www.ietdl.org

Published in IET Information Security
Received on 23rd July 2010
Revised on 29th June 2011
doi: 10.1049/iet-ifs.2010.0180

ISSN 1751-8709

Using opcode sequences in single-class learning to detect unknown malware

I. Santos F. Brezo B. Sanz C. Laorden P.G. Bringas

DeustoTech – University of Deusto, Laboratory for Smartness, Semantics and Security (S^3Lab), Avenida de las Universidades 24, 48007 Bilbao, Spain
E-mail: isantos@deusto.es

Abstract: Malware is any type of malicious code that has the potential to harm a computer or network. The volume of malware is growing at a faster rate every year and poses a serious global security threat. Although signature-based detection is the most widespread method used in commercial antivirus programs, it consistently fails to detect new malware. Supervised machine-learning models have been used to address this issue. However, the use of supervised learning is limited because it needs a large amount of malicious code and benign software to be labelled first. In this study, the authors propose a new method that uses single-class learning to detect unknown malware families. This method is based on examining the frequencies of the appearance of opcode sequences to build a machine-learning classifier using only one set of labelled instances within a specific class of either malware or legitimate software. The authors performed an empirical study that shows that this method can reduce the effort of labelling software while maintaining high accuracy.

1 Introduction

Malware is computer software designed to damage computers. In the past, fame or glory were the main goals for malware writers, but nowadays, the reasons have evolved mostly into economical matters [1]. However, there are several exceptions to this general trend like the recent malware ‘Stuxnet’, which spies SCADA systems within industrial environments and reprograms them [2].

Commercial anti-malware solutions base their detection systems on signature databases [3]. A signature is a sequence of bytes always present within malicious executables together with the files already infected by that malware. The main problem of such an approach is that specialists have to wait until the new malware has damaged several computers to generate a file signature and thereby provide a suitable solution for that specific malware. Suspect files subject to analysis are compared with the list of signatures. When a match is found, the file being tested is flagged as malware. Although this approach has been demonstrated to be effective against threats that are known beforehand, signature methods cannot cope with code obfuscation, previously unseen malware or large amounts of new malware [4].

Two approaches exist that can deal with unknown malware that the classic signature method cannot handle, namely, anomaly detectors and machine-learning-based detectors. Regarding the way information is retrieved, there are two malware analysis approaches: static analysis which is performed without executing the file and dynamic analysis which implies running the sample in an isolated and controlled environment monitoring its behaviour.

Anomaly detectors retrieve significant information from non-malicious software and use it to obtain benign behaviour profiles. Every significant deviation from such profiles is flagged as suspicious. Li *et al.* [5] proposed a static fileprint (or n-gram) analysis in which a model or set of models attempt to construct several file types within a system based on their structural (i.e. byte) composition. This approach bases analysis on the assumption that non-malicious code is composed of predictably regular byte structures. On a similar vein, Cai *et al.* [6] employed static byte sequence frequencies to detect malware by applying a Gaussian likelihood model fitted with principal component analysis [7]. Dynamic anomaly detectors have been also proposed by the research community. For instance, Milenković *et al.* [8] employed a technique that guaranteed only secure instructions were actually executed in the system. The system employed signatures that were verified in execution time. Masri and Podgurski [9] described dynamic information flow analysis which worked as a specification system. The system was designed only for Java applications. Unfortunately, these methods usually show high false positive rates (i.e. benign software is incorrectly classified as malware), which presents difficulties for their adoption by commercial antivirus vendors.

Machine-learning-based approaches build classification tools that detect malware in the wild (i.e. undocumented malware) by relying on datasets composed of several characteristic features of both malicious samples and benign software. Schultz *et al.* [10] were the first to introduce the concept of applying data-mining models to the detection of malware based on respective binary codes. Specifically, they applied several classifiers to three different feature extraction approaches, namely, program headers, string features and byte sequence features. Subsequently, Kolter and Maloof [11] improved the results obtained by Schultz *et al.* by applying n-grams (i.e. overlapping byte sequences) instead of non-overlapping sequences. The method employed several algorithms to achieve optimal results using a Boosted (Boosting is machine-learning technique that builds a strong classifier composed by weak classifiers [12]) Decision Tree. Similarly, substantial research has focused on n-gram distributions of byte sequences and data mining [13–16]. Additionally, opcode sequences have recently been introduced as an alternative to byte n-grams [4, 17, 18]. This approach appears to be theoretically better because it relies on source code rather than the bytes of a binary file [19] (for a more detailed review of static features for machine-learning unknown malware detection refer to [20].

There are also machine-learning approaches that employ a dynamic analysis to train the classifiers. Rieck *et al.* [21] proposed the use of machine learning for both variant and unknown malware detection. The system employed application programming interface (API) calls to train the classifiers. On a similar vein, Devesa *et al.* [22] employed a sandbox to monitor the behaviour of an executable and vectors containing the binary occurrences of several specific behaviours (mostly dangerous system calls) were extracted and used to train several classic machine-learning methods. Recently, machine-learning approaches have been used for a complete system that includes early detection, alert and response [23].

Machine-learning classifiers require a high number of labelled executables for each of the classes (i.e. malware and benign). Furthermore, it is quite difficult to obtain this amount of labelled data in the real-world environment in which malicious code analysis would take place. To generate these data, a time-consuming process of analysis is mandatory, and even so, some malicious executables can avoid detection. Within machine-learning analysis, several approaches have been proposed to address this issue.

