Information Sciences 231 (2013) 64–82

Contents lists available at SciVerse ScienceDirect

Information Sciences

journal homepage: www.elsevier.com/locate/ins

Opcode sequences as representation of executables for data-mining-based unknown malware detection

Igor Santos *, Felix Brezo, Xabier Ugarte-Pedrero, Pablo G. Bringas

University of Deusto, Laboratory for Smartness, Semantics and Security (S³Lab), Avenida de las Universidades 24, 48007 Bilbao, Spain

ARTICLE INFO

Article history:
Available online 26 August 2011

Keywords:
Malware detection
Computer security
Data mining
Machine learning
Supervised learning

ABSTRACT

Malware can be defined as any type of malicious code that has the potential to harm a computer or network. The volume of malware is growing faster every year and poses a serious global security threat. Consequently, malware detection has become a critical topic in computer security. Currently, signature-based detection is the most widespread method used in commercial antivirus. In spite of the broad use of this method, it can detect malware only after the malicious executable has already caused damage and provided the malware is adequately documented. Therefore, the signature-based method consistently fails to detect new malware. In this paper, we propose a new method to detect unknown malware families. This model is based on the frequency of the appearance of opcode sequences. Furthermore, we describe a technique to mine the relevance of each opcode and assess the frequency of each opcode sequence. In addition, we provide empirical validation that this new method is capable of detecting unknown malware.

1. Introduction

Malware (or malicious software) is defined as computer software that has been explicitly designed to harm computers or networks [58]. In the past, malware creators were motivated mainly by fame or glory. Most current malware, however, is economically motivated [51].

Commercial anti-malware solutions rely on a signature database [49] (i.e., list of signatures) for detection. An example of a signature is a sequence of bytes that is always present within a malicious executable and within the files already infected by that malware. In order to determine a file signature for a new malicious executable and to devise a suitable solution for it, specialists must wait until the new malicious executable has damaged several computers or networks. In this way, suspect files can be analysed by comparing bytes with the list of signatures. If a match is found, the file under test will be identified as a malicious executable. This approach has proved to be effective when the threats are known beforehand.

Several issues render the signature-based method less than completely reliable: it cannot cope with *code obfuscations* and cannot detect *previously unseen malware*.

Malware writers use code obfuscation techniques [40] to hide the actual behaviour of their malicious creations [8,84,13,33]. Examples of these obfuscation algorithms include *garbage insertion*, which consists on adding sequences which do not modify the behaviour of the program (e.g., *nop* instructions[1]); *code reordering*, which changes the order of program instructions and *variable renaming*; which replaces a variable identifier with another one [15].

* Corresponding author.
E-mail addresses: isantos@deusto.es (I. Santos), felix.brezo@deusto.es (F. Brezo), xabier.ugarte@deusto.es (X. Ugarte-Pedrero), pablo.garcia.bringas@deusto.es (P.G. Bringas).

[1] Nop or Noop is an machine code instruction that does not perform any action.

0020-0255/$ - see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.ins.2011.08.020

Several approaches have been proposed by the research community to counter these obfuscation techniques. For instance, Sung et al. [74,82] introduced a method for computing the similarity between two executables by focusing on the degree of similarity within *syscall*[2] *sequences*. This approach offered only a limited performance because of its inability to detect malware while maintaining a low false positive ratio (i.e., benign executables misclassified as malware).

Several other approaches have been based on so-called *Control Flow Graph* (*CFG*) analysis. An example was introduced by Lo et al. [45] as part of the *Malicious Code Filter* (*MCF*) project. Their method slices a program into blocks while looking for tell-tale signs (e.g., operations that change the state of a program such as network access events and file operations) in order to determine whether an executable may be malicious.

Bergerson et al. [3] presented several methods for disassembling of binary executables, helping to build a representation of the execution of binary executables and improved the slicing of a program into *idioms* (i.e., sequences of instructions). It is also worth to mention the work of Christodorescu and Jha [16], who proposed a method based on CFG analysis to handle obfuscations in malicious software. Later, Christodorescu et al. [17] improved on this work by including semantic-templates of malicious specifications.

Two approaches have been developed to deal with unknown malware that classic signature method cannot handle: *anomaly detectors* and *data-mining-based detectors*. These approaches have been also used in similar domains like intrusion detection [39,23,77,27].

Anomaly detectors use information retrieved from benign software to obtain a *benign behaviour profile*. Then, every significant deviation from this profile is qualified as *suspicious*. Li et al. [42] proposed the *fileprint* (or n-gram) analysis in which a model or set of models attempt to characterise several file types on a system based on their structural (byte) composition. The main assumption behind this analysis is that benign files are composed of predictable regular byte structures for their respective types. Likewise, Cai et al. [9] used byte sequence frequencies detect malware. Their goal was to use only information about benign software to measure the deviations from the benign behaviour profile. Specifically, they applied a *Gaussian Likelihood Model* fitted with *Principal Component Analysis* (*PCA*) [31]. Unfortunately, these methods usually have a high false positive ratio that renders them difficult for commercial antivirus vendors to adopt.

Data-mining-based approaches rely on datasets that include several characteristic features for of both malicious samples and benign software to build classification tools that detect malware *in the wild* (i.e., undocumented malware). To this end, Schultz et al. [69] were the first to introduce the idea of applying data-mining models to the detection of different malicious programs based on their respective binary codes. Specifically, they applied several classifiers to three different feature extraction approaches: *program headers*, *string features* and *byte sequence features*. Later, Kolter et al. [36] improved the results obtained by Schulz et al. by using n-grams (i.e., overlapping byte sequences) instead of non-overlapping sequences. This method used several algorithms and the best results were achieved with a *Boosted*[3] *Decision tree*. In a similar vein, substantial research has focussed on n-gram distributions of byte sequences and data mining [50,71,85,67]. Still, most of these methods are limited as they count certain bytes in the malware body; because most of the common transformations operate at the source level, these detection methods can be easily thwarted [14]. These approaches were also used by Perdisci et al. [54] to detect packed executables. Perdisci et al. [54] proposed their first approach based on (i) the extraction of some features from the Portable Executable (PE), e.g., the number of standard and non-standard sections, the number of executable sections, the entropy of the PE header; and (ii) the classification through machine-learning models, e.g., Naïve Bayes, J48 and MLP. Later, Perdisci et al. [55] evolved their method and developed a fast statistical malware detection tool. They added new n-gram-based classifiers to combine their results with the previous MLP-based classifier.

Given the state of current research, we propose a new method for unknown malware detection using a data-mining-based approach. We use a representation based on *opcodes* (i.e., operational codes in machine language). Our method is based on the frequency of appearance of opcode-sequences: it trains several data-mining algorithms in order to detect unknown malware. A recent study [4] statistically analysed the ability of single opcodes to serve as the basis for malware detection and confirmed their high reliability for determining the maliciousness of executables. In a previous work [66], we presented an approach focused on detecting obfuscated malware variants using the frequency of appearance of opcode-sequences to build an information retrieval representation of executables. We now extend our research by focusing on the detection of unknown malware using data-mining techniques.

Specifically, we advance the state of the art with the following three contributions:

- We show how to use an opcode-sequence-frequency representation of executables to detect and classify malware.
- We provide empirical validation of our method with an extensive study of data-mining models for detecting and classifying unknown malicious software.
- We show that the proposed methods achieve high detection rates, even for completely new and previously unseen threats. We also discuss the shortcomings of the proposed model and suggest possible enhancements.

The remainder of this paper is organised as follows. Section 2 describes a method to calculate a weight of the relevance of each opcode. Section 3 details the method for the representation of executables. Section 4 outline the data-mining

[2] A syscall or system call is the procedure through which an executable requests a service from the kernel of the operating system.
[3] Boosting is a machine learning technique that builds a strong classifier composed by weak classifiers [68].

approaches used. Section 5 describes the experiments performed and present results. Section 6 discusses the proposed method. Finally, Section 7 concludes the paper and outlines avenues for future work.

2. Finding a measure for the ability of opcodes to detect malware

In a recent work Bilar [4] investigated the ability of operational codes to detect malware. This study concluded that opcodes reveal significant statistical differences between malware and legitimate software and that rare opcodes are better predictor than common opcodes. The operands in the machine code were omitted and the opcodes by themselves were capable to statistically explain the variability between malware and legitimate software.

Therefore, our approach only uses opcodes and we discard the operands within the instructions. First, we extend this previous study by providing a method that measures the relevance of individual opcodes. To this end, we employ a methodology that computes a weight for each operational code. This weighting represents the relevance of the opcode to malicious and benign executables based on whether it appears more frequently in malware or in benign executables.

We collected malware from the VxHeavens website[4] to assemble a malware dataset of 13,189 malware executables. This dataset contained only *Portable Executable* (PE)[5] executable files, and it was made up of different types of malicious software (e.g., computer viruses, Trojan horses and spyware). For the benign software dataset, we collected 13,000 executables from our computers. This benign dataset included text processors, drawing tools, windows games, Internet browsers and PDF viewers. We confirmed that the benign executables were not infected because any infections would have distorted the results. We achieved this via analysis of the benign files using *Eset Antivirus*.[6]

The method for computing the relevance of opcodes is composed of the following steps:

1. **Disassembling the executables:** We used the NewBasic Assembler[7] as the main tool for obtaining the assembly files.
2. **Generation of opcode profile file**: Using the generated assembly files, we built opcode profiles. Each file contains a list with the operational code and the times that each opcode appears within both the benign software dataset and the malicious software dataset.
3. **Computation of opcode relevance:** We computed the relevance of each opcode based on the frequency it appears in each dataset. We used *Mutual Information* [53] (shown in Eq. (1)) to measure the statistical dependence of the two variables:

$$I(X;Y) = \sum_{y\in Y}\sum_{x\in X} p(x,y)\log\left(\frac{p(x,y)}{p(x)\cdot p(y)}\right) \tag{1}$$

where X is the opcode frequency and Y is the class of the file (i.e., malware or benign software), $p(x,y)$ is the joint probability distribution function of X and Y, and $p(x)$ and $p(y)$ are the marginal probability distribution functions of X and Y. In our particular case, we defined the two variables as the single opcode and whether or not the instance was malware. Note that this weight only measures the relevance of a single opcode and not the relevance of an opcode sequence.

Once we had computed the mutual information between each opcode and the executable class, we sorted and generated an opcode relevance file. The opcode frequency file was saved so that we may calculate the relevance of the opcodes in future research using other measures such as the gain ratio [46] or chi-square [30].

This list of opcode relevances helped with more accurate malware detection because we were able to weight the final representation of executables using the calculated opcode relevances and reducing noise from irrelevant opcodes. Specifically, we observed that the most common opcodes, such as `push`, `mov` or `add`, tended to be weighted low in the results.

These weights may be considered a replacement for the Inverse Document Frequency (IDF) measure [62] used in the Vector Space Model [65] for information retrieval. The IDF weighting terms occur in documents based on the frequency with which they appear in the whole document. Our method performs a similar task by using mutual information instead of simply counting occurrences. Our method is also easier to update, using the total number of opcode occurrences in each dataset (both malicious and benign).

