined in (5) as per [20]

$$C_{ij} = \frac{1}{n-1}\sum_{m=1}^{n}\left(X_{im}-\bar{X}_i\right)\left(X_{jm}-\bar{X}_j\right) \tag{5}$$

where C = Covariance matrix of PCA transformation;
X = dataset value;
$\bar{X}$ = 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) which are uncorrelated and are ordered by their contribution (usefulness/eigenvalue) to the total information that each contains.

The pre-filter has two stages: First, PCA is used to identify the significant PCs, that is, the number of PCs that correlate to greater than 95% of the data variance. Based on PCA, it was identified that the first 8 PC values accounted for 99.5% of the variance. Therefore the 8 largest (most significant) eigenvalues are used to locate the most significant eigenvectors (meaningful data) which is $K = 8$ in (9). Secondly, PCA is an algorithm that operates on variance of data, that is, a covariance matrix of the training dataset, which is calculated in MATLAB as follows

$$C = \mathrm{cov}(\text{training Data}) \tag{6}$$

$$[V, \lambda] = \mathrm{eig}(C) \tag{7}$$

$$d = \mathrm{diag}(\lambda) \tag{8}$$

The significant values are calculated by multiplying the significant eigenvector column with the respective eigenvalues and then summing each row

$$R_k = \sum_{k=1}^{8} V\, d_k \tag{9}$$

where R = sum of the matrix variance;
C = covariance;
V = eigenvector;
λ = eigenvalue matrix;
d = eigenvalue scalar;

The results for the 15 highest ranking (out of 150) opcodes for the longest programs run (8 192 k) are illustrated in Table 1; note $\lambda_{3->7}$ values are not shown due to layout considerations.

Table 1 is sorted based on the sum (largest first) of the eigenvectors. This same information is also displayed in Fig. 3 in column form. As a proof of concept, the SVM is used to create a reference model from the original dataset to validate the proposed filter. The SVM reference model is created by configuring the SVM to search sequentially through the dataset for groups of opcodes that can differentiate between benign and malicious software. The test was configured to record the opcodes that performed a detection rate better than 70, 75, 80, 85, 90 and 95%. The algorithm starts by selecting a group of seven opcodes and performs a training cycle and tests the classifier against 10% hold-back data. Any results with a test detection rate better than those specified above are recorded into their respective bin. This is repeated until all permutations for the seven opcodes are complete; the number of opcodes within the selected group is decremented and the scan is repeated. The search is repeated for groups of 6, 5 and 4 opcodes. Only unique opcodes groups are selected for each SVM classification test and no duplicates of repeated opcode patterns are processed. The results are summed with a weighting factor to give importance to those opcodes involved in the higher detection rates.

$$y_i = \sum_{i=1}^{150} \boldsymbol{b}_1(\boldsymbol{x}_1) \tag{10}$$

where i is the individual opcode index, that is, 1–150.
b is the weighting for the respective bin, that is, 0.7, 0.75, 0.8, 0.85, 0.9 and 0.95.

Table 1 Ranking by eigenvector

Opcode	$\lambda_1 x_{[1..150]}(10^{-3})$	$\lambda_2 x_{[1..150]}(10^{-3})$	$\lambda_8 x_{[1..150]}(10^{-3})$	$\sum_{k=1}^{8}(10^{-3})$
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