Semi-supervised learning is a type of machine-learning technique that is especially useful when a limited amount of labelled data exist for each class. These techniques train a supervised classifier based on labelled data and predict the label for unlabelled instances. The instances with classes that have been predicted within a certain threshold of confidence are added to the labelled dataset. The process is repeated until certain conditions are satisfied; one commonly used criterion is the maximum likelihood from the expectation-maximisation technique [24]. These approaches improve the accuracy of fully unsupervised (i.e. no labels within the dataset) methods [25]. However, semi-supervised approaches require a minimal amount of labelled data for each class; therefore they cannot be applied in domains in which only the instances belonging to a class are labelled (e.g. malicious code).

Datasets of labelled instances for only a single class are known as 'partially labelled datasets' [26]. The class that has labelled instances is known as the 'positive class' [27]. Building classifiers using this type of dataset is known as single-class learning [28] or learning from positive and unlabelled data.

With this background in mind, we propose the adoption of single-class learning for the detection of unknown malware based on opcode sequences. As the amount of malware is growing faster every year, the task of labelling malware is becoming harder, and approaches that do not require all data to be labelled are thus needed. Therefore we studied the potential of a two-step single-class learner called Roc-SVM (Rocchio-support vector machines) [26], which has already been used for text categorisation problems [26], for unknown malware detection. The main contributions of our study are as follows:

- We describe how to adopt Roc-SVM for unknown malware detection.
- We investigate whether it is better to label malicious or benign software.
- We study the optimal number of labelled instances and how it affects the final accuracy of models.
- We show that labelling efforts can be reduced in the anti-malware industry by maintaining a high rate of accuracy.

The remainder of this paper is organised as follows. Section 2 provides background regarding the representation of executables based on opcode-sequence frequencies. Section 3 describes the Roc-SVM method and how it can be adopted for unknown malware detection. Section 4 describes the experiments performed and presents the results. Section 5 discusses the obtained results and their implications for the anti-malware industry. Finally, Section 6 concludes the paper and outlines avenues for future work.

2 Opcode-sequence features for malware detection

To represent executables using opcodes, we extracted the opcode sequences and their frequency of appearance. More specifically, a program ρ may be defined as a sequence of instructions I, where $\rho = (I_1, I_2, \ldots, I_{n-1}, I_n)$. An instruction is a two-tuple composed of an operational code and a parameter or a list of parameters. As opcodes are significant by themselves [29], we discard the parameters and assume that the program is composed of only opcodes. These opcodes are gathered into several blocks that we call opcode sequences.

Specifically, we define a program ρ as a set of ordered opcodes o, $\rho = (o_1, o_2, o_3, o_4, \ldots, o_{n-1}, o_n)$, where n is the number of instructions I of a program ρ. An opcode sequence os is defined as an ordered subgroup of opcodes within the executable file, where os $\subseteq \rho$. It is made up of ordered opcodes o and os $= (o_1, o_2, o_3, o_4, \ldots, o_{m-1}, o_m)$, where m is the length of the sequence of opcodes os. We used the NewBasic Assembler (http://www.frontiernet.net/fys/newbasic.htm) as the tool for obtaining the assembly files in order to extract the opcode sequences of the executables.

Consider an example based on the assembly code snippet shown in Fig. 1. The following sequences of length 2 can be generated: $s_1 = (\text{mov}, \text{add})$, $s_2 = (\text{add}, \text{push})$, $s_3 = (\text{push}, \text{add})$, $s_4 = (\text{add}, \text{and})$, $s_5 = (\text{and}, \text{push})$,

```
mov    ax,0000h
add    [0BA1Fh],cl
push   cs
add    [si+0CD09h],dh
and    [bx+si+4C01h],di
push   sp
push   7369h
and    [bx+si+72h],dh
```

Fig. 1 *Assembly code example*

$s_6 = (\text{push, push})$ and $s_7 = (\text{push, and})$. As most of the common operations that can be used for malicious purposes require more than one machine code operation, we propose the use of sequences of opcodes instead of individual opcodes. As adding syntactical information with opcode sequences, we aim at identifying better the blocks of instructions (i.e. opcode sequences) that pass on the malicious behaviour to an executable.

We used this approach to choose the lengths of the opcode sequences. Nevertheless, it is hard to establish an optimal value for the lengths of the sequences; a small value will fail to detect complex malicious blocks of operations whereas long sequences can easily be avoided with simple obfuscation techniques.

We use 'term frequency inverse document frequency' (tf·idf) [30] to obtain the weight of each opcode sequence; the weight of the ith n-gram in the jth executable, denoted by weight(i, j), is defined by

$$\text{weight}(i, j) = \text{tf}_{i,j}\,\text{idf}_i \tag{1}$$

Note that term frequency $\text{tf}_{i,j}$ [30] is defined as

$$\text{tf}_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

Note that $n_{i,j}$ is the number of times the sequence $s_{i,j}$ (in our case an opcode sequence) appears in an executable e, and $\sum_k n_{k,j}$ is the total number of terms in the executable e (in our case the total number of possible opcode sequences).

We compute this measure for every possible opcode sequence of fixed length n, thereby acquiring a vector $\boldsymbol{v}$ of the frequencies of opcode sequences $s_i = (o_1, o_2, o_3, \ldots, o_{n-1}, o_n)$. We weight the frequency of occurrence of this opcode sequence using inverse document frequency idf_i is defined as

$$\text{idf}_i = \frac{|\mathcal{E}|}{|\mathcal{E}: t_i \in e|} \tag{3}$$

$|\mathcal{E}|$ is the total number of executables and $|\mathcal{E}: t_i \in e|$ is the number of documents containing the opcode sequence t_i.