3. Feature extraction

To represent executables using opcodes, we extract 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 2-tuple composed of an operational code and a parameter or a list of parameters. Since opcodes are significant by themselves [4], we discard the parameters and assume that the program was composed only of opcodes. These opcodes are gathered into several blocks that we call opcode sequences.

[4] http://vx.netlux.org/.
[5] The Portable Executable (PE) format is a file format for executables, object code and DLLs, used in Microsoft Windows operating systems.
[6] http://www.eset.com/.
[7] http://www.frontiernet.net/fys/newbasic.htm.

```
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.

Specifically, we define a program ρ as a set of ordered opcodes o, $\rho = (o_1, o_2, o_3, o_4, \ldots, o_{\ell-1}, o_\ell)$, where ℓ is the number of instructions I of the program ρ. An opcode sequence os is defined as a subgroup of opcodes within the executable file where $os \subseteq \rho$; it is made up of opcodes o,$os = (o1, o2, o3, \ldots, o_{m1}, o_m)$ where m is the length of the sequence of opcodes os.

Consider an example based on the assembly code snippet shown Fig. 1; the following sequences of length 2 can be generated: $s_1 = (\texttt{mov}, \texttt{add})$, $s_2 = (\texttt{add}, \texttt{push})$, $s_3 = (\texttt{push}, \texttt{add})$, $s_4 = (\texttt{add}, \texttt{and})$, $s_5 = (\texttt{and}, \texttt{push})$, $s_6 = (\texttt{push}, \texttt{push})$ and $s_7 = (\texttt{push}, \texttt{and})$. Because 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. We added syntactical information by using this representation since we wished to better identify blocks of instructions (opcode sequences) that pass on malicious behaviour to an executable.

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. Besides, long opcode sequences will introduce a high performance overhead.

Afterwards, we compute the frequency of occurrence of each opcode sequence within the file by using *Term Frequency* (TF) [48] (shown in Eq. (2)) that is a weight widely used in information retrieval [83]:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{2}$$

where $n_{i,j}$ is the number of times the sequence $s_{i,j}$ (in our case 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 $\vec{v}$ of the frequencies of opcode sequences $s_i = (o_1, o_2, o_3, \ldots, o_{n-1}, o_n)$. We weighted the frequency of occurrence of this opcode sequence using the relevance weights described in Section 2.

We define the *Weighted Term Frequency* (WTF) as the result of weighting the TF with the relevance of each opcode when calculating the term frequency. Specifically, we computed the WTF as the product of sequence frequency and the calculated weight of every opcode in the sequence:

$$wtf_{i,j} = tf_{i,j} \cdot \prod_{o_z \epsilon S} \frac{weight(o_z)}{100} \tag{3}$$

where $weight(o_z)$ is the calculated weight, by means of mutual information gain, for the opcode o_z and $tf_{i,j}$ is the *sequence frequency measure* for the given opcode sequence.

Applying the previously calculated weighted sequence frequencies, we obtain a vector $\vec{v}$ composed of weighted opcode-sequence frequencies, $\vec{v} = ((os_1, wtf_1), (os_2, wtf_2), (os_3, wtf_3), \ldots, (os_{n-1}, wtf_{n-1}), (os_n, wtf_n))$, where os_i is the opcode sequence and wtf_i is the weighted term frequency for that particular opcode sequence. Using the resultant vector representation of the files, we are able to, one hand, compute a similarity between two input files to detect malware variants and on the other hand, train machine-learning classifiers to detect unknown malware.

4. Data-mining-based classification

Offering protection from unknown malware is an important challenge in malware detection due to the increasing growth of malware. Data mining approaches usually rely on *machine-learning* algorithms that use both malicious executables and benign software to detect malware in the wild [28,29,81,70].

Machine learning is a discipline within *Artificial Intelligence* (AI) concerned with the design and development of algorithms that allow computers to reason and make decisions based on data [5]. Generally, machine-learning algorithms can be classified into three different types: *supervised learning*, *unsupervised learning* and *semi-supervised learning* algorithms. First, supervised machine-learning algorithms, or classifying algorithms, require the training dataset to be properly labelled (in our case, knowing whether an instance is malware) [37]. Second, unsupervised machine-learning algorithms, or clustering algorithms, try to assess how data are organised into different groups called *clusters*. In this type of machine-learning,

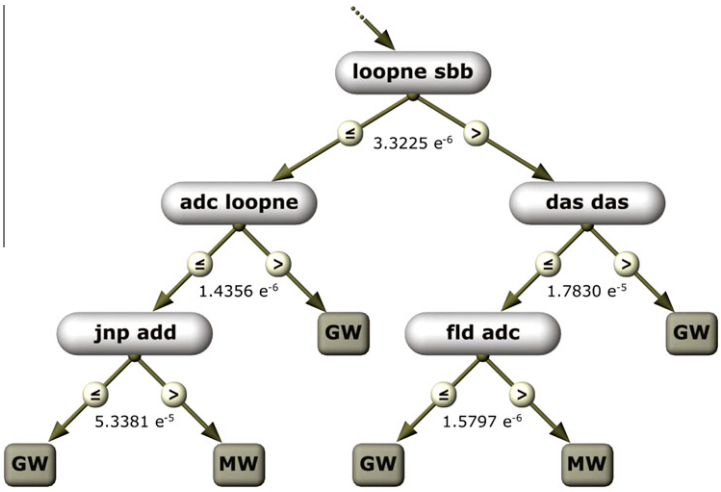

Fig. 2. Extract of the decision tree.

data do not need to be labelled [38]. Finally, semi-supervised machine-learning algorithms use a mixture of both labelled and unlabelled data in order to build models, thus improving the accuracy of unsupervised methods [12].

Since in our case malware can be properly labelled, we use supervised machine-learning. In future work, however, we would also like to test the ability of unsupervised methods to detect malware. In the remainder of this section, we review several supervised machine-learning approaches that have succeeded in similar domains [67,69,36].

4.1. Decision trees

Decision tree classifiers are a type of machine-learning classifiers that can be graphically represented as a tree (Fig. 2 shows an snippet of the actual decision tree generated). The internal nodes represent conditions of the problem variables, and their final nodes (or leaves) constitute the final decision of the algorithm [60]. In our case, the final nodes would represent whether an executable is malware or not.

Formally, a decision tree graph $G = (V,E)$ consists on a not empty set of finite nodes V and a set of arcs E. If the set of arcs is composed of ordered bi-tuples (v,w) of vertices, then we say that the graph is *directed*. A *path* is defined as an arc sequence of the form $(v_1, v_2),(v_2, v_3),\ldots,(v_{n-1}, v_n)$. Paths can be expressed by origin, end and distance (i.e., the minimum number of arcs from the origin to the end). In addition, if (v,w) is an arc within the tree, v is considered the *parent* of w. Otherwise, w is defined as the *child* node of v. The single node with no parent is defined as the *root node*. Every other node in the tree is defined as an *internal node*. In order to build the representation of the tree, a set of binary questions (yes–no) are answered.

There are several training algorithms that are typically used to learn the graph structure of these trees by means of a labelled dataset. In this work, we use *Random Forest*, which is an ensemble (i.e., combination of weak classifiers) of different randomly-built decision trees [7]. Further, we also use *J48*, the *Weka* [25] implementation of the *C4.5* algorithm [61].

4.2. Support Vector Machines (SVM)

SVM classifiers consist of a hyperplane dividing a n-dimensional-space-based representation of the data into two regions (shown in Fig. 3). This hyperplane is the one that maximises the *margin* between the two regions or classes (in our case, malware or benign software). Maximal margin is defined by the largest distance between the examples of the two classes computed from the distance between the closest instances of both classes (called *supporting vectors*) [80].

Formally, the optimal hyperplane is represented by a vector w and a scalar m in a way that the inner products of w with vectors $\phi(X_i)$ from the two classes are divided by an interval between -1 and +1 subject to m:

$$(w, \phi(X_i)) - m \geqslant +1 \tag{4}$$

for every X_i that belongs to the first class (in our case malware) and

$$(w, \phi(X_i)) - m \leqslant -1 \tag{5}$$

for every X_i that belongs to the second class (in our case benign software).

The optimisation problem that involves finding w and m can be formulated solely in terms of inner products $\phi(x_i),\phi(x_j)$ between data points when this problem is stated in the dual form from quadratic programming [5]. In mathematics, an inner product space is a vector space with the additional structure called an inner product. This structure relates each pair of