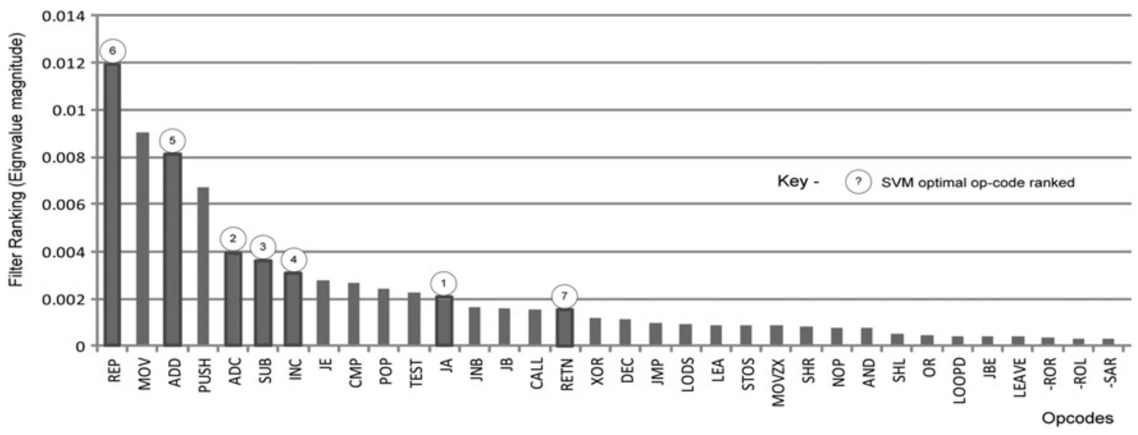

Fig. 3 *Eigenvector magnitude*

x is the count in each bin for achieving a particular detection rate.

The top seven opcodes were chosen as they account for 95% of the total count. These seven opcodes are overlaid on the eigenvector ranked values shown in Fig. 3, which shows a comparison between the opcodes ranked by the filter and those chosen by the SVM. The 'eigenvector' filter has not only correctly chosen those opcodes selected by the SVM but has grouped them into the most significant range (highest eigenvalues).

Given that the eight opcodes chosen by the SVM are clustered among the top 12 opcodes, that is, top 10% thereby removing the 90% irrelevant opcodes, this demonstrates that the eigenvector approach is an effective filtering mechanism to reduce opcodes prior to the SVM training phase.

This filtering process is carried out for all program run-lengths and the results are shown in Table 2, sorted based on the sum (largest first) of the eigenvectors. Table 2 shows a similar filter output with little difference between the various program run-lengths, which implies a consistent program structure regardless of program run-length. While only 11 opcodes are shown in Table 2, all 20 opcodes selected by the filter are then fed into the SVM for further reduction and selection.

7 Support vector machine

SVM is a machine learning classifier, introduced by Boser *et al.* in 1992 [21]. A SVM maximises the predictive accuracy of a model by mapping the data into a high-dimensional feature space and builds a hyperplane to separate different instances into their respective classes. SVMs use kernel method algorithms that depend on dot-product functions, which can be replaced by other kernel functions, such as a linear, radial basis function (RBF) or polynomial kernel that can be used to transpose the data as it is mapped into a higher dimensional feature space. With the correct choice of kernel, the data mapped into feature space is transformed in a way that increases the class separation, simplifying the construction of a hyperplane easier. This has two advantages: First, the ability to generate a non-linear decision plane and secondly, enabling the user to apply a classification to data that does not have an intuitive approach, that is, SVM training when the data has a non-regular or unknown distribution. The dataset consists of 150 different opcodes (which are obtained during the longest program runs), 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, for example, in the case of the linear kernel and the polynomial kernel, large attribute values might cause numerical problems [22].

In this research, RBF kernel is used as it is considered a reasonable first choice in that it provides a non-linear