Finally, we obtain a vector $\boldsymbol{v}$ composed of opcode-sequence frequencies, $\boldsymbol{v} = ((\text{os}_1, \text{weight}_1), (\text{os}_2, \text{weight}_2), \ldots, (\text{os}_{m-1}, \text{weight}_{m-1}), (\text{os}_m, \text{weight}_m))$, where os_i is the opcode sequence and weight_i is the tf·idf for that particular opcode sequence.

3 Roc-SVM method for learning from partially labelled data

Roc-SVM [26] is based on a combination of the Rocchio method [31] and SVM [32]. The method utilises the Rocchio method to select some significant negative instances belonging to the unlabelled class; SVM is then applied iteratively to generate several classifiers and then to select one of them.

For the first step (shown in Fig. 2) the method assumes that the entire unlabelled dataset $\mathcal{U}$ is composed of negative instances and then uses the positive set $\mathcal{P}$ together with $\mathcal{U}$ as the training data to generate a Rocchio classifier. We configured $\alpha = 16$ and $\beta = 4$ as recommended in [26, 33].

The model is then employed to predict the class of instances within $\mathcal{U}$. For the prediction, each test instance $e \in \mathcal{U}$ is compared with each prototype vector $e \in \mathcal{P}$ using

```
// Assign the unlabelled set $\mathcal{U}$ the negative class, and the
   positive set $\mathcal{P}$ the positive class
foreach $e \in \mathcal{U}$ do
    $e \leftarrow \vec{c}^-$;
foreach $e \in \mathcal{P}$ do
    $e \leftarrow \vec{c}^+$;
$\vec{c}^+ \leftarrow \alpha \frac{1}{|\mathcal{P}|} \sum_{e \in \mathcal{P}} \frac{\vec{e}}{||\vec{e}||} - \beta \frac{1}{|\mathcal{U}|} \sum_{e \in \mathcal{U}} \frac{\vec{e}}{||\vec{e}||}$ ;
$\vec{c}^- \leftarrow \alpha \frac{1}{|\mathcal{U}|} \sum_{e \in \mathcal{U}} \frac{\vec{e}}{||\vec{e}||} - \beta \frac{1}{|\mathcal{P}|} \sum_{e \in \mathcal{P}} \frac{\vec{e}}{||\vec{e}||}$;
foreach $e \in \mathcal{U}$ do
    if $sim(\vec{c}^+, \vec{e}) \leq sim(\vec{c}^-, \vec{e})$ then
        $N \leftarrow N \cup e$ ;
```

Fig. 2 *Rocchio selection of negative instances from $\mathcal{U}$ to $\mathcal{N}$*

the cosine measure [34]. The instances that are classified as negative are considered significant negative data and are denoted by $\mathcal{N}$.

In the second step (shown in Fig. 3), Roc-SVM trains and tests several SVMs [26] iteratively and then selects a final classifier. The SVM algorithms divide the n-dimensional spatial representation of the data into two regions using a hyperplane. This hyperplane always maximises the margin between the two regions or classes. The margin is defined by the longest distance between the examples of the two classes and is computed based on the distance between the closest instances of both classes, which are called supporting vectors [32]. The selection of the final classifier is determined by the amount of positive examples in $\mathcal{P}$ which are classified as negative. In [27] they define that if more than 8% of the positive documents are classified as

```
// Assign the set $\mathcal{N}$ the negative class, and the positive
   set $\mathcal{P}$ the positive class
foreach $e \in \mathcal{N}$ do
    $e \leftarrow \vec{c}^-$;
foreach $e \in \mathcal{P}$ do
    $e \leftarrow \vec{c}^+$;
repeat
    // Use $\mathcal{P}$ and $\mathcal{U}$ to train a SVM classifier $S_i$
    $S_i \leftarrow train(\mathcal{P} \cup \mathcal{N})$;
    // Classify $\mathcal{Q}$ using $S_i$.  The set of instances in $Q$ that
       are classified as negative is denoted by $W$
    $\mathcal{Q}' \leftarrow Classify(\mathcal{Q})$;
    $\mathcal{W} \leftarrow \{e \in \mathcal{Q}' : e = \vec{c}^-\}$;
    if $|\mathcal{W}| \neq 0$ then
        $\mathcal{Q} \leftarrow \mathcal{Q} - \mathcal{W}$;
        $\mathcal{N} \leftarrow \mathcal{N} \cup \mathcal{W}$;
until $|\mathcal{W}| = 0$;
// Use the last SVM classifier $S_1$ if > 8% positive are
   classified as negative or the first classifier $S_{last}$ in
   other cases, to classify $\mathcal{P}$
if $|\{e \in \mathcal{Q}' : e = \vec{c}^+\}| > \frac{8}{100}|\mathcal{Q}'|$ then
    Select $S_1$ as the final classifier;
else
    Select $S_{last}$ as the final classifier;
```

Fig. 3 *Generating the classifier*

negatives, SVM has been wrongly chosen, therefore S_1 is used. In other cases, S_{last} is employed. As they stated in [27], they used 8% because they wanted to be conservative enough not to select a weak last SVM classifier.

This generation is performed using the datasets $\mathcal{P}$ and $\mathcal{N}$. $\mathcal{Q}$ is the set of remaining unlabelled instances such that $\mathcal{Q} = \mathcal{U} - \mathcal{N}$.