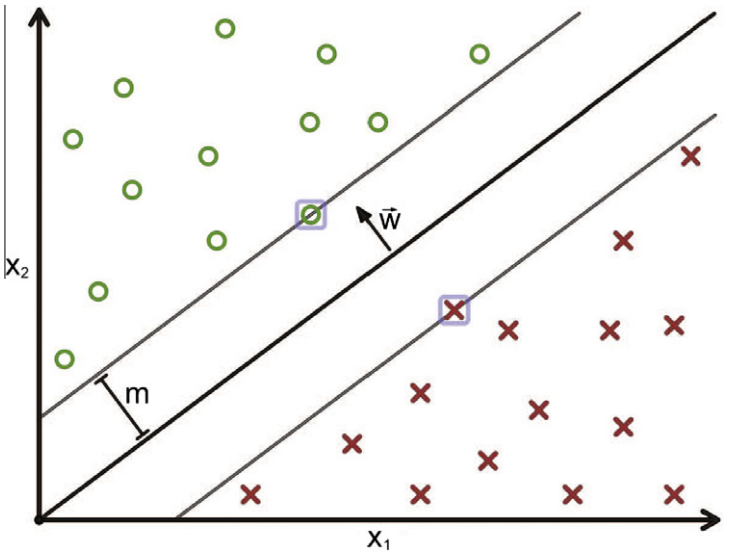

Fig. 3. Example of a SVM classifier in a bi-dimensional space.

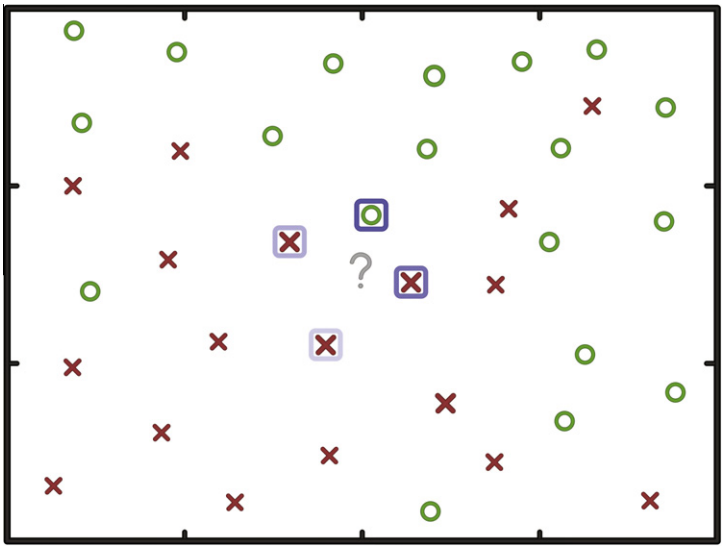

Fig. 4. Example of a KNN classifier in a bidimensional space.

vectors in the space with a scalar quantity known as the inner product of the vectors. A specific case of inner product is the well-known dot product between vectors.

A kernel is thus defined by $\alpha(\phi(X_i), \phi(X_j)) = x_i \cdot \phi(X_j)$ forming the linear kernel with inner products. Generally, instead of using inner products, the so-called *kernel functions* are applied. These kernel functions lead to non-linear classification surfaces, such as polynomial, radial or sigmoid surfaces [1].

4.3. K-Nearest Neighbours

The *K-Nearest Neighbour* (KNN) [24] algorithm is one of the simplest supervised machine-learning algorithms for classifying instances. This method classifies an unknown instance based on the class (in our case, malware or benign software) of the instance nearest to it in the training space (see Fig. 4).

Specifically, the training phase of this algorithm presents a set of training data instances $S = \{s_1, s_2, \ldots, s_m\}$ in a n-dimensional space where n is the number of variables for each instance (e.g., the frequency of occurence for each opcode sequence).

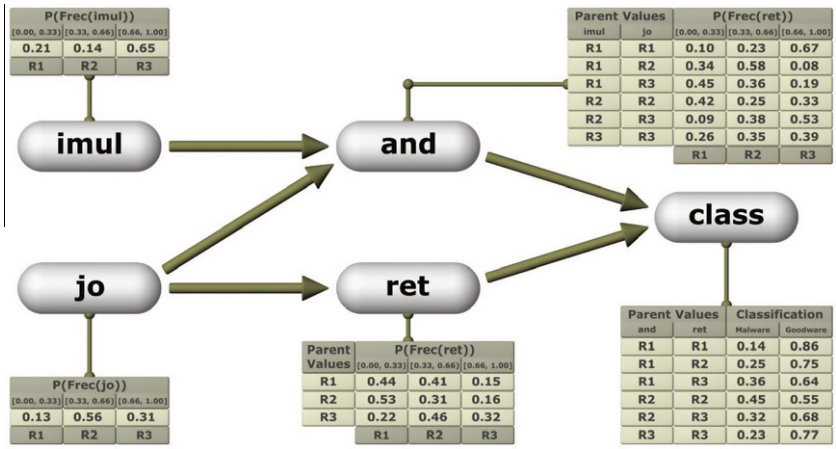

Fig. 5. Example of a Bayesian network representing the malware classification problem.

Furthermore, the classification phase of an unknown instance is conducted by measuring the distance between the training instances and the unknown specimen. In this way, establishing the distance between two points X and Y in a n-dimensional space can be achieved by using any distance measure. In our case, we use *Euclidean Distance*:

$$\sqrt{\sum_{i=0}^{n}(X_i - Y_i)^2} \tag{6}$$

There are several methods for determining the class of the unknown sample; the most commonly used technique is to classify the unknown instance as the most common class among its *K-Nearest Neighbours*.

4.4. Bayesian networks

Bayes' theorem [2] is the basis of the so-called Bayesian inference, a statistical reasoning method that determines, based on a number of observations, the probability that a hypothesis may be true. Bayes' theorem adjusts the probabilities as new informations becomes available. According to its classical formulation (shown in Eq. (7)), given two events A and B, the conditional probability $P(A|B)$ that A occurs if B occurs can be obtained if we know the probability that A occurs, $P(A)$, the probability that B occurs, $P(B)$, and the conditional probability of B given A, $P(B|A)$.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{7}$$

Bayesian networks [52] are probabilistic models for multivariate analysis. Formally, they are directed acyclic graphs associated with a probability distribution function [11]. Nodes in the graph represent variables (which can be either a premise or a conclusion) while the arcs represent conditional dependencies between such variables. The probability function illustrates the strength of these relationships in the graph [11]. Fig. 5 shows a representation of the actual problem as a simplified Bayesian network. Nodes in the graph are variables of the classification problem. Each node has a conditional probability function associated to it. One of the nodes (or several in other cases) is the output node, which in our case is whether an executable is malicious or not.

If we let a Bayesian network B be defined as a pair, $B = (D,P)$, where D is a directed acyclic graph, $P = \{p(x_1|\Psi_2),\ldots, p(x_n|\Psi_n)\}$ is the set composed by n conditional probability functions, one for each variable and Ψ_i is the set of parent nodes of the node X_i in D. The set P is defined as a *joint probability density function* (shown in Eq. (8)):

$$P(x) = \prod_{i=1}^{n} p(x_i|\Psi_i) \tag{8}$$

For our needs, the most important ability of Bayesian networks is their ability to infer the probability of a certain hypothesis being true (e.g., the probability of an executable being malicious or legitimate) given a historical dataset.

5. Experiments and results

In order to validate our proposed method, we used two different datasets to test the system: a malware dataset and a benign software dataset. We downloaded several malware samples from the VxHeavens website to assemble a malware

dataset of 17,000 malicious programs, including 585 malware families that represent different types of malware such as Trojan horses, viruses and worms. Among the malware programs used for our experiment, we use variants of SDBot, Alamar, Bropia, Snowdoor, JSGen, Kelvir, Skydance, Caznova, Sonic, Redhack and Theef.

Even though they had already been labelled with their family and variant names, we scanned them using *Eset Antivirus* to confirm this labelling. We also use PEiD[8] to analyse whether malware was packed. We removed any packed samples and selected a total of 1000 malicious executables. We confirmed that the remainder of the executables were not the result of compilation by many different compilers to avoid distortion of the results (a more detailed discussion is provided in Section 6).

For the benign dataset, we gathered 1000 legitimate executables from our computers. As we did when computing the relevance of opcodes in Section 2, we performed an analysis of the benign files using *Eset Antivirus*.

We extracted the opcode-sequence representation for every file in that dataset for different opcode-sequence lengths n. Specifically, we extracted features for $n = 1$ and $n = 2$. The reason of not extracting further opcode-sequence lengths is that the underlying complexity of machine-learning methods and the huge amount of features obtained would render the extraction very slow. We also used the two opcode-sequence lengths combined: a feature-set composed by the frequencies of occurrence with lengths of one and two.

The number of features obtained with an opcode-sequence length of two and above was very high (see Fig. 8). To deal with this, we applied a feature selection step using *Information Gain* [34] and we selected the top 1000 features, which represent the 0.6% of the total number of features in the case of $n = 2$. On we obtained the reduced datasets, we tested the suitability of our proposed approach using the following steps:

- **Cross validation:** In order to evaluate the performance of machine-learning classifiers, *k-fold cross validation* is usually used in machine-learning experiments [5].
  This technique assesses how the results of the predictive models will generalise to an independent data set. It involves partitioning the sample of data into subsets, performing the training step with one subset (called the training set) and validating with the remaining dataset (called the test set). To reduce the variability, cross validation performs multiple rounds with different partitions, which are defined with the parameter k. The results of each round are averaged to estimate the global measures of the tested model.
  Thereby, for each classifier we tested, we performed a k-fold cross validation [35] with $k = 10$. In this way, our dataset was split 10 times into 10 different sets of learning (90% of the total dataset) and testing (10% of the total data).
- **Learning the model:** For each validation step, we conducted the learning phase of each algorithm with each training dataset, applying different parameters or learning algorithms depending on the concrete classifier. The algorithms use the default parameters in the well-known machine-learning tool WEKA [25]. Specifically, we used the following four models:
  - *Decision trees* (*DT*): We used *Random Forest* [7] and *J48* (Weka's *C4.5* [61] implementation).
  - *K-Nearest Neighbour* (*KNN*): We performed experiments over the range $k = 1$ to $k = 10$ to train KNN.
  - *Bayesian networks* (*BN*): We used several structural learning algorithms; *K2* [18], *Hill Climber* [64] and *Tree Augmented Naïve* (TAN) [26]. We also performed experiments with a *Naïve Bayes* classifier [41].
  - *Support Vector Machines* (*SVM*): We used a *Sequential Minimal Optimization* (SMO) algorithm [57] and performed experiments with a *polynomial kernel* [1], a *normalised polynomial kernel* [1], *Pearson VII function-based* universal kernel [79], and a *Radial Basis Runction* (RBF) based kernel [1].