Table 2 Eigenvalue ranking

Length (*k*)	Largest				Opcodes selected on magnitude of eigenvalues						Smallest
1	*rep*	*mov*	*inc*	*push*	*add*	*cmp*	*test*	*dec*	*jb*	*je*	*mul*
2	*rep*	*mov*	*inc*	*add*	*push*	*cmp*	*test*	*dec*	*jb*	*je*	*mul*
4	*rep*	*mov*	*add*	*inc*	*push*	*cmp*	*test*	*adc*	*jb*	*dec*	*mul*
8	*rep*	*add*	*mov*	*inc*	*push*	*cmp*	*adc*	*jb*	*test*	*mul*	*je*
16	*rep*	*mov*	*add*	*inc*	*Push*	*adc*	*cmp*	*jb*	*test*	*je*	*pop*
32	*rep*	*mov*	*add*	*push*	*Inc*	*adc*	*cmp*	*jb*	*mul*	*pop*	*test*
64	*rep*	*mov*	*add*	*push*	*Inc*	*adc*	*cmp*	*jb*	*mul*	*pop*	*sub*
128	*rep*	*mov*	*add*	*push*	*Inc*	*adc*	*cmp*	*jb*	*mul*	*pop*	*sub*
256	*rep*	*add*	*mov*	*push*	*Adc*	*inc*	*cmp*	*jb*	*pop*	*je*	*sub*
512	*mov*	*add*	*rep*	*push*	*Adc*	*cmp*	*inc*	*mul*	*je*	*jb*	*sub*
1 024	*mov*	*add*	*rep*	*push*	*Adc*	*cmp*	*inc*	*mul*	*je*	*sub*	*jnb*
2 048	*mov*	*add*	*rep*	*push*	*Adc*	*cmp*	*inc*	*je*	*jnb*	*jb*	*pop*
4 096	*mov*	*rep*	*add*	*push*	*Adc*	*inc*	*cmp*	*je*	*sub*	*pop*	*test*
8 192	*rep*	*mov*	*add*	*push*	*Adc*	*sub*	*inc*	*je*	*cmp*	*pop*	*test*

© The Institution of Engineering and Technology 2014

Table 3 Program run length against % detection rate

Length (*k*)	Number of features used within the SVM search																			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	73	73	87	87	89	**90**	89	89	89	89	89	89	89	89	89	87	87	87	87	–
2	67	67	87	87	87	**89**	87	86	86	86	86	86	84	84	83	83	66	56	56	–
4	68	68	**81**	81	81	81	81	73	72	72	72	72	72	61	58	56	56	56	56	–
8	–	73	80	81	**86**	86	84	78	81	76	72	72	69	69	67	67	56	56	56	–
16	–	70	81	81	**83**	81	81	78	80	75	72	69	66	64	58	56	56	56	56	–
32	–	73	83	83	**89**	86	86	86	86	86	86	83	83	83	81	67	56	56	56	–
64	–	–	83	**84**	84	84	84	84	84	84	84	84	84	81	84	81	81	72	56	–
128	–	76	84	84	84	81	**86**	84	78	73	72	72	66	60	56	56	56	56	56	–
256	–	75	84	**87**	87	86	87	87	87	87	87	87	87	87	80	75	70	58	56	–
512	–	73	86	87	87	87	87	**89**	89	89	87	89	87	87	86	86	86	84	75	–
1 024	–	–	86	87	87	89	89	**90**	90	90	89	89	89	89	87	87	87	86	86	–
2 048	–	73	87	87	89	89	89	89	**90**	89	89	89	89	89	87	87	86	86	80	–
4 096	–	71	86	86	86	**90**	89	89	89	89	89	89	89	89	89	89	87	86	84	–
8 192	–	–	84	89	90	90	90	**92**	90	90	92	89	89	90	87	86	83	69	63	–

mapping of samples into a higher dimensional space. This caters for instances where the relationship between the class label and attributes are non-linear. The parameters for the RBF kernel play an important role in the performance of SVM as an incorrect selection could result in either over-fitting or under-fitting of the training data. These parameters are C and λ, where C is a penalty value that determines the tradeoff between bias and variance of the model and λ is a parameter that determines the shape and the width of the decision boundary of the data points mapped into the feature space. Grid searches are a reliable method of parameter selection (SVM tuning). To perform an exhaustive fine grain grid search, over the entire range of λ and C, it is computationally intensive and therefore the approach taken in this research is to perform two grid searches. The search starts with a coarse grain search covering ($\lambda = 1\text{e}{-5}$ to $1\text{e}5$ and $C = 0 \text{–} > 10$) followed with a fine grain search (increments 0.1) within a reduced range ($\lambda = \pm 10$, $C = 0 \text{–} > 3$). The optimal values identified are $\lambda = 1$ and $C = 0.8$.