4 Empirical study

The research questions we aimed to answer with this empirical study were as follows.

- What class (that is, malware or benign software) is of better use to label when using an opcode-sequence-based representation of executables?
- What is the minimum number of labelled instances required to assure suitable performance when using an opcode-sequence-based representation of executables?

To this end, we conformed a dataset comprising 1000 malicious executables and 1000 benign ones. For the malware, we gathered random samples from the website VX Heavens (http://vx.netlux.org/), which assembles a malware collection of more than 17 000 malicious programs, including 585 malware families that represent different types of current malware such as Trojan horses, viruses and worms. Since our method would not be able to detect packed executable, we removed any packed malware before selecting the 1000 malicious executables. Although they had already been labelled with their family and variant names, we analysed them using Eset Antivirus (http://www.eset.com/) to confirm this labelling.

This malware dataset contains executables coded with diverse purposes, as shown in Table 1, where backdoors, email worms and hacktools represent half of the whole malware population. The average file size is 299 KB, ranging from 4 to 5832 KB, representing the files smaller than 100 KB the 43.8% of the dataset, the files between 100 and 1000 KB the 49.6% and the files bigger than 1000 KB the final 6.6%.

These executables were compiled with very different generic compilers including Borland C++, Borland Delphi, Microsoft Visual C++, Microsoft Visual Basic and FreeBasic as it is shown in Table 2. Note that 44 of them were compiled with debugger versions and 70 were compiled with overlaying versions of the platforms shown in the table, whereas the other 886 were generated with standard versions of these compilers.

Table 1 Categorisation of the malware dataset depending on their functionality

Functionality	Number of instances
backdoor	305
Hacktool	130
email worm	124
email flooder	82
exploit	73
DOS	72
flooder	61
IM flooder	55
constructor	48
IRC worm	28
IM worm	16
net worm	6

Table 2 Categorisation of the malware dataset depending on the used compiler

Compiler	Instances per version	Total instances
Borland C++	19	56
Borland C + + 1999	36	–
Borland C + + DLL Method 2	1	–
Borland Delphi 2.0	12	183
Borland Delphi 3.0	25	–
Borland Delphi 4.0–5.0	76	–
Borland Delphi 6.0	4	–
Borland Delphi 6.0–7.0	66	–
FreeBasic 0.14	1	1
Microsoft VisualBasic 5.0	528	528
Microsoft Visual C++	4	232
Microsoft Visual C + + 4.x	14	–
Microsoft Visual C + + 5.0	42	–
Microsoft Visual C + + 6.0	154	–
Microsoft Visual C + + 7.0	15	–
Microsoft Visual C + + 8.0	3	–

For the benign dataset, we collected legitimate executables from our own computers. We also performed an analysis of the benign files using Eset Antivirus to confirm the correctness of their labels. In a previous work [18], a larger dataset was employed to validate the model.

This benign dataset is composed of different applications, such as installers or uninstallers, updating packages, tools of the Operating System, printer drivers, registry editing tools, browsers, PDF viewers, maintenance and performance tools, instant messaging applications, compilers, debuggers etc. The average file size is 222 KB, ranging from 4 to 5832 KB, representing the files smaller than 100 KB the 69.6% of the dataset, the files between 100 and 1000 KB the 25.4% and the files bigger than 1000 KB the final 5.0%.

Again, these executables were compiled with very different generic compilers like Borland C++, Borland Delphi, Dev-C++, Microsoft Visual C++, MingWin32 and Nullsoft; and two packers: ASProtect and UPX; as it is shown in Table 3. Note that 69 of them were compiled with debugger versions and 28 were compiled with overlaying versions of the already mentioned platforms, whereas 179 were generated with standard versions of the aforementioned compilers.

Using these datasets, we formed a total dataset of 2000 executables. In a previous work [18], a larger dataset was employed to validate the model. We did not use a larger training dataset because of technical limitations. However, the randomly selected dataset was heterogeneously enough to raise sound conclusions. In a further work, we would like to test how this technique scales with larger datasets.

Next, we extracted the opcode-sequence representations of opcode-sequence length $n = 2$ for every file in each dataset. The number of features obtained with an opcode length of two was very high at 144 598 features. To address this, we applied a feature selection step using Information Gain [35], selecting the top 1000 features. We selected 1000 features because it is a usual number to work with in text categorisation [36]. However, this value may change performance: a low number of features can decrease representativeness whereas a high number of features slows

Table 3 Categorisation of the benign dataset depending on the used compiler

Compiler	Instances per version	Total instances
ASProtect 2.1x	1	1
Borland C++	4	4
Borland Delphi 2.0	7	12
Borland Delphi 5.0	1	–
Borland Delphi 6.0	1	–
Borland Delphi Setup	3	–
Module	–	–
Dev-C++ 4.9.9.2	2	2
Microsoft Visual C++ 4.x	6	249
Microsoft Visual C++ 5.0	45	–
Microsoft Visual C++ 6.0	151	–
Microsoft Visual C++ 7.0	47	–
MingWin32 GCC 3.x	1	1
Nullsoft Install System 2.x	2	6
Nullsoft PiMP Stub	4	–
UPX 0.89.6–1.02	1	1
unknown	724	724

down the training step. The reason of not extracting further opcode-sequence lengths is that the underlying complexity of the feature selection step and the huge amount of features obtained would render the extraction very slow. Besides, an opcode-sequence length of 2 has proven to be the best configuration in a previous work [18].