- **Testing the model:** In order to measure the processing overhead of the proposed model, we measure the required representation time, number of features, feature selection time, training time and testing time:
  - *Disassembling time:* It measures the time that the disassembler requires to extract the assembly representation of the files. For this step, we used an Intel Pentium M clocked at 1.73 Ghz with 1 GB of RAM memory because it was the only machine available with a $\times 32$ Operative System installed, which NewBasic Assembler requires.
  - *Representation time:* It measures the time required to build the representation of the executables. To this end, we tested with different opcode-sequence length ranked from 1 to 4. For this step, we used a Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory.
  - *Number of features*: It measures the number of different opcode sequences for each opcode-sequence length from 1 to 4. For this step, we used a Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory.
  - *Feature selection*: It measures the time required for the selection of the most relevant opcode sequences. We used a Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory.
  - *Training time*: It measures the needed overhead for building the different machine-learning algorithms. We used a Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory.
  - *Testing time*: It measures the total time that the models require for evaluating the testing instances in the dataset. We used a Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory.

To evaluate each classifier's capability, we measured the *True Positive Ratio* (TPR), i.e., the number of malware instances correctly detected, divided by the total number of malware files (shown in Eq. (9)):

[8] http://www.peid.info/.

$$TPR = \frac{TP}{TP + FN} \tag{9}$$

where *TP* is the number of malware cases correctly classified (true positives) and *FN* is the number of malware cases misclassified as legitimate software (false negatives).

We also measured the *False Positive Ratio* (FPR), i.e., the number of benign executables misclassified as malware divided by the total number of benign files (shown in Eq. (10)):

$$FPR = \frac{FP}{FP + TN} \tag{10}$$

where *FP* is the number of benign software cases incorrectly detected as malware and *TN* is the number of legitimate executables correctly classified.

TPR and FPR establish the cost of misclassification. It is important to set the cost of false negatives ($1 - TPR$) and false positives, in other words, establish whether is better to classify a malware as legitimate or to classify a benign software as malware. In particular, since our framework is devoted to detect new and unknown malware, one may think that it is more important to detect more malware than to minimise false positives. However, for commercial reasons, one may think just the opposite: a user can be bothered if their legitimate applications are flagged as malware. Therefore, we consider that the importance of the cost is established by the way our framework will be used. If it is used as a complement to standard anti-malware systems then we should focus on minimising false positives. Otherwise, if the framework is used within anti-virus laboratories to decide which executables should be further analysed then we should minimise false negatives (or maximise true positives). To tune the proposed method, we can apply two techniques: (i) whitelisting and blacklisting or (ii) cost-sensitive learning. White and black lists store a signature of an executable in order to be flagged either as malware (blacklisting) or benign software (whitelisting). On the other hand, cost-sensitive learning is a machine-learning technique where one can specify the cost of each error and the classifiers are trained taking into account that consideration [22].

Furthermore, we measured *accuracy*, i.e., the total number of the classifier's hits divided by the number of instances in the whole dataset (shown in Eq. (11)):

$$Accuracy(\%) = \frac{TP + TN}{TP + FP + TP + TN} \cdot 100 \tag{11}$$

Besides, we measured the *Area Under the ROC Curve* (AUC) that establishes the relation between false negatives and false positives [73]. The ROC (Receiver Operator Characteristics) curve is obtained by plotting the TPR against the FPR.

Fig. 6 shows the time required for NewBasic Assembler to extract the assembly representation from the binary executables. Obviously, the greater the size of the executable, the greater the required time for the disassembly step.

Fig. 7 shows the representation times for $n = 1$, $n = 2$, $n = 3$ and $n = 4$. This step utilises the assembly files generated by the NewBasic Assembler and builds the opcode-sequence representations. In particular, for a opcode-sequence length of 1, the average representation time was 149.31 ms; for $n = 2$ the average time was 182.12 ms; for $n = 3$ was 285.56 milliseconds and for $n = 4$ it was 1254.36 ms. Therefore, the required representation time is exponential to the opcode-sequence length while remains lineal with regards to the executable size.

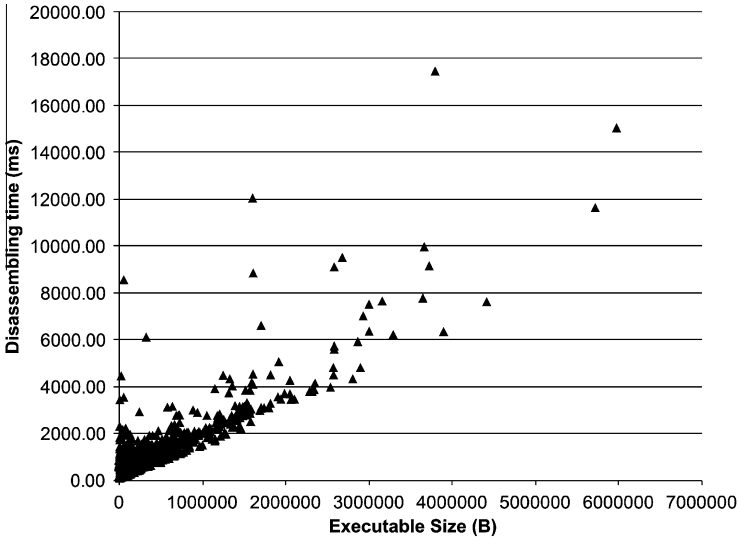

Fig. 6. Time results for extracting the assembly code from the binary. As we can see the required time increases along with the opcode-sequence length and the size of the executable.

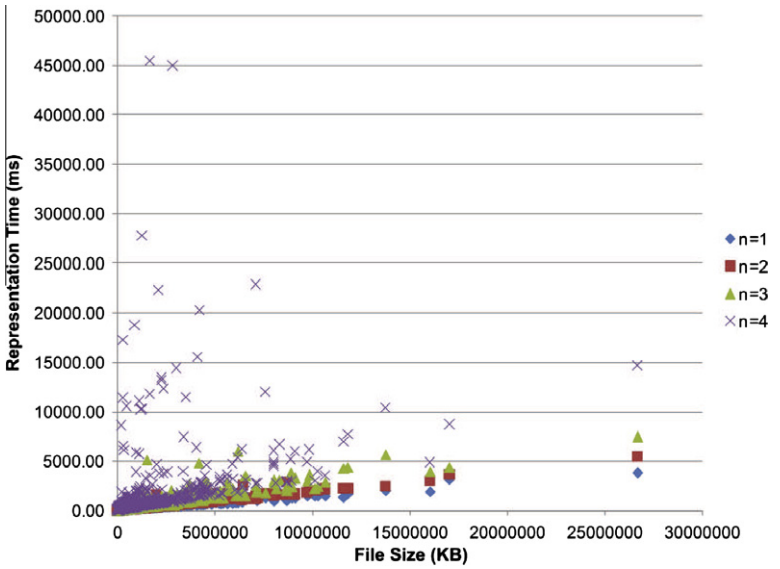

Fig. 7. Time results for the representation time. As we can see the required time increases along with the opcode-sequence length and the size of the executable.

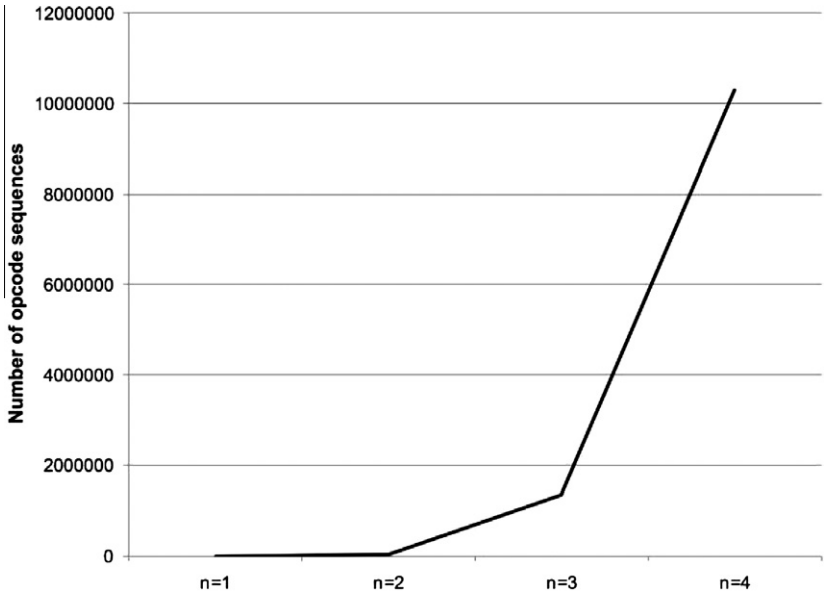

Fig. 8. The number of features for different opcode-sequence lengths. As we can see the number of different opcode sequences increases along with the opcode-sequence length.

With regards to number of different opcode-sequences extracted, Fig. 8 shows the tendency of the number of different opcode sequences. In this step, we deleted the opcode-sequences which have always a value of 0 of WTF (Weighted Term Frequency) from the representation to reduce the total number of different features. The number of different opcode sequences increases with the chosen opcode-sequence length. In particular, for $n = 1$ we obtained 348 different sequences, 51,949 for $n = 2$, 1,360,744 for $n = 3$ and 10,309,792 different opcode sequences for $n = 4$.

Table 1 shows the required time for the selection of the top 1000 opcode sequences using *Information Gain* [34]. For $n = 1$, there was no need of a feature selection step since the total number of opcode sequences was 348. For $n = 2$, a total time of 731,038.17 ms were required. For opcode-sequence length longer than 2 we were unable to perform a feature selection step using the following machine: Intel Core 2 Quad CPU clocked at 2.83 GHz with 8 GB of RAM memory. In fact, we were not able to open the file with the WEKA software [25] to perform the top 1000 feature selection step neither to use an

Table 1
Time results for the feature reduction step.

N	Feature selection time (ms)
1	Not required
2	731,038.17
1 and 2 combined	1,003,985.76
3	N/A
4	N/A

implementation of the feature selection step using the WEKA API. Therefore, the rest of the steps are only tested for $n = 1$, $n = 2$ and the combination of opcode sequences of length 1 and 2.

Table 2 shows the required time to train and test the data-mining models with an opcode-sequence length of 1. Note that we did not reduce the number of features of this dataset. The KNN algorithm did not require any training time. However, it was the slowest in terms of testing time, with results between 1.13 and 1.66 ms. SVM with polynomial kernel was the fastest of the tested configurations for SVM, achieving a training time of 1.42 ms and a testing time of 0.01 ms. Naïve Bayes required 5.83 ms for training and 0.53 s for testing. The performance of the Bayesian networks depended on the algorithm used. Overall, we found that K2 is the fastest Bayesian classifier, requiring 9.29 ms for training and 0.22 ms for testing. TAN had the slowest training time at 2585.08 ms; however, it only required 0.32 ms for the testing step. Among the decision trees, Random Forest performed faster than J48, with 4.37 ms of training time and 0.00 ms of testing time.