The SVM searches through all the opcodes and identifies a subset of opcodes that produces the maximum malware detection rate. The test was configured to record only those opcodes that achieved the highest detection rate. The SVM starts with one opcode and scans across the filtered opcodes. Next, the SVM uses two features and scans for all unique permutations for that number of features. Each search is repeated by incrementing the number of features used until all 20 opcode features are used in the search. The results are shown in Table 3 with the maximum % detection rates shaded. Note that on the horizontal axis, 1 to 20 represent the number of opcodes within each test and on the vertical axis 1, 2, 4, 8… represent the program run-length. The best detection rate is shown against that number of opcodes and program run-length, that is, the first column represents a single opcode feature with the highest percentage detection rate for each program run-length and the second column represents the two opcode features with the highest percentage detection rate for each program run-length and so on. In Table 3 the maximum values are underscored. Two assumptions are made: (i) when the maximum detection is exceeded and starts to reduce, over-fitting is occurring and (ii) when multiple maximum detection rates occur, the group with the least number of opcodes is selected as the additional opcodes add no value and are considered irrelevant.

Even though a good detection rate is important, careful consideration must be given to the false negative rate in the context of malware analysis. In malware detection, it can be easily argued that the false negative rate is as important as detection rate, if not more so. The aim on any good detection strategy is to convict all suspect files and let