We performed two different experiments. In the first experiment, we selected the positive and labelled class stored in $\mathcal{P}$ as malware, whereas in the second experiment, we selected the benign executables as the positive class. For both experiments, we split the dataset into ten subsets of training and test datasets using cross-validation [37]. In this way, we have the same training and test sets for both experiments. Later, we changed the number of labelled instances in the training datasets of each subset to 100, 200, 300, 400, 500, 600, 700, 800 and 900, taking into account which class is going to be the labelled one in each experiment. The unlabelled ones within the training set still belonged to the training set but their labels would be unknown until the first step of the algorithm finishes. In this way, we measure the effects of the number of labelled instances on the final performance of Roc-SVM's ability to detect unknown malware. In summary, we performed seven runs of Roc-SVM for each possible labelled class (malware or legitimate software) for each of the ten subsets in each experiment (malware or legimate software labelled). A summary of the construction of the different training and test datasets is shown in Fig. 4.

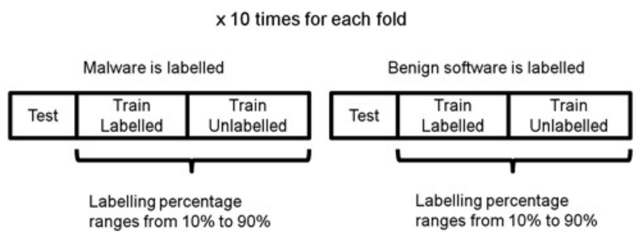
x 10 times for each fold
Malware is labelled
Test
Train Labelled
Train Unlabelled
Labelling percentage ranges from 10% to 90%
Benign software is labelled
Test
Train Labelled
Train Unlabelled
Labelling percentage ranges from 10% to 90%

Fig. 4 *Representation of the construction of training and test sets*

Ten different divisions of the data were performed of training and test set. For each division, two different types of labelled class was selected: malware and legitimate software. For each of them, the amount of labelled instances ranged from the 10 to 90%

To evaluate the results of Roc-SVM, we measured the precision of the malware (M_P) instances in each run, which is the amount of malware correctly classified divided by the amount of malware correctly classified and the number of legitimate executables misclassified as malware

$$M_P = \frac{TP}{TP + FP} \tag{4}$$

where TP is the number of true positives that is number of malware instances correctly classified and FP is the number of false positives that is number of legitimate executables misclassified as malware.

In addition, we measured the precision of the legitimate executables (L_P), which is the number of benign executables correctly classified divided by the number of legitimate executables correctly classified and the number of malicious executables misclassified as benign executables

$$L_P = \frac{TN}{TN + FN} \tag{5}$$

where TN is the number of legitimate executable correctly classified that is true negatives and FN, or false negatives, is the number of malicious executables incorrectly classified as benign software.

We also measured the recall of the malicious executables (M_R), which is the number of malicious executables correctly classified divided by the amount of malware correctly classified and the number of malicious executables misclassified as benign executables

$$M_R = \frac{TP}{TP + FN} \tag{6}$$

where TP is the number of true positives that is number of malware instances correctly classified and FN, or false negatives, is the number of malicious executables incorrectly classified as benign software. This measure is also known as false positive rate.

Next, we measured the recall of legitimate executables (L_R) in each run, which is the number of benign executables correctly classified divided by the number of legitimate executables correctly classified and the number of legitimate executables misclassified as malware

$$L_R = \frac{TN}{TN + FP} \tag{7}$$

where TN is the number of legitimate executable correctly classified that is true negatives and FP is the number of false positives that is number of legitimate executables misclassified as malware.

We also computed the F-measure, which is the harmonic mean of both the precision and recall

$$F - \text{measure} = 2 \times \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{8}$$

where Precision is the mean value between both malware and legitimate precision (M_P and L_P) and Recall is the mean value between both malware and legitimate recall (M_R and L_R).

Finally, we measured the accuracy of Roc-SVM, which is the number of the classifier's hits divided by the total

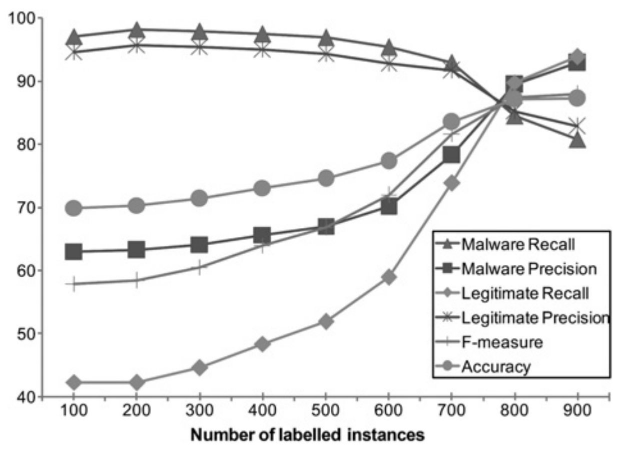

Fig. 5 *Results after labelling the class composed of malicious executables*

Overall results improve when the number of labelled malicious executables increases. Roc-SVM can guarantee an overall accuracy of 83.432% when 600 executables are labelled, which requires labelling 60% of the malware and 30% of the total corpus

number of classified instances

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} \quad (9)$$

Fig. 5 shows the results from selecting malware as the class for labelling. In this way, we can appreciate how the overall results improve when more malware executables are added. With regards to malware recall, when the size of the set of labelled instances increases, the rate of malware recall decreases. In other words, the more malicious executables are added to the labelled set, the less capable Roc-SVM is of detecting malware. Malware precision increases with the size of the labelled dataset, meaning that the confidence of Roc-SVM's detection of malware also increases. Legitimate precision decreases when the size of the labelled set increases, which indicates that more malicious executables are classified as benign software. However, legitimate recall increases, which shows that as the amount of labelled malware increases, so does the number of correctly classified instances of software. Both the F-measure and accuracy increase along with the size of the labelled dataset.