Table 3 shows the required time to train and test the data-mining models with an opcode-sequence length of 2. This time, we reduced the number of features of this dataset to 1000. KNN was also the slowest in terms of testing time for this configuration, with results between 2.96 and 4.82 ms. SVM with polynomial kernel was also the fastest of the tested configurations for SVM, achieving a training time of 3.76 ms and a testing time of 0.01 ms. Naïve Bayes required 4.29 ms for training and 0.10 s for testing. Overall, we found that K2 is the fastest Bayesian classifier, requiring 8.93 ms for training and 0.10 ms for testing. TAN had the slowest training time at 488.69 ms while it only required 0.15 ms for the testing step. Random Forest performed also faster than J48, with 2.68 ms of training time.

Table 4 shows the required time to train and test the data-mining models with a combined opcode-sequence length of 1 and 2. This time, the results were nearly the same than the results for an opcode-sequence length of 2 because the number of features was the same: 1000 features.

Table 5, Figs. 9 and 10 show the results for an opcode-sequence length of one. This opcode-sequence length represents only executable files as a statistical distribution of opcodes. The best results were obtained by the SVM trained with Pearson VII kernel, which yielded an accuracy of 92.92%. Nearly every classifier yielded results with accuracies greater than 90%. Nevertheless, Bayesian networks trained with K2, Hill Climber and Naïve Bayes performed worse that their counterparts. The best TPR results were achieved by KNN $K = 2$ with a TPR of 0.95. Most of classifiers obtained a detection ratio higher than 85%, except of, as occurred in accuracy results, Bayesian networks trained with K2, Hill Climber, Naïve Bayes and SVM trained with Radial Basis Function Kernel. In terms of FPR false positive ratio, a very important measure, especially for commercial antivirus software, the lowest rate of false positive was achieved using SVM trained with Pearson VII Kernel. Bayesian networks trained with K2 and Hill Climber, Naïve Bayes and KNN with $K = 2$ yielded a FPR higher or equal than 0.10.

Table 2
Time results of data-mining classifiers for $n = 1$.

Classifier	Training time (ms)	Testing time (ms)
KNN $K = 1$	0 ± 0.00	1.13 ± 0.10
KNN $K = 2$	0 ± 0.00	1.29 ± 0.10
KNN $K = 3$	0 ± 0.00	1.35 ± 0.10
KNN $K = 4$	0 ± 0.00	1.43 ± 0.10
KNN $K = 5$	0 ± 0.00	1.43 ± 0.10
KNN $K = 6$	0 ± 0.00	1.44 ± 0.10
KNN $K = 7$	0 ± 0.00	1.50 ± 0.10
KNN $K = 8$	0 ± 0.00	1.58 ± 0.11
KNN $K = 9$	0 ± 0.00	1.62 ± 0.09
KNN $K = 10$	0 ± 0.00	1.66 ± 0.12
DT: J48	43.03 ± 3.56	0.00 ± 0.00
DT: random forest	4.37 ± 0.20	0.00 ± 0.00
SVM: RBF	14.18 ± 1.68	0.58 ± 0.04
SVM: polynomial	1.42 ± 0.11	0.01 ± 0.00
SVM: normalised polynomial	5.58 ± 0.55	0.36 ± 0.03
SVM: Pearson VII	9.22 ± 0.76	0.49 ± 0.04
Naïve Bayes	5.83 ± 0.11	0.53 ± 0.06
BN: K2	9.29 ± 0.80	0.22 ± 0.06
BN: hill climber	2069.48 ± 34.89	0.20 ± 0.03
BN: TAN	2585.08 ± 226.31	0.32 ± 0.04

Table 3
Time results of data-mining classifiers for $n = 2$.

Classifier	Training time (ms)	Testing time (ms)
KNN $K = 1$	0.00 ± 0.00	2.96 ± 0.16
KNN $K = 2$	0.00 ± 0.00	3.95 ± 0.39
KNN $K = 3$	0.00 ± 0.00	4.11 ± 0.26
KNN $K = 4$	0.00 ± 0.00	4.34 ± 0.18
KNN $K = 5$	0.00 ± 0.00	4.54 ± 0.17
KNN $K = 6$	0.00 ± 0.00	4.6 ± 0.175
KNN $K = 7$	0.00 ± 0.00	4.96 ± 0.28
KNN $K = 8$	0.00 ± 0.00	5.18 ± 0.19
KNN $K = 9$	0.00 ± 0.00	4.70 ± 0.20
KNN $K = 10$	0.00 ± 0.00	4.82 ± 0.16
DT: J48	32.43 ± 3.73	0.00 ± 0.00
DT: random forest $N = 10$	2.68 ± 0.09	0.00 ± 0.01
SVM: RBF	33.08 ± 7.44	1.76 ± 0.09
SVM: polynomial	3.76 ± 0.25	0.01 ± 0.00
SVM: normalised polynomial	16.35 ± 1.26	1.30 ± 0.07
SVM: Pearson VII	56.47 ± 3.31	3.20 ± 0.09
Naïve Bayes	4.29 ± 0.24	0.10 ± 0.03
BN: K2	8.93 ± 1.09	0.10 ± 0.02
BN: hill climber	501.71 ± 31.60	0.10 ± 0.02
BN: TAN	488.69 ± 7.80	0.15 ± 0.01

Table 4
Time results of data-mining classifiers for the combination of features of opcode-sequence length of 1 and 2.

Classifier	Training time (ms)	Testing time (ms)
KNN $K = 1$	0.00 ± 0.00	3.64 ± 0.22
KNN $K = 2$	0.00 ± 0.00	4.31 ± 0.20
KNN $K = 3$	0.00 ± 0.02	4.60 ± 0.28
KNN $K = 4$	0.00 ± 0.00	4.88 ± 0.22
KNN $K = 5$	0.00 ± 0.00	5.20 ± 0.28
KNN $K = 6$	0.00 ± 0.00	5.23 ± 0.25
KNN $K = 7$	0.00 ± 0.00	5.53 ± 0.24
KNN $K = 8$	0.00 ± 0.00	5.09 ± 6.38
KNN $K = 9$	0.00 ± 0.00	5.84 ± 0.27
KNN $K = 10$	0.00 ± 0.00	5.97 ± 0.27
DT: J48	32.43 ± 2.98	0.00 ± 0.00
DT: random forest $N = 10$	2.68 ± 0.16	0.00 ± 0.00
SVM: RBF	34.24 ± 3.07	1.76 ± 2.02
SVM: polynomial	5.35 ± 0.34	0.02 ± 0.01
SVM: normalised polynomial	18.88 ± 1.69	1.87 ± 0.17
SVM: Pearson VII	58.19 ± 4.03	3.41 ± 0.20
Naïve Bayes	4.29 ± 0.19	0.12 ± 0.03
BN: K2	9.16 ± 0.33	0.12 ± 0.02
BN: hill climber	453.93 ± 35.06	0.10 ± 0.03
BN: TAN	448.63 ± 10.27	0.14 ± 0.01

Finally, with regards to the area under the ROC curve, all classifiers other than Naïve Bayes and SVM trained with RBF Kernel, achieved areas bigger than 0.90.

When using a length of two (shown in Table 6, Figs. 11 and 12), the results substantially improved. The results for SVM trained with Normalised Polynomial Kernel were the best, at 95.90%. Bayesian networks trained with K2 and Hill Climber were the worst in terms of accuracy—weaker than 90%. In terms of TPR, Random Forest, SVM trained with Polynomial Kernel and KNN $K = 2$ yielded the best results, 96%. With the exception of Bayesian networks trained with K2 and Hill Climber, the classifiers all achieved ratios higher than 0.85. For FPR, SVM trained with the Normalised Polynomial Kernel outperformed the rest of classifiers with a ratio of 0.2. Finally, every model obtained results for AUC greater than 90%.

Surprisingly, the results for the combination of opcode sequences with lengths one and two (refer to Table 7, Figs. 13 and 14), did not yield better results than experiments with $n = 2$. Specifically, the results were slightly lower than those obtained with the sequences of length two, with the exception of the Bayesian networks algorithms. In terms of model accuracy, SVM trained with the Normalised Polynomial Kernel achieved 95.80% accuracy, 0.10% less than the previous result using an opcode-sequence length of two. The results for TPR were the same as with a sequence length of two: Random Forest, SVM trained with the Polynomial Kernel and KNN $K = 2$ yielded 0.96 for TPR. The same outcome occurred for FPR, for which SVM trained with the Normalised Polynomial Kernel was the best, at 0.02. Finally, the result for AUC were also the same as in opcode length 2.

Table 5
Results for an opcode-sequence length of 1.

Classifier	Accuracy (%)	TPR	FPR	AUC
KNN $K = 1$	92.83 ± 1.90	0.93 ± 0.02	0.07 ± 0.03	0.93 ± 0.02
KNN $K = 2$	90.75 ± 2.04	0.95 ± 0.02	0.14 ± 0.03	0.95 ± 0.02
KNN $K = 3$	91.71 ± 1.95	0.91 ± 0.03	0.08 ± 0.03	0.96 ± 0.01
KNN $K = 4$	91.40 ± 2.04	0.93 ± 0.03	0.10 ± 0.03	0.96 ± 0.01
KNN $K = 5$	90.95 ± 2.08	0.90 ± 0.03	0.08 ± 0.02	0.96 ± 0.01
KNN $K = 6$	90.87 ± 2.08	0.91 ± 0.03	0.10 ± 0.03	0.96 ± 0.01
KNN $K = 7$	90.80 ± 2.28	0.89 ± 0.04	0.07 ± 0.03	0.97 ± 0.01
KNN $K = 8$	90.68 ± 2.25	0.90 ± 0.03	0.09 ± 0.03	0.97 ± 0.01
KNN $K = 9$	90.40 ± 2.25	0.88 ± 0.03	0.08 ± 0.03	0.97 ± 0.01
KNN $K = 10$	90.36 ± 2.31	0.90 ± 0.03	0.09 ± 0.03	0.97 ± 0.01
DT: J48	91.25 ± 2.15	0.92 ± 0.03	0.09 ± 0.03	0.91 ± 0.03
DT: random forest $N = 10$	91.43 ± 1.98	0.92 ± 0.03	0.09 ± 0.03	0.97 ± 0.01
SVM: RBF	81.67 ± 2.57	0.67 ± 0.05	0.03 ± 0.02	0.82 ± 0.03
SVM: polynomial	89.65 ± 2.08	0.88 ± 0.03	0.09 ± 0.03	0.90 ± 0.02
SVM: normalised polynomial	91.67 ± 2.05	0.88 ± 0.04	0.05 ± 0.02	0.92 ± 0.02
SVM: Pearson VII	92.92 ± 1.86	0.89 ± 0.03	0.03 ± 0.02	0.93 ± 0.02
Naïve Bayes	79.64 ± 3.02	0.90 ± 0.03	0.31 ± 0.05	0.86 ± 0.02
BN: K2	84.24 ± 3.02	0.79 ± 0.05	0.10 ± 0.03	0.92 ± 0.03
BN: hill climber	84.24 ± 3.03	0.79 ± 0.05	0.10 ± 0.03	0.92 ± 0.02
BN: TAN	90.65 ± 2.14	0.87 ± 0.04	0.05 ± 0.02	0.97 ± 0.01

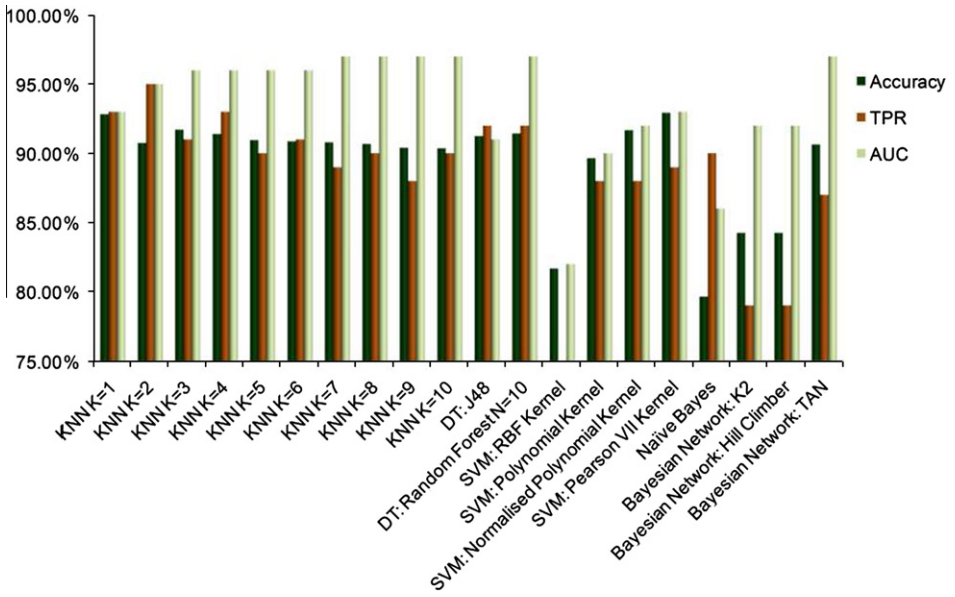

Fig. 9. Comparison of the results in terms of accuracy of the classifiers for an opcode-sequence length of 1.