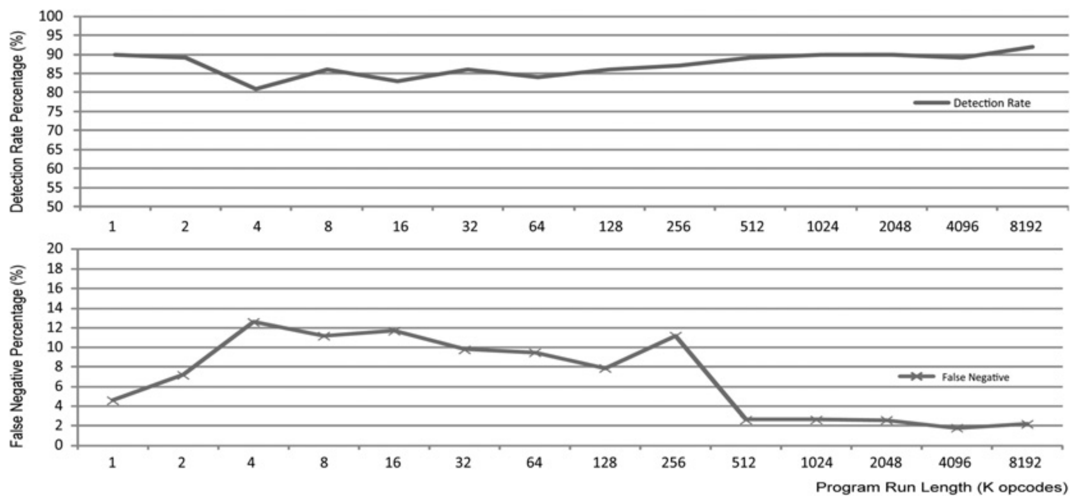

Fig. 4 *Detection rate and false negative rate against program run length*

further malware analysis determine if the file in question is benign. An ideal system would have a 0% false negative and a good detection rate and therefore any files that are wrongly labelled as malicious can be analysed further to determine their true stats. A further point to consider is that the detection rate of the proposed system must be high enough to justify the overhead of running the system. Fig. 4 is a plot of the highest detection rates (from Table 3 results) with the corresponding false negative rates. It can be seen that the detection rate is not constant. The maximum detection rate is achieved for the longest program run lengths, but good detection rates can be obtained for the shortest program run-lengths. However, the false negative rates need to be considered in the context of the detection rate as shown in Fig. 4. Fig. 4 shows the best detection rates for the various program run-lengths at the top of the graph while the bottom of the graph shows the false negative, which has a magnified scale so that the variations in false negatives can be compared to the detection rate above. There is no scale of values that is universally defined; the values obtained here need to be placed in context with other detection system. Curtsinger *et al.* define 0.003% false negative as an ‘extremely low false negative system’ [23] and Dahl classified a system with <5% false negative as a ‘reasonably low’ false negative rate [24]. Ye *et al.* [4] carried out performance analysis on several detection methods which are Naive Nayes having 10.4% false negative, SVM having 1.8% false negative, Decision Tree (J48) having 2.2% and their proposed method IMDS which has 1.6% false negative. Clearly, our proposed system does not meet the criteria of an ‘extremely low’ false negative system. Our system does meet the criteria for a ‘reasonably low’ false negative rate at various program run-lengths, for example, at 1 k and above 512 k.

8 Discussion

This study presents an argument for the analysis of run-time opcode traces to detect malware. The main thrust of this research is to find the optimum subset of opcodes that make the best indicators of malware and to determine how long a program has to be monitored to ensure an accurate SVM classifier. Table 4 lists the maximum detection rate in the right-hand column (from the results of Table 3) for the various program run-lengths as indicated in the left-most column, that is, 1 k, 2 k, 4 k …. The respective opcodes used to achieve these detection rates are shown in the central shaded columns. The shaded columns categorise the opcodes into their different functions, that is, logical and arithmetic, address manipulation and flow control. Flow control is further divided into redirection operators and conditional redirection operators. An additional column, ‘category score’, is added to aid the discussion of program structure by adding a opcode count for each opcode functional category, for example, *n*, *y*, *z*: states that the maximum detection rate was achieved with *n* logic & arithmetic operators, *y* address manipulations and *z* flow control operators. The category score column shows the number of opcodes in each functional area and the final column shows the maximum detection rate for that program run length.

Key points:

(1) Considering Table 3, it can be seen that when all 20 of the opcodes are used, the classifier fails to produce any meaning classification and therefore the detection rates were not recorded. Looking closely at this step change in detection rate between using 19 and 20 opcodes, the *mov* opcode is the only difference in all instances (1 k, 2 k, 4 k, etc.) and has a catastrophic impact on the results. It is not clear why this is the case as *mov* opcode population distribution is similar in shape to other opcode distribution shapes. However, the *mov* opcode has a high population density in all programs, both benign and malicious, consisting of 30%–40% of the program. Scaling was applied to the SVM input in order to avoid such numerical difficulties within the kernel function. This researcher would suggest that the large variability associated with the *mov* opcode creates an over-fitting problem.

(2) More is not always best; ignoring the extreme example of 20 opcodes, the optimum number of opcode varies with program run-length and ranges from 3 to 8 opcodes. As an example, considering the maximum detection of 90% for the 1 k length program shows that six opcodes yield the best results, but adding any additional opcodes reduces the detection rate, which is characteristic of all the program run-lengths.

(3) Polymorphic and encryption-based malware commonly use the *xor* instruction as the transfer/encryption function. It can be seen that *xor* plays a key role in malware detection for short program run-lengths, such as 1, 2, 8 and 16 k, which is expected behaviour as much of the unpacking and decrypting of software often occurs at the earlier stage of a program. One exception is the 4 k length program, as the SVM does not select *xor* as a good indicator of malware.

(4) False negative rates are not ideal, but looking at program run-lengths of 1 k and greater than 512 K, the system can be considered as having a ‘reasonably low’ false negative rate (false negative <5%). While mid-length program slices present good detection rates (81%–>89%), the false negative rates (7.2%–>12.6) could be considered unacceptable.

