Boosting Scalability in Anomaly-based Packed Executable Filtering

Xabier Ugarte-Pedrero, Igor Santos, and Pablo G. Bringas

S^{3}Lab, DeustoTech - Computing, Deusto Institute of Technology
University of Deusto,
Avenida de las Universidades 24, 48007
Bilbao, Spain
{xabier.ugarte,isantos,pablo.garcia.bringas}@deusto.es

Abstract. During the last years, malware writers have been using several techniques to evade detection. One of the most common techniques employed by the anti-virus industry is signature scanning. This method requires the end-host to compare files against a database that should contain signatures for each malware sample. In order to allow their creations to bypass these protection systems, programmers use software encryption tools and code obfuscation techniques to hide the actual behaviour of their malicious programs. One of these techniques is packing, a method that encrypts the real code of the executable and places it as data in a new executable that contains an unpacking routine. In previous work, we designed and implemented an anomaly detector based on PE structural characteristics and heuristic values, and we were able to decide whether an executable was packed or not. We stated that this detection system could serve as a filtering step for a generic and time consuming unpacking phase. In this paper, we improve that system applying a data reduction algorithm to our representation of normality (i.e., not packed executables), finding similarities among executables and grouping them to form consistent clusters that reduce the amount of comparisons needed. We show that this improvement reduces drastically the processing time, while maintaining detection and false positive rates stable.

Key words: malware, packer, anomaly detection, dataset clustering, computer security

1 Introduction

Malware (or malicious software) is the term used to define any software that has been written with malicious intentions to harm computers or networks and usually to obtain personal benefits in an illegitimate way. Malware authors' intentions have evolved in the last years. In the past, the intentions behind malware were fame and self-pride, but nowadays money is the main motivation. For this reason, efforts to bypass anti-virus tools have increased and thus, the power and variety of malware programs, together with their ability to overcome all kinds of security barriers [1]. One of the most commonly used techniques

is executable packing, which consists of cyphering or compressing the actual malicious code in order to hide it and evade signature scanning methods. Packed programs include a decryption routine that is first executed. This code extracts the real payload from memory and executes it. Some reports claim that up to an 80 % of the malware analysed is packed [2].

Traditional anti-virus systems apply signature scanning to identify malicious code. This technique has been also applied to detect executables protected with well known packers by scanning for certain byte sequences. PEID [3] is able to detect a wide range of well-known packers. Besides, Faster Universal Unpacker (FUU) [4] tries to identify the packer utilised to hide the original code and then applies custom unpacking routines designed and written for each packer. However, this approach has the same shortcoming as signatures for malware detection: it is not effective with unknown obfuscation techniques, nor with custom packers (i.e.,executable packing-unpacking algorithms exclusively designed for a certain malicious program). Actually, according to Morgenstern and Pilz [5], 35 % of malware is packed by a custom packer. This fact makes custom packers an important issue to consider.

Several approaches have been proposed to overcome this evasion technique. We can divide these approaches into static and dynamic approaches. Static approaches gather information about the employed packer without executing the sample, while dynamic unpacking approaches trace the execution of an executable and extract its protected code once unpacked. Normally, the samples are run inside an isolated environment like a virtual machine or an emulator [6].

Numerous dynamic unpackers try to identify the original entry point (i.e., where the execution jumps from the unpacking routine to the original code) by using heuristics. Once the execution flow reaches that point, the memory content is dumped to disk in order to obtain an unpacked version of the malicious code (e.g., Universal PE Unpacker [7] and OllyBonE [8]). Nevertheless, specific heuristics cannot be universalised to all the packers in the wild, since all of them work in very different manners. For instance, some packers use virtual instruction sets and attach an interpreter to the executable in such a way that the original code is never present in memory [9]. Other approaches decrypt frames of code before they are executed and once executed they encode them again. In this way, the whole malicious code is never loaded in memory at the same time [10].

In contrast, not so highly heuristic-dependent approaches have been proposed for generic dynamic unpacking (e.g., PolyUnpack [11], Renovo [12], OmniUnpack [13] and Eureka [14]). Nonetheless, these methods are time-consuming and cannot counter conditional execution of unpacking routines, a technique used for anti-debugging and anti-monitoring defense [15–17].

PE-Miner [18] extracts characteristics from the PE file header and builds classifiers that determine if an executable is malicious or not. PE-Probe [19], an improvement of PE-Miner, previously determines if the executable is packed and then applies a different classifier in each case. Perdisci et al. proposed in [20] a method for the classification of packed executables using certain heuristics

extracted from the PE structural data, as a previous step to the actual unpacking process.

In previous work [21], we proposed a method based on anomaly detection to filter executables that are not packed in order to avoid the processing overhead caused by generic unpackers. Our system calculated vectors composed of certain structural and heuristic features and compared the samples against a set of vectors representing not packed executables. If the sample was different enough, then it was considered as packed. Although the results obtained were significant enough to validate our method, the number of comparisons needed to analyse each sample was considerably high and consequently, it presented a high processing overhead.