Fig. 6 shows the results when we select benign software as the labelled class. Overall, the results improve when more labelled executables are added. However, it only increases until 600 benign executables are labelled. Then, the classifier worsens. This indicates that too much legitimate software is redundant for the classifier. These general trends are very similar to the previous results. Malware recall decreases when the number of labelled instances increases. Malware precision increases with the size of the labelled dataset, and legitimate precision decreases when the size of the labelled set increases.

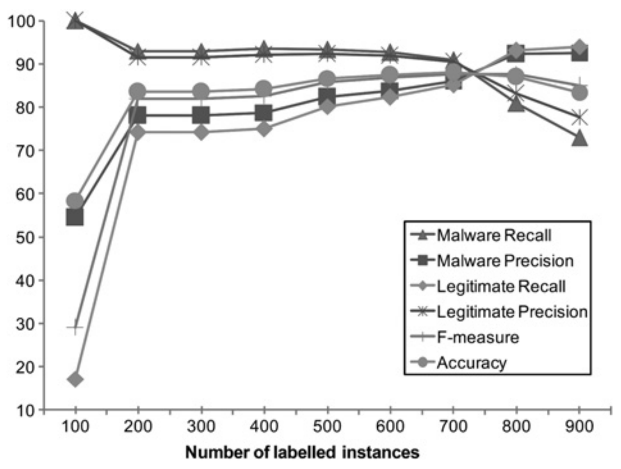

Fig. 6 *Results from labelling the class composed of benign executables*

Overall results improve when the number of labelled benign executables increases up to 600 labelled instances. After that, accuracy decreases. Roc-SVM can guarantee an overall accuracy of 84.221% when only 400 benign executables are labelled, which requires labelling 40% of the benign software and 20% of the total executable corpus. Labelling 600 benign executables obtains a higher accuracy of 87.456%

To compare the results obtained by Roc-SVM, we have defined two type of baselines: simple euclidean distance with malware labelled and the same distance measure with legitimate software labelled. For both baselines, we have used a ten-fold cross validation and the maximum amount of labelled software we have used to validate Roc-SVM: 900 instances. We have not used lower training set sizes because the results obtained with 900 instances, which will be the highest possible using this simple measure, are lower than the ones obtained with our single-class approach (as shown in Table 4). In order to provide a better distance measure we have weighted each feature with its information gain value with respect to the class.

Thereafter, we have measured the euclidean distance between the test dataset, composed of 100 malicious instances and 100 benign executable for each fold, the 900 training instances for each fold. In order to select the global deviation from the training set (that can be either malware or legitimate software) three combination rules were used: (i) the mean value, (ii) the lowest distance value and (iii) the highest value of the computed distances. Next, we

Table 4 Results for Euclidean distance using malware and legitimate software as training dataset for 900 labelled instances (the maximum amount tested in our single-class approach)

Approach	Acc., %	M_R, %	M_P, %	L_R, %	L_P, %	F-M, %
Euclidean with mean value and legitimate software	49.67	45.33	90.11	54.00	54.00	58.80
Euclidean with maximum value and legitimate software	68.60	83.61	94.34	53.60	100.00	80.43
Euclidean with minimum value and legitimate software	64.86	95.22	93.07	34.50	00.00	54.19
Euclidean with mean value and malicious software	71.60	67.10	73.74	76.10	76.10	73.22
Euclidean with maximum value and malicious software	59.35	19.40	96.51	99.30	100.00	74.01
Euclidean with minimum value and malicious software	79.50	90.50	74.18	68.50	00.00	50.58

Acc. stands for accuracy, M_R stands for malware recall, M_P stands for malware precision, L_R stands for legitimate recall, L_P stands for legitimate precision and F-M stands for f-measure have selected the threshold as the value with highest f-measure, selected from ten possible values between the value that minimised the false positives and the value that minimised the false negatives.

Table 4 shows the obtained results with Euclidean distance. The distance approach, although we have used the maximum number of training examples, obtained much worse results than the Roc-SVM approach proposed in this paper. Indeed, several configurations were as bad as a random classifier, showing that this simplistic approach is not feasible and that our single-class approach is far much better for classifying malware using the information of only one class of executables.

In summary, the obtained results show that it is better to label benign software rather than malware when we can only label a small number of benign executables. This results are in concordance with the work of Song *et al.* [38] regarding feasibility of blacklisting. However, if we can label a large amount of malware, the classifier would likely improve. The impact of the number of labelled instances is positive, enhancing the results when the size of the labelled dataset increases.

5 Discussion

We believe that our results will have a strong impact on the study of unknown malware detection, which usually relies on supervised machine learning. The use of supervised machine-learning algorithms for model training can be problematic because supervised learning requires that every instance in the dataset be properly labelled. This requirement means that a large amount of time is spent labelling. We have dealt with this problem using single-class learning that only needs a limited amount of a class (whether malware or benign) to be labelled. Our results outline the amount of labelled malware that is needed to assure a certain performance level in unknown malware detection. In particular, we found out that if we labelled 60% of the benign software, which is the 30% of the total corpus, the Roc-SVM method can achieve an accuracy and F-measure above 85%.