(5) Table 4 shows that the subset of optimum opcodes does not remain constant; in fact they change with different program run-lengths. Detection using shorter program run-lengths relies on opcodes *xor, sub, add, adc,* which are ‘logic and arithmetic’, whereas the longer program run-lengths rely on *add, inc, sub, xor, pop* and *rep* that are predominantly ‘flow control’ opcodes. This implies that the detection of longer program runs relies on the complexity of the program structure. While the maximum detection rate with the lowest false negative is achieved for the longest program run-lengths, a good detection rate with an acceptable false negative rate is obtained with ‘logic and arithmetic’ opcodes with a program run length of 1 K. This is consistent with present understanding of malware in that, in the early stages of start-up, it needs to unpack or decipher the executable code. Bilar *et al.* [3] show that the structure complexity of malware is less than that of non-malicious software, which has been borne out by the finding produced by the SVM.

(6) Fig. 5 presents the data in a format that – The opcodes used for the maximum detection are summed across all the different program run-lengths and weighted with their respective percentage detection rates. They are then normalised using (11).

$$O_n = \frac{1}{\text{peakValue}} \sum_{n=1}^{150} \boldsymbol{DR}(X_n) \qquad (11)$$

© The Institution of Engineering and Technology 2014

Table 4 Optimum opcodes for malware detection

Length (*k*)	Logical and arithmetic	Address manipulation	Flow control	Category score	Detection rate (%)
1	*add inc sub xor*	*pop*	*rep*	4,1,1 (6)	90
2	*xor*	*pop lods*	*nop cmp jb*	1,2,3 (6)	89
4	*sub*		*loopd jb*	1,0,2 (3)	81
8	*adc sub xor*		*rep je*	3,0,2 (5)	86
16	*xor*	*pop*	*rep cmp jb*	1,1,3 (5)	83
32	*add inc*	*movzx*	*cmp*	2,1,1 (4)	83
64	*adc*	*mov movzx*	*je*	1,2,1 (4)	84
128	*xor*	*mov*	*rep retn ja je jnb*	1,1,5 (7)	86
256	*dec inc*	*mov pop*	*cmp ja je*	2,2,2 (6)	87
512	*add sub*	*lea pop push*	*jmp rep je jnb test*	2,3,5(10)	89
1 024	*add dec*	*pop push movzx*	*call ja test*	2,3,3 (8)	90
2 048	*add adc sub inc*	*push*	*rep jb test*	4,1,3 (8)	90
4 096	*add*		*rep retn cmp jnb test*	1,0,5 (6)	90
8 192	*adc inc sub*	*pop*	*rep retn cmp ja*	3,1,4 (8)	92

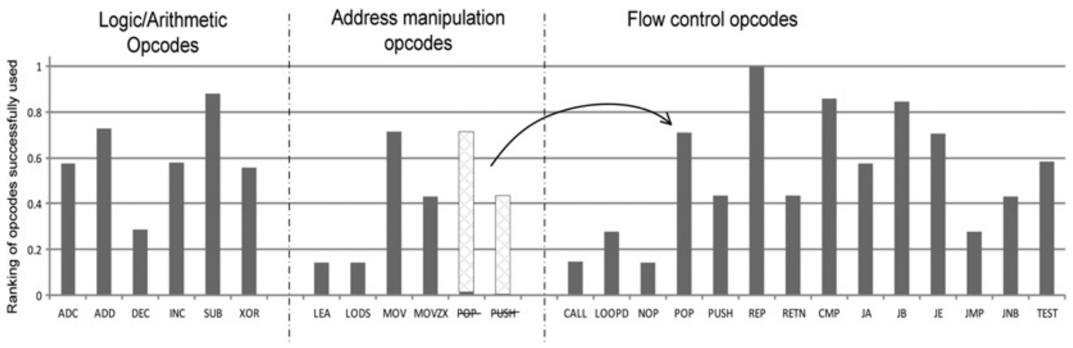

Fig. 5 *Opcodes grouped into functional categories*

where O is a normalised value that represent the success of that opcode in detecting malware.
peakValue is the maximum value found across O_{1-150} and is used to normalise the results;
n is an index used to scan through all 150 opcodes.