We would like to point out several observations from the experiments. First, the best overall results were obtained with polynomial kernel classifiers and decision trees. These two types of classifiers have a long history on text classification and they behave well in this domain too. Second, Bayesian networks, even though they detect much of the malware, yielded a high false positive ratio. Because minimising false positive ratio is one of our goals, we do not support these methods for unknown malware detection. Finally, K-Nearest Neighbour, in spite of being a very simple method, yielded very favourable results.

6. Discussion

The obtained results validate our initial hypothesis that building an unknown malware detector based on opcode-sequence is feasible. The machine-learning classifiers achieved high performance in classifying unknown malware. Nevertheless, there are several considerations regarding the viability of this method.

Fist, the processing overhead of method is highly dependant of the length of the opcode sequences. In our experiments, we analysed how the length of opcode sequences influences the processing overhead of the method. In particular, we were

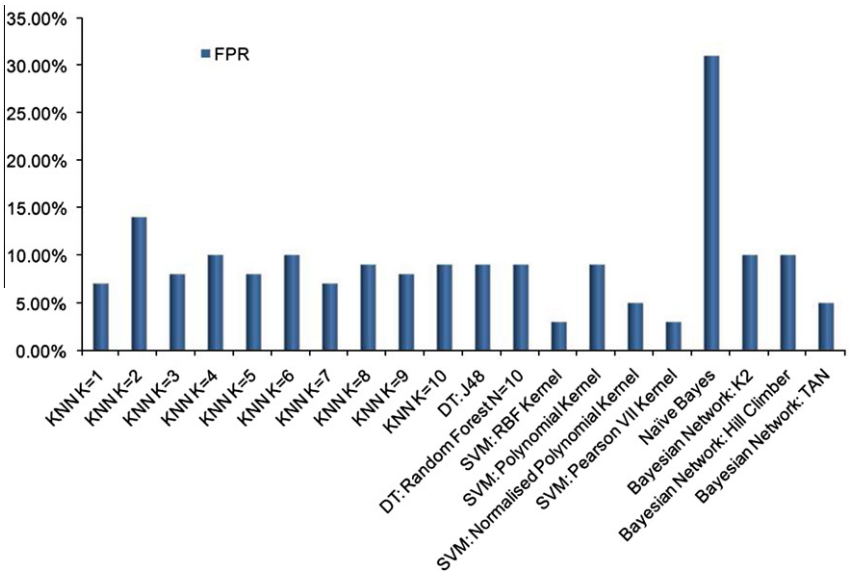

Fig. 10. Comparison of the results in terms of FPR of the classifiers for an opcode-sequence length of 1.

Table 6
Results for an opcode-sequence length of 2.

Classifier	Accuracy (%)	TPR	FPR	AUC
KNN $K = 1$	94.83 ± 1.43	0.95 ± 0.02	0.05 ± 0.02	0.95 ± 0.01
KNN $K = 2$	93.15 ± 1.72	0.96 ± 0.02	0.10 ± 0.03	0.96 ± 0.01
KNN $K = 3$	94.16 ± 1.67	0.94 ± 0.02	0.05 ± 0.03	0.97 ± 0.01
KNN $K = 4$	93.89 ± 1.68	0.95 ± 0.02	0.07 ± 0.03	0.97 ± 0.01
KNN $K = 5$	93.50 ± 1.85	0.92 ± 0.03	0.05 ± 0.02	0.97 ± 0.01
KNN $K = 6$	93.38 ± 1.83	0.93 ± 0.03	0.06 ± 0.02	0.98 ± 0.01
KNN $K = 7$	92.87 ± 1.77	0.90 ± 0.03	0.04 ± 0.02	0.98 ± 0.01
KNN $K = 8$	92.89 ± 1.92	0.91 ± 0.03	0.05 ± 0.02	0.98 ± 0.01
KNN $K = 9$	92.10 ± 2.07	0.88 ± 0.03	0.04 ± 0.02	0.98 ± 0.01
KNN $K = 10$	92.24 ± 2.04	0.90 ± 0.03	0.05 ± 0.02	0.97 ± 0.01
DT: J48	92.61 ± 1.95	0.93 ± 0.02	0.08 ± 0.03	0.93 ± 0.02
DT: random forest $N = 10$	95.26 ± 1.57	0.96 ± 0.02	0.06 ± 0.03	0.99 ± 0.01
SVM: RBF	91.93 ± 2.01	0.89 ± 0.03	0.05 ± 0.02	0.92 ± 0.02
SVM: polynomial	95.50 ± 1.56	0.96 ± 0.02	0.05 ± 0.02	0.95 ± 0.02
SVM: normalised polynomial	95.90 ± 1.59	0.94 ± 0.03	0.02 ± 0.01	0.96 ± 0.02
SVM: Pearson VII	94.35 ± 1.70	0.95 ± 0.02	0.06 ± 0.03	0.94 ± 0.02
Naïve Bayes	90.02 ± 2.16	0.90 ± 0.03	0.10 ± 0.03	0.93 ± 0.02
BN: K2	86.73 ± 2.70	0.83 ± 0.04	0.09 ± 0.03	0.94 ± 0.02
BN: hill climber	86.73 ± 2.70	0.83 ± 0.04	0.09 ± 0.03	0.94 ± 0.02
BN: TAN	93.40 ± 1.77	0.91 ± 0.03	0.04 ± 0.02	0.98 ± 0.01

not capable of building the classifiers with opcode sequences longer than 2 because we could not perform the feature selection step. If we were able to perform that step and we selected the top ranked 1000 opcode sequences then the classifiers could have been generated. Since the accuracy results using a length of 2 are high, we consider that there is not benefit of such lengths. Besides, a long opcode sequence can be easily evaded by a malware obfuscator through code transposition techniques whereas a short one may be harder to evade.

Second, our representation technique only employs opcodes and discards the operands in the machine code instructions. The work of Bilar [4] studied the representativeness of single opcodes for determine the legitimacy of an application and it was proved that single opcodes are statistically dependant with regards to the class of a software. However, we do not discard to enhance our representation including operands within the instructions in a further work. To use these operands, we have to first perform a classification of the operands, grouping every type of operand that have the same meaning together.

Third, because of the static nature of the proposed method, it cannot counter *packed* malware. Packed malware is the result of cyphering the payload of the executable and deciphering it when the executable is finally loaded into memory. Indeed, static detection methods can deal with packed malware only by using the signatures of the packers. Accordingly,

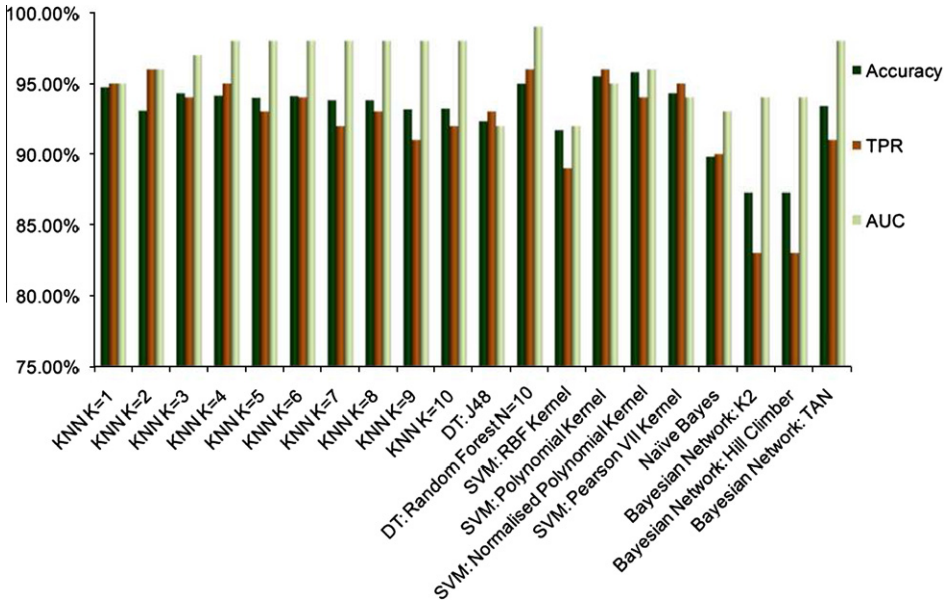

Fig. 11. Comparison of the results in terms of accuracy of the classifiers for an opcode-sequence length of 2.

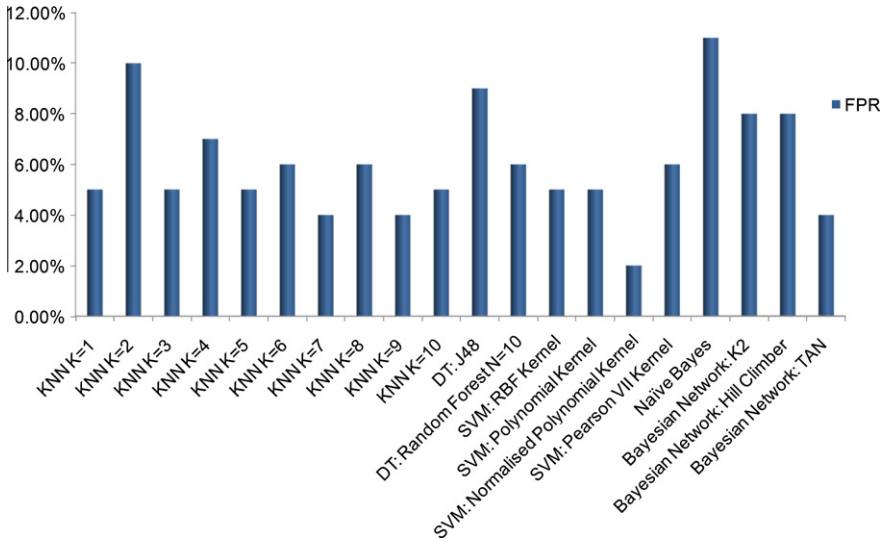

Fig. 12. Comparison of the results in terms of FPR of the classifiers for an opcode-sequence length of 2.

dynamic analysis seems a more promising solution to this problem [32]. One solution to solve this obvious limitation of our malware detection method is the use of a generic dynamic unpacking schema such as PolyUnpack [63], Renovo [32], Omni-Unpack [47] and Eureka [72]. These methods execute the sample in a contained environment and extract the actual payload, allowing further static or dynamic analysis of the executable. Another solution is to use concrete unpacking routines to recover the actual payload that requires one routine per packing algorithm [75]. 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 [56]. Still, these techniques cannot cope with the increasing use of packing techniques, and, we suggest the use of dynamic unpacking schemas to confront the problem.

Fourth, it may seem that our method detects mainly the compiler used to create executables. In fact, the use of a specific compiler inherently renders an executable rich in several opcode sequences. Nonetheless, our machine-learning method represents all information in the training dataset. We also included a pre-processing step that conducts a feature extraction. Hence, selection of the most relevant opcode-sequence confirms our approach goes further than detecting different compilers. In addition, when selecting executables to be part of the dataset, we analysed them using PEiD, a tool that detects most

Table 7
Results for the combination of features of opcode-sequence length of 1 and 2.

Classifier	Accuracy (%)	TPR	FPR	AUC
KNN $K = 1$	94.73 ± 1.47	0.95 ± 0.02	0.05 ± 0.02	0.95 ± 0.01
KNN $K = 2$	93.06 ± 1.75	0.96 ± 0.02	0.10 ± 0.03	0.96 ± 0.01
KNN $K = 3$	94.30 ± 1.63	0.94 ± 0.02	0.05 ± 0.02	0.97 ± 0.01
KNN $K = 4$	94.13 ± 1.77	0.95 ± 0.02	0.07 ± 0.03	0.98 ± 0.01
KNN $K = 5$	93.98 ± 1.87	0.93 ± 0.03	0.05 ± 0.02	0.98 ± 0.01
KNN $K = 6$	94.09 ± 1.80	0.94 ± 0.02	0.06 ± 0.02	0.98 ± 0.01
KNN $K = 7$	93.80 ± 1.88	0.92 ± 0.03	0.04 ± 0.02	0.98 ± 0.01
KNN $K = 8$	93.80 ± 1.79	0.93 ± 0.03	0.06 ± 0.02	0.98 ± 0.01
KNN $K = 9$	93.16 ± 1.92	0.91 ± 0.03	0.04 ± 0.02	0.98 ± 0.01
KNN $K = 10$	93.20 ± 1.94	0.92 ± 0.03	0.05 ± 0.02	0.98 ± 0.01
DT: J48	92.34 ± 1.58	0.93 ± 0.02	0.09 ± 0.02	0.92 ± 0.02
DT: random forest $N = 10$	94.98 ± 1.77	0.96 ± 0.02	0.06 ± 0.03	0.99 ± 0.01
SVM: RBF	91.70 ± 2.00	0.89 ± 0.02	0.05 ± 0.03	0.92 ± 0.02
SVM: polynomial	95.50 ± 1.52	0.96 ± 0.02	0.05 ± 0.02	0.95 ± 0.02
SVM: normalised polynomial	95.80 ± 1.59	0.94 ± 0.02	0.02 ± 0.01	0.96 ± 0.02
SVM: Pearson VII	94.29 ± 1.77	0.95 ± 0.02	0.06 ± 0.03	0.94 ± 0.02
Naïve Bayes	89.81 ± 2.30	0.90 ± 0.03	0.11 ± 0.03	0.93 ± 0.02
BN: K2	87.29 ± 2.59	0.83 ± 0.04	0.08 ± 0.03	0.94 ± 0.02
BN: hill climber	87.29 ± 2.59	0.83 ± 0.04	0.08 ± 0.03	0.94 ± 0.02
BN: TAN	93.40 ± 1.80	0.91 ± 0.03	0.04 ± 0.02	0.98 ± 0.01

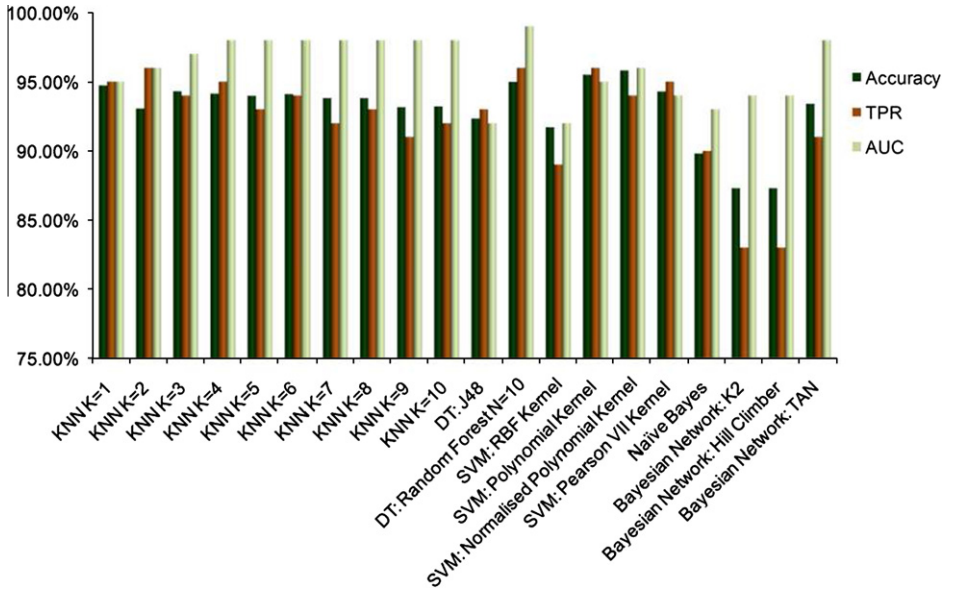

Fig. 13. Comparison of the results in terms of accuracy for the combination of features of opcode-sequence length of 1 and 2.

common packers, cryptors and compilers for Portable Executable (PE)[9] files. After removing the packed ones (our method would not be able to detect them), there was no significant difference in the compilers used for benign software and malware. We found that the most common known compilers in the malware dataset were Microsoft Visual Basic, Microsoft Visual C++, Borland Delphi and Borland C++. In the benign dataset, the most common compilers were Microsoft Visual C++, Borland C++and Borland Delphi. We may be able to apply the ability to detect compilers for our own benefit. We can apply our method to detect whether an executable is packed. If the executable is packed, then we may unpack it using a dynamic unpacking schema capable of extracting the original payload. Afterwards or if the sample is not packed, we can analyse it to determine out whether it is malware.

Finally, the use of supervised machine-learning algorithms for the model training, can be a problem in itself. In our experiments, we used a training dataset that is very small when compared with commercial antivirus databases. As the dataset size grows, so does the issue of scalability. This problem produces excessive storage requirements, increases time complexity and impairs the general accuracy of the models [10]. To reduce disproportionate storage and time costs, it is necessary to