Although these results of accuracy are high, they may be not enough for an actual working environment. A solution to this problem is to employ user feedback and generate both blacklisting of the known malicious files and whitelisting of the confirmed benign applications.

It should also be interesting to evaluate how our method behaves chronologically in order to establish the importance of keeping updated the training set as suggested in [39], but we did not have an accurate information about the actual date each executable was retrieved. We would like to test this capability in a further work. On a similar vein, the imbalance problem has been introduced in previous work [13, 18]; basically it is stated that the balance of each class depends on the final results of a classifier. In our context, where we use a set of labelled instances and a set of unlabelled ones to train, an investigation of the effects in the balance between labelled and unlabelled instances is interesting as further work.

However, because of the static nature of the features we used with Roc-SVM, it cannot counter packed malware. Packed malware results from ciphering the payload of the executable and deciphering it when it finally loads into memory. Indeed, broadly used static detection methods can deal with packed malware only by using the signatures of the packers. As such, dynamic analysis seems like a more promising solution to this problem [40]. One solution to solve this obvious limitation of our malware detection method may involve the use of a generic dynamic unpacking schema, such as PolyUnpack [41], Renovo [40], OmniUnpack [42] and Eureka [43]. These methods execute the sample in a contained environment and extract the actual payload, allowing for further static or dynamic analysis of the executable. Another solution is to use concrete unpacking routines to recover the actual payload, but this method requires one routine per packing algorithm [44]. Obviously, this approach is limited to a fixed set of known packers. Likewise, commercial antivirus software also applies X-ray techniques that can defeat known compression schemes and weak encryption [45]. Nevertheless, these techniques cannot cope with the increasing use of packing techniques, and we thus suggest the use of dynamic unpacking schema to address this problem.

6 Conclusions

Unknown malware detection has become a research topic of great concern owing to the increasing growth in malicious code in recent years. In addition, it is well known that the classic signature methods employed by antivirus vendors are no longer completely effective against the large volume of new malware. Therefore signature methods must be complemented with more complex approaches that allow the detection of unknown malware families. Although machine-learning methods are a suitable solution for combating unknown malware, they require a high number of labelled executables for each of the classes under consideration (i.e. malware and benign datasets). As it is difficult to obtain this amount of labelled data in a real-word environment, a time-consuming analysis process is often mandatory.

In this paper, we propose the use of a single-class learning method for unknown malware detection based on opcode sequences. Single-class learning does not require a large amount of labelled data, as it only needs several instances that belong to a specific class to be labelled. Additionally, we found that it is more important to obtain labelled malware samples than benign software. By labelling 60% of the legitimate software, we can achieve results above 85% accuracy.

Future work will be oriented towards four main directions. First, we will use different features as data for training this kind of models. Second, we will focus on detecting packed executables using a hybrid dynamic–static approach. Third, we plan to perform a chronological evaluation of this method, where the update need of the training set will be determined. Fourth, we would like to investigate in the effect of the balance between labelled and unlabelled instances in single-class learning.

7 Acknowledgments

We would like to acknowledge to the anonymous reviewers for their helpful comments and suggestions.

8 References

1 Ollmann, G.: ‘The evolution of commercial malware development kits and colour-by-numbers custom malware’, *Comput. Fraud Secur.*, 2008, (9), pp. 4–7

2 Marks, P.: 'Stuxnet: the new face of war', *New Sci*, 2010, **208**, (2781), pp. 26–27

3 Morley, P.: 'Processing virus collections'. Proc. Virus Bulletin Conf., (VB2001), 2001, pp. 129–134

4 Santos, I., Brezo, F., Nieves, J., *et al.*: 'Idea: Opcode-sequence-based malware detection', in Massacci, F., Wallach, D.S., Zannone, N. (Eds.): 'Proceedings of the 2nd International Symposium on Engineering Secure Software and Systems', (*Lecture Notes in Computer Science*, vol. 5965, Springer), 2010, pp. 35–43

5 Li, W., Wang, K., Stolfo, S., Herzog, B.: 'Fileprints: identifying file types by n-gram analysis'. Proc. IEEE Workshop on Information Assurance and Security, 2005

6 Cai, D., Theiler, J., Gokhale, M.: 'Detecting a malicious executable without prior knowledge of its patterns'. Proc. Defense and Security Symp.. Information Assurance, and Data Network Security, 2005, vol. 5812, pp. 1–12

7 Jolliffe, I.: 'Principal component analysis' (Springer verlag, 2002)

8 Milenković, M., Milenković, A., Jovanov, E.: 'Using instruction block signatures to counter code injection attacks', *ACM SIGARCH Comput. Arch. News*, 2005, **33**, (1), pp. 108–117

9 Masri, W., Podgurski, A.: 'Using dynamic information flow analysis to detect attacks against applications'. Proc. Workshop on Software Engineering for Secure Systems Building Trustworthy Applications, 2005, pp. 1–7

10 Schultz, M., Eskin, E., Zadok, F., Stolfo, S.: 'Data mining methods for detection of new malicious executables'. Proc. 22nd IEEE Symp. on Security and Privacy, 2001, pp. 38–49

11 Kolter, J., Maloof, M.: 'Learning to detect malicious executables in the wild'. Proc. 10th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining, 2004, pp. 470–478

12 Schapire, R.: 'The boosting approach to machine learning: an overview'. Proc. MSRI Workshop on Nonlinear Estimation and Classification, 2001