Grouping the opcode-weighted values (O_n) into their respective functional categories, which highlights the occurrence and detection ability of the various opcodes relating to their function category, that is, arithmetic, address manipulation and flow control. While push and pop opcodes are memory address manipulators they have been moved to the flow control section because they are frequently used to manage the stack whose primary function is to assist with program flow.

Fig. 5 shows that the most effective opcodes across all the program run-lengths are those opcodes that relate to the program flow control; *rep, cmp, jb, je* and *pop* have the most significant ability to classify malware. While there are opcodes from all the categories, flow control does present the largest selection of opcodes for detecting malware.

9 Conclusion

This paper proposes the use of a SVM as a means of identifying malware. It shows that detection of packed or encrypted malware is possible using a SVM for various program run-lengths. The dataset is created from program run-time traces, which is marshalled into density histograms and then filtered prior to being processed by a SVM to identify a subset of opcodes used to detect malware. The method used in this paper investigates the full spectrum of opcodes. While much existing research uses static analysis, the approach presented in this paper uses dynamic analysis and therefore evaluates actual execution paths, as opposed to evaluating all possible paths of executions as is normally the case in static analysis.

The results presented in this paper identify three key points. First, the optimum opcodes for detecting malware are not constant over different program run-lengths. Detection of shorter program run-lengths relies on arithmetic opcodes, while longer program runs rely on flow control opcodes. Secondly, opcodes relating to program flow control present some of the strongest indicators of malware with the exception of short run programs. Finally, the main contribution of this research is the identification of an optimum program run length. This paper has demonstrated that running malware to completion ensures the best detection rates. However, malware can be detected without running the program to completion and a short program run length of 1 k can yield a detection rate of 90% with a false negative rate of 4.6%. The medium program run lengths presented a good detection rate, however, the associated false negative rates are unacceptable. The detection characteristics of a 1 K program run length could enable the run-time detection of malware before it has had the chance to carry out any malicious activity i.e. at the point when it has unpacked/deciphered but not yet run.

This research relied on fixed program run-lengths using a delimiter defined by the number of opcodes, that is, 1 k, 2 k, 4 k etc. Further research is required to determine if a flexible delimiter can be found such as: a unique opcode sequence, OS call or the first suspicious activity detected.

10 Acknowledgment

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 20120006492).

11 References

1 Lakhotia, A., Uday Kumar, E., Venable, M.: 'A method for detecting obfuscated calls in malicious binaries', *IEEE Trans. Softw. Eng.*, 2005, **31**, (11), pp. 955–968

2 Bilar, D.: 'Opcodes as predictor for malware', *Int. J. Electron. Secur. Digit. Forensics*, 2007, **1**, (2), pp. 156–168

3 Bilar, D.: 'Callgraph properties of executables and generative mechanisms', *AI Commun. Special Issue Netw. Anal. Nat. Sci. Eng.*, 2007, **20**, (4), pp. 231–243

4 Ye, Y., Wang, D., Li, T., Ye, D.: 'IMDS: intelligent malware detection system'. Proc. 13th ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining, 2007, pp. 1043–1047

5 Santos, I., Penya, Y., Devesa, J., Bringas, P.: 'N-Grams-based file signatures for malware detection'. Proc. 11th Int. Conf. Enterprise Information Systems (ICEIS), Volume AIDSS, 2009, pp. 317–320