[9] PE is the format of windows binary files.

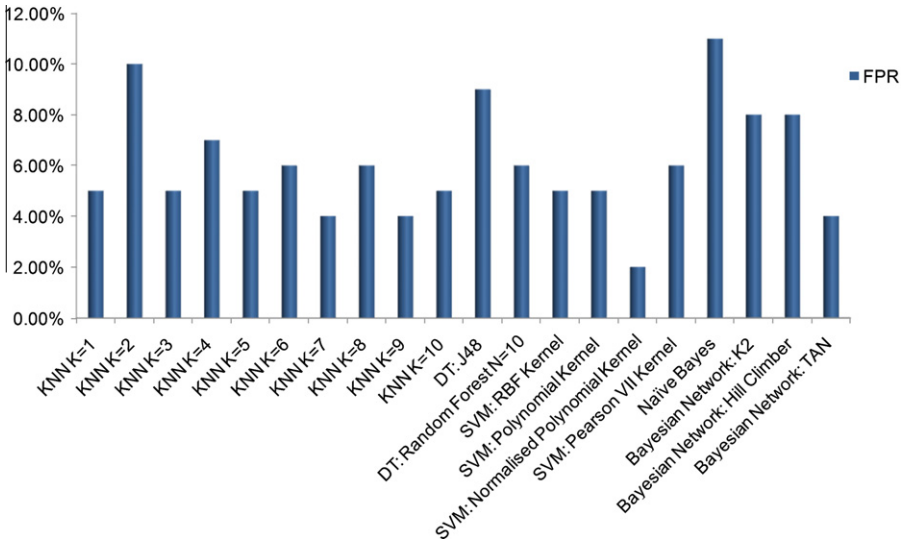

Fig. 14. Comparison of the results in terms of FPR for the combination of features of opcode-sequence length of 1 and 2.

reduce the original training set [19]. In order to solve this issue, *data reduction* is normally considered an appropriate pre-processing optimisation technique [59,78]. Such techniques have many potential advantages such as reducing measurement, storage and transmission; decreasing training and testing times; confronting the *curse of dimensionality* to improve prediction performance in terms of speed, accuracy and simplicity and facilitating data visualization and understanding [76,20]. Data reduction can be implemented in two ways. On the one hand, *Instance Selection* (IS) seeks to reduce the evidences (i.e., number of rows) in the training set by selecting the most relevant instances or re-sampling new ones [43]. On the other hand, *Feature Selection* (FS) decreases the number of attributes or features (i.e., columns) in the training set [44]. We applied FS in our experiments when selecting the 1000 top-ranked opcode sequences. Because both IS and FS are very effective at reducing the size of the training set and helping to filtrate and clean noisy data, thereby improving the accuracy of machine-learning classifiers [6,21], we strongly encourage the use of these methods.

7. Conclusions

Malware detection has become a major topic of research and concern owing to the increasing growth of malicious code in recent years. The classic signature methods employed by antivirus vendors are no longer completely effective because the large volume of new malware renders them impractical. Therefore, signature methods must be complemented with more complex approaches that provide detection of unknown malware families.

In this paper, we present a method for malware detection. Specifically, we propose a method for representing malware that relied on opcodes sequences in order to construct a vector representation of the executables. In this way, we were able to train machine-learning algorithms to detect unknown malware variants. Our experiments show that this method provides a good detection ratio of unknown malware while keeping a low false positive ratio.

The future development of this malware detection system will be concentrated in three main research areas. First, we will focus on facing packed executables using a hybrid dynamic-static approach. Second, we plan to apply this method for the detection of packed executables. Finally, we will study the problem of scalability of malware databases using a combination of feature and instance selection methods.

Acknowledgements

This research was financially supported by the Ministry of Industry of Spain, project Cenit SEGUR@, Security and Trust in the Information Society, (BOE 35, 09/02/2007, CDTI). We would also like to acknowledge to the anonymous reviewers for their helpful comments and suggestions.

References

[1] S. Amari, S. Wu, Improving support vector machine classifiers by modifying kernel functions, Neural Networks 12 (1999) 783–789.
[2] T. Bayes, An essay towards solving a problem in the doctrine of chances, Philosophical Transactions of the Royal Society 53 (1763) 370–418.
[3] J. Bergeron, M. Debbabi, M. Erhioui, B. Ktari, Static analysis of binary code to isolate malicious behaviors, in: Proceedings of the 1999 Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises, 1999, pp. 184–189.
[4] D. Bilar, Opcodes as predictor for malware, International Journal of Electronic Security and Digital Forensics 1 (2007) 156–168.

[5] C. Bishop, Pattern Recognition and Machine Learning, Springer, New York, 2006.
[6] A. Blum, P. Langley, Selection of relevant features and examples in machine learning, Artificial Intelligence 97 (1997) 245–271.
[7] L. Breiman, Random forests, Machine Learning 45 (2001) 5–32.
[8] D. Bruschi, L. Martignoni, M. Monga, Detecting self-mutating malware using control-flow graph matching, Lecture Notes in Computer Science 4064 (2006) 129.
[9] D. Cai, J. Theiler, M. Gokhale, Detecting a malicious executable without prior knowledge of its patterns, in: Proceedings of the 2005 Defense and Security Symposium. Information Assurance, and Data Network Security, vol. 5812, 2005, pp. 1–12.
[10] J. Cano, F. Herrera, M. Lozano, On the combination of evolutionary algorithms and stratified strategies for training set selection in data mining, Applied Soft Computing Journal 6 (2006) 323–332.
[11] E. Castillo, J.M. Gutiérrez, A.S. Hadi, Expert Systems and Probabilistic Network Models, erste ed., Springer, New York, NY, USA, 1996.
[12] O. Chapelle, B. Schölkopf, A. Zien, Semi-Supervised Learning, MIT Press, 2006.
[13] M. Chouchane, A. Lakhotia, Using engine signature to detect metamorphic malware, in: Proceedings of the 2006 ACM workshop on Recurring Malcode, ACM, New York, NY, USA, 2006, pp. 73–78.
[14] M. Christodorescu, Behavior-based malware detection, Ph.D. Thesis, 2007.
[15] M. Christodorescu, S. Jha, Testing malware detectors, ACM SIGSOFT Software Engineering Notes 29 (2004) 34–44.
[16] M. Christodorescu, S. Jha, Static analysis of executables to detect malicious patterns, in: Proceedings of the 12th USENIX Security Symposium, 2003, pp. 169–186.
[17] M. Christodorescu, S. Jha, S. Seshia, D. Song, R. Bryant, Semantics-aware malware detection, in: Proceedings of the 2005 IEEE Symposium on Security and Privacy, 2005, pp. 32–46.
[18] G.F. Cooper, E. Herskovits, A bayesian method for constructing bayesian belief networks from databases, in: Proceedings of the 1991 Conference on Uncertainty in Artificial Intelligence, 1991.
[19] I. Czarnowski, P. Jedrzejowicz, Instance reduction approach to machine learning and multi-database mining, in: Proceedings of the 2006 Scientific Session organized during XXI Fall Meeting of the Polish Information Processing Society, Informatica, ANNALES Universitatis Mariae Curie-Skłodowska, Lublin, 2006, pp. 60–71.
[20] M. Dash, H. Liu, Consistency-based search in feature selection, Artificial Intelligence 151 (2003) 155–176.
[21] J. Derrac, S. Garcıa, F. Herrera, A first study on the use of coevolutionary algorithms for instance and feature selection, in: Proceedings of the 2009 International Conference on Hybrid Artificial Intelligence Systems, Springer, 2009, pp. 557–564.
[22] C. Elkan, The foundations of cost-sensitive learning, in: Proceedings of the 2001 International Joint Conference on Artificial Intelligence, 2001, pp. 973–978.
[23] D. Fisch, A. Hofmann, B. Sick, On the versatility of radial basis function neural networks: a case study in the field of intrusion detection, Information Sciences 180 (2010) 2421–2439.
[24] E. Fix, J.L. Hodges, Discriminatory analysis: nonparametric discrimination: small sample performance, Technical Report Project 21-49-004, Report Number 11, 1952.
[25] S. Garner, Weka: the Waikato environment for knowledge analysis, in: Proceedings of the 1995 New Zealand Computer Science Research Students Conference, 1995, pp. 57–64.
[26] D. Geiger, M. Goldszmidt, G. Provan, P. Langley, P. Smyth, Bayesian network classifiers, in: Machine Learning 29 (1997) 131–163.
[27] J. Huang, I. Liao, et al, Shielding wireless sensor network using markovian intrusion detection system with attack pattern mining, Information Sciences (2011).
[28] N. Idika, A. Mathur, A survey of malware detection techniques, Technical Report, Department of Computer Science, Purdue University, 2007.
[29] G. Jacob, H. Debar, E. Filiol, Behavioral detection of malware: from a survey towards an established taxonomy, Journal in Computer Virology 4 (2008) 251–266.
[30] X. Jin, A. Xu, R. Bie, P. Guo, Machine learning techniques and chi-square feature selection for cancer classification using SAGE gene expression profiles, Lecture Notes in Computer Science 3916 (2006) 106–115.
[31] I. Jolliffe, Principal Component Analysis, Springer Verlag, 2002.
[32] M. Kang, P. Poosankam, H. Yin, Renovo: a hidden code extractor for packed executables, in: Proceedings of the 2007 ACM workshop on Recurring Malcode, 2007, pp. 46–53.
[33] M. Karim, A. Walenstein, A. Lakhotia, L. Parida, Malware phylogeny generation using permutations of code, Journal in Computer Virology 1 (2005) 13–23.