13 Moskovitch, R., Stopel, D., Feher, C., Nissim, N., Elovici, Y.: 'Unknown malcode detection via text categorization and the imbalance problem'. Proc. Sixth IEEE Int. Conf. on Intelligence and Security Informatics (ISI), 2008, pp. 156–161

14 Shafiq, M., Khayam, S., Farooq, M.: 'Embedded malware detection using Markov n-Grams', *Lect. Notes Comput. Sci.*, 2008, **5137**, pp. 88–107

15 Zhou, Y., Inge, W.: 'Malware detection using adaptive data compression'. Proc. First ACM Workshop on AISec, 2008, pp. 53–60

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

17 Dolev, S., Tzachar, N.: 'Malware signature builder and detection for executable code'. EP patent 2,189,920, 26 May 2008

18 Moskovitch, R., Feher, C., Tzachar, N., *et al.*: 'Unknown malcode detection using opcode representation', *Intell. Secur. Informatics*, 2008, pp. 204–215

19 Christodorescu, M.: 'Behavior-based malware detection'. PhD thesis, 2007

20 Shabtai, A., Moskovitch, R., Elovici, Y., Glezer, C.: 'Detection of malicious code by applying machine learning classiers on static features: a state-of-the-art survey'. Information Security Technical report 14, 2009, no. 1, pp. 16–29

21 Rieck, K., Holz, T., Willems, C., D'ussel, P., Laskov, P.: 'Learning and classification of malware behavior'. Proc. Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA), 2008, pp. 108–125

22 Devesa, J., Santos, I., Cantero, X., Penya, Y.K., Bringas, P.G.: 'Automatic behaviour-based analysis and classification system for malware detection'. Proc. 12th Int. Conf. on Enterprise Information Systems (ICEIS), 2010

23 Shabtai, A., Potashnik, D., Fledel, Y., Moskovitch, R., Elovici, Y.: 'Monitoring, analysis, and filtering system for purifying network traffic of known and unknown malicious content', *Secur. Commun. Netw.*, 2010, **4**, (8), pp. 947–965

24 Zhu, X.: '*Semi-supervised learning* literature survey'. Technical report 1530, Computer Sciences, University of Wisconsin-Madison, 2005

25 Chapelle, O., Schölkopf, B., Zien, A.: 'Semi-supervised learning' (MIT Press, 2006)

26 Li, X., Liu, B.: 'Learning to classify texts using positive and unlabeled data'. Proc. Int. Joint Conf. on Artificial Intelligence (IJCAI), 2003, vol. 18, pp. 587–594

27 Liu, B., Dai, Y., Li, X., Lee, W., Yu, P.: 'Building text classifiers using positive and unlabeled examples'. Proc. Third IEEE Int. Conf. on Data Mining (ICDM), 2003, pp. 179–186

28 Wei, C., Chen, H., Cheng, T.: 'Effective spam filtering: a single-class learning and ensemble approach', *Decis. Support Syst.*, 2008, **45**, (3), pp. 491–503

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

30 Baeza-Yates, R.A., Ribeiro-Neto, B.: 'Modern information retrieval' (Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999)

31 Rocchio, J.: 'Relevance feedback in information retrieval. The SMART retrieval system: experiments in automatic document processing', 1971, pp. 313–323

32 Vapnik, V.: 'The nature of statistical learning theory' (Springer, 2000)

33 Buckley, C., Salton, G., Allan, J.: 'The effect of adding relevance information in a relevance feedback environment'. Proc. 17th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval, 1994, pp. 292–300

34 McGill, M., Salton, G.: 'Introduction to modern information retrieval' (McGraw-Hill, 1983)

35 Kent, J.: 'Information gain and a general measure of correlation', *Biometrika*, 1983, **70**, (1), p. 163

36 Forman, G.: 'An extensive empirical study of feature selection metrics for text classification', *J. Mach. Learn. Res.*, 2003, **3**, pp. 1289–1305

37 Bishop, C.: 'Pattern recognition and machine learning' (Springer, New York, 2006)

38 Song, Y., Locasto, M., Stavrou, A., Keromytis, A., Stolfo, S.: 'On the infeasibility of modeling polymorphic shellcode'. Proc. 14th ACM Conf. on Computer and Communications Security, 2007, pp. 541–551

39 Moskovitch, R., Elovici, Y.: 'Unknown malicious code detection–practical issues'. Proc. Seventh European Conf. on Information Warfare, 2008, pp. 145–153

40 Kang, M., Poosankam, P., Yin, H.: 'Renovo: a hidden code extractor for packed executables'. Proc. ACM Workshop on Recurring Malcode, 2007, pp. 46–53

41 Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: 'Polyunpack: automating the hidden-code extraction of unpack-executing malware'. Proc. 22nd Annual Computer Security Applications Conf (ACSAC), 2006, pp. 289–300

42 Martignoni, L., Christodorescu, M., Jha, S.: 'Omniunpack: fast, generic, and safe unpacking of malware'. Proc. 23rd Annual Computer Security Applications Conf. (ACSAC), 2007, pp. 431–441

43 Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., Lee, W.: 'Eureka: a framework for enabling static malware analysis'. Proc. European Symp. on Research in Computer Security (ESORICS), 2008, pp. 481–500

44 Ször, P.: 'The art of computer virus research and defense' (Addison-Wesley Professional, 2005)

45 Perriot, F., Ferrie, P.: 'Principles and practise of x-raying'. Proc. Virus Bulletin International Conf., 2004, pp. 51–66