6 Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: 'A fast automaton-based method for detecting anomalous program behaviors', in: Needham, R., Abadi, M.(eds.): 'Proc. 2001 IEEE Symp. Security and Privacy' (IEEE Computer Society, Los Alamitos, CA, 2001), pp. 144–155

7 Li, W., Wang, K., Stolfo, S., Herzog, B.: 'Fileprints: identifying file types by n-gram analysis'. Sixth IEEE Information Assurance Workshop, June 2005, pp. 64–71

8 Santos, I., Brezo, F., Nieves, J., *et al.*: 'Opcode-sequence-based malware detection'. Proc. Second Int. Symp. Engineering Secure Software and Systems (ESSoS), Pisa (Italy), 3–4th February 2010, LNCS 5965, pp. 35–43

9 Santos, I., Brezo, F., Sanz, B., Laorden, C., Bringas, P.G.: 'Using opcode sequences in single-class learning to detect unknown malware', *IET Inf. Secur.*, 2011, **5**, (4), pp. 220–227

10 Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: 'Opcode sequences as representation of executables for data-mining-based unknown malware detection', *Inf. Sci.*, 2011, **231**, 64–82

11 Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y.: 'Detecting unknown malicious code by applying classification techniques on opcode patterns', *Secur. Inf.*, 2012, **1**, pp. 1–22

12 Moskovitch, R., Feher, C., Tzachar, N., *et al.*: 'Unknown malcode detection using opcode representation'. Proc. 1st European Conf. Intelligence and Security Informatics (EuroISI08), 2008, pp. 204–215

13 Song, Y., Locasto, M., Stavro, A.: 'On the infeasibility of modeling polymorphic shellcode'. ACM CCS, 2007, pp. 541–551

14 Kolbitsch, C., Milani Comparetti, P., Kruegel, C., Kirda, E., Zhou, X., Wang, X.: 'Effective and efficient malware detection at the end host'. 18th Usenix Security Symp., 2009, pp. 351–366

15 Eilam, E.: 'REVERSING secrets of reverse engineering' (Wiley Publishing Inc., 2005), ISBN-10:0764574817, pp. 109–122

16 Younge, A.J., Henschel, R., Brown, J.T., von Laszewski, G., Qiu, J., Fox, G.C.: 'Analysis of virtualization technologies for high performance computing environments'. IEEE Fourth Int. Conf. Cloud Computing, 2011, pp. 9–16

17 TUTS4YOU: 'OllyDbg 1.x.xplugins', http://thelegendofrandom.com/blog/archives/63, Version: StrongOD 0.4.8.892, lasted access 17 July 2013

18 Ferrie, P.: 'The ultimate anti debugge reference', http://pferrie.host22.com/papers/antidebug.pdf, Written May 2011, last accessed 11 October 2012

19 Chen, X., Andersen, J., Mao, Z.M., Bailey, M., Nazario, J.: 'Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware'. ICDSN Proc., 2008, pp. 177–186

20 Schölkopf, B., Smola, A., Müller, K.-R.: 'Kernel principal component analysis', *Artif. Neural Netw.—ICANN'97 Lect. Notes Comput. Sci.*, 1997, **1327**, pp. 583–588

21 Boser, B.E., Guyon, I.M., Vapnik, V.N.: 'A training algorithm for optimal margin classifiers', in Haussler, D. (ed.): 'Fifth Annual ACM Workshop on COLT' (ACM Press, Pittsburgh, PA, 1992), pp. 144–152

22 Hsu, C.-W., Chang, C.-C., Lin, C.-J.: '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

23 Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: 'Zozzle: low-overhead mostly static javascript malware detection'. Proc. Usenix Security Symposium, August 2011

24 Dahl, G., Stokes, J.W., Deng, L., Yu, D.: 'Large-scale malware classification using random projections and neural networks' (IEEE Signal Processing Society, Vancouver Canada, 2013), Poster (MLSP-P5.4), May ICASSP 2013

© The Institution of Engineering and Technology 2014

View publication stats