[34] J. Kent, Information gain and a general measure of correlation, Biometrika 70 (1983) 163.
[35] R. Kohavi, A study of cross-validation and bootstrap for accuracy estimation and model selection, in: Proceedings of the 1995 International Joint Conference on Artificial Intelligence, vol. 14, 1995, pp. 1137–1145.
[36] J. Kolter, M. Maloof, Learning to detect malicious executables in the wild, in: Proceedings of the 2004 ACM SIGKDD International Conference on Knowledge discovery and Data Mining, ACM, New York, NY, USA, 2004, pp. 470–478.
[37] S. Kotsiantis, Supervised machine learning: a review of classification techniques, in: Proceeding of the 2007 Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies, 2007, pp. 3–24.
[38] S. Kotsiantis, P. Pintelas, Recent advances in clustering: a brief survey, WSEAS Transactions on Information Science and Applications 1 (2004) 73–81.
[39] G. Kou, Y. Peng, Z. Chen, Y. Shi, Multiple criteria mathematical programming for multi-class classification and application in network intrusion detection, Information Sciences 179 (2009) 371–381.
[40] N. Kuzurin, A. Shokurov, N. Varnovsky, V. Zakharov, On the concept of software obfuscation in computer security, Lecture Notes in Computer Science 4779 (2007) 281.
[41] D. Lewis, Naive (Bayes) at forty: the independence assumption in information retrieval, Lecture Notes in Computer Science 1398 (1998) 4–18.
[42] W. Li, K. Wang, S. Stolfo, B. Herzog, Fileprints: identifying file types by n-gram analysis, in: Proceedings of the 2005 IEEE Workshop on Information Assurance and Security, 2005.
[43] H. Liu, H. Motoda, Instance Selection and Construction for Data Mining, Kluwer Academic Publication, 2001.
[44] H. Liu, H. Motoda, Computational Methods of Feature Selection, Chapman and Hall/CRC, 2008.
[45] R. Lo, K. Levitt, R. Olsson, MCF: a malicious code filter, Computers and Security 14 (1995) 541–566.
[46] R. Mántaras, A distance-based attribute selection measure for decision tree induction, Machine Learning 6 (1991) 81–92.
[47] L. Martignoni, M. Christodorescu, S. Jha, Omniunpack: fast, generic, and safe unpacking of malware, in: Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC), 2007, pp. 431–441.
[48] M. McGill, G. Salton, Introduction to Modern Information Retrieval, McGraw-Hill, 1983.
[49] P. Morley, Processing virus collections, in: Proceedings of the 2001 Virus Bulletin Conference (VB2001), Virus Bulletin, 2001, pp. 129–134.
[50] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, Y. Elovici, Unknown malcode detection via text categorization and the imbalance problem, in: Proceedings of the 2008 IEEE International Conference on Intelligence and Security Informatics (ISI), 2008, pp. 156–161.
[51] G. Ollmann, The evolution of commercial malware development kits and colour-by-numbers custom malware, Computer Fraud and Security 2008 (2008) 4–7.
[52] J. Pearl, Reverend bayes on inference engines: a distributed hierarchical approach, in: Proceedings of the 1982 National Conference on Artificial Intelligence, 1982, pp. 133–136.
[53] H. Peng, F. Long, C. Ding, Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy, IEEE Transactions on Pattern Analysis and Machine Intelligence (2005) 1226–1238.

[54] R. Perdisci, A. Lanzi, W. Lee, Classification of packed executables for accurate computer virus detection, Pattern Recognition Letters 29 (2008) 1941–1946.
[55] R. Perdisci, A. Lanzi, W. Lee, McBoost: boosting scalability in malware collection and analysis using statistical classification of executables, in: Proceedings of the 23rd Annual Computer Security Applications Conference, 2008, pp. 301–310.
[56] F. Perriot, P. Ferrie, Principles and practise of x-raying, in: Proceedings of the 2004 Virus Bulletin International Conference, 2004, pp. 51–66.
[57] J. Platt, Sequential minimal optimization: a fast algorithm for training support vector machines, Advances in Kernel Methods-Support Vector Learning 208 (1999).
[58] B. Potter, G. Day, The effectiveness of anti-malware tools, Computer Fraud and Security 2009 (2009) 12–13.
[59] D. Pyle, Data Preparation for Data Mining, Morgan Kaufmann, 1999.
[60] J. Quinlan, Induction of decision trees, Machine Learning 1 (1986) 81–106.
[61] J. Quinlan, C4. 5 Programs for Machine Learning, Morgan Kaufmann Publishers, 1993.
[62] S. Robertson, Understanding inverse document frequency: on theoretical arguments for IDF, Journal of Documentation 60 (2004) 503–520.
[63] P. Royal, M. Halpin, D. Dagon, R. Edmonds, W. Lee, Polyunpack: automating the hidden-code extraction of unpack-executing malware, in: Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC), 2006, pp. 289–300.
[64] S.J. Russell, Norvig, second ed., Artificial Intelligence: A Modern Approach, Prentice Hall, 2003.
[65] G. Salton, A. Wong, C. Yang, A vector space model for automatic indexing, Communications of the ACM 18 (1975) 613–620.
[66] I. Santos, F. Brezo, J. Nieves, Y. Penya, B. Sanz, C. Laorden, P. Bringas, Idea: opcode-sequence-based malware detection, in: Engineering Secure Software and Systems, LNCS 5965 (2010) 35–43, doi:10.1007/978-3-642-11747-3_3.
[67] I. Santos, Y. Penya, J. Devesa, P. Bringas, N-Grams-based file signatures for malware detection, in: Proceedings of the 2009 International Conference on Enterprise Information Systems (ICEIS), vol. AIDSS, 2009, pp. 317–320.
[68] R. Schapire, The boosting approach to machine learning: an overview, Lecture Notes in Statistics (2003) 149–172.
[69] M. Schultz, E. Eskin, F. Zadok, S. Stolfo, Data mining methods for detection of new malicious executables, in: Proceedings of the 2001 IEEE Symposium on Security and Privacy, 2001, pp. 38–49.
[70] A. Shabtai, R. Moskovitch, Y. Elovici, C. Glezer, Detection of malicious code by applying machine learning classifiers on static features: a state-of-the-art survey, Information Security Technical Report 14 (2009) 16–29.
[71] M. Shafiq, S. Khayam, M. Farooq, Embedded malware detection using markov n-grams, Lecture Notes in Computer Science 5137 (2008) 88–107.
[72] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, W. Lee, Eureka: a framework for enabling static malware analysis, in: Proceedings of the 2008 European Symposium on Research in Computer Security (ESORICS), 2008, pp. 481–500.
[73] Y. Singh, A. Kaur, R. Malhotra, Comparative analysis of regression and machine learning methods for predicting fault proneness models, International Journal of Computer Applications in Technology 35 (2009) 183–193.
[74] A. Sung, J. Xu, P. Chavez, S. Mukkamala, Static analyzer of vicious executables (save), in: Proceedings of the 2004 Annual Computer Security Applications Conference (ACSAC), 2004, pp. 326–334.
[75] P. Ször, The Art of Computer Virus Research and Defense, Addison-Wesley Professional, 2005.
[76] K. Torkkola, Feature extraction by non parametric mutual information maximization, The Journal of Machine Learning Research 3 (2003) 1415–1438.
[77] C. Tsai, Y. Hsu, C. Lin, W. Lin, Intrusion detection by machine learning: a review, Expert Systems with Applications 36 (2009) 11994–12000.
[78] E. Tsang, D. Yeung, X. Wang, OFFSS: optimal fuzzy-valued feature subset selection, IEEE Transactions on Fuzzy Systems 11 (2003) 202–213.
[79] B. Üstün, W. Melssen, L. Buydens, Facilitating the application of support vector regression by using a universal Pearson VII function based kernel, Chemometrics and Intelligent Laboratory Systems 81 (2006) 29–40.
[80] V. Vapnik, The Nature of Statistical Learning Theory, Springer, 2000.
[81] P. Vinod, R. Jaipur, V. Laxmi, M. Gaur, Survey on malware detection methods, in: Hack, 2009.
[82] J. Xu, A. Sung, P. Chavez, S. Mukkamala, Polymorphic malicious executable scanner by API sequence analysis, in: Proceedings of the 2004 International Conference on Hybrid Intelligent Systems, 2004, pp. 378–383.
[83] C. Zhai, J. Lafferty, A study of smoothing methods for language models applied to information retrieval, ACM Transactions on Information Systems 22 (2004) 179–214.
[84] Q. Zhang, D. Reeves, Metaaware: identifying metamorphic malware, in: Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC), 2007, pp. 411–420.
[85] Y. Zhou, W. Inge, Malware detection using adaptive data compression, in: Proceedings of the 2008 ACM Workshop on AISec, ACM, New York, NY, USA, 2008, pp. 53–60.