In consideration of this background, we propose here an enhancement of our previous method [21], that applies partitional clustering to the dataset in order to reduce the number of vectors in the knowledge base. This improvement boosts the scalability of the system, reducing the processing time. The results obtained for the reduced dataset and the time saved by this technique reaffirms our initial hypothesis: A fast and efficient initial filtering step can improve generic and dynamic unpacking systems' performance by reducing the amount of executables to be analysed.

Summarising, our main contributions are:

- We propose a method for dataset reduction based on the partitional clustering algorithm Quality Threshold (QT) clustering, and generate reduced datasets of different sizes.
- We evaluate our system for different reduction rates, testing its accuracy results and comparing them to previous work.
- We prove that a unique sample synthetically generated from not-packed executables is sufficiently representative to implement an anomaly detection system without compromising accuracy results.

The remainder of this paper is organised as follows. Section 2 details our anomaly-based method. Section 3 describes the experiments and presents results. Section 4 discusses the obtained results and their implications, and outlines avenues for future work.

2 Method Description

The method described in this paper is based on our previous work, a packed executable detector based on an anomaly detection system [21]. Our approach consisted in the measurement of the distance from binary files to a set of binaries not packed. Any sample that deviates sufficiently from a representation of normality (not packed executables) is classified as packed. Contrary to supervised learning approaches, this method does not need a model training phase, and thus it does not require labelled packed executables, reducing the efforts needed to find and label a set of packed binaries. Nevertheless, it is necessary to compute as many distance values as executables in the not packed set.

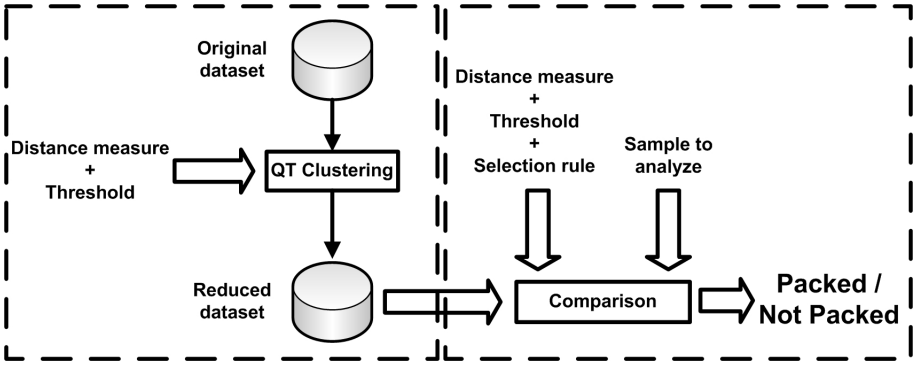


Fig. 1. Architecture of the proposed system. The QT clustering algorithm transforms the original dataset into a new reduced synthetic dataset. It requires 2 parameters: the distance measure and the threshold. The comparison system compares samples against the reduced dataset obtained, applying a distance measure, a distance threshold, and a selection rule. Finally the system is classifies the sample as packed or not packed.

In this paper, we improve the efficiency of our system by designing a data reduction phase capable of boosting the detector's scalability. Fig. 1 shows the architecture of our proposed system. The first objective of our method is to improve its efficiency by applying data reduction. The data reduction phase consists in the application of the QT clustering algorithm to the original dataset to obtain a reduced version that conserves the original dataset's features. In this way, the number of comparisons performed, and thus, the comparison time required for the analysis of each sample are much lower.

The second objective is to measure the precision of our system when the training set is incrementally reduced, in order to evaluate the trade-off between efficiency and accuracy. In addition, this data reduction approach enables us to test the performance of the system when a unique representation of a 'normal' executable is used, and to determine if it can to correctly classify packed and not packed executables.

2.1 Structural features

In previous work [21], we selected a set of 211 structural features of the PE executables from the conclusions obtained in previous research in the area [18–20]. Some features are extracted directly from the PE file header, while others are calculated values based on heuristics commonly used for detecting packers. Farooq et al. [18] and Perdisci et al. [20] used PE executable structural features, as well as heuristics like entropy analysis or certain section characteristics to determine if an executable is packed or not, as a previous step to a second analysis phase. To select the set of features we combined both points of view, structural characteristics and heuristics, and analysed their individual relevance

by statistical methods to determine how they impact on the classification of packed and not packed executables.

A second issue to consider in the feature selection was extraction time, since the system is aimed at becoming a filter to reduce the amount of executables analysed in dynamic environments that can be much more time consuming. Therefore, we selected a set of features that, unlike techniques such as code dissassembly, string extraction or n-gram analysis [20], do not require a significant processing time.

Features are classified into four main groups: 125 raw header characteristics [19], 33 section characteristics (i.e., number of sections that meet certain properties), 29 characteristics of the section containing the entry point (the section which will be executed first once the executable is loaded into memory) and, finally, 24 entropy values. For each feature, we calculated the Information Gain (IG) value [22]. IG provides a ratio for each feature that outlines its importance in order to classify a sample as packed or not packed. To calculate these weight values, we used a dataset comprised of 1,000 packed and 1,000 not packed executables.

- **DOS header characteristics (31).** The first bytes of the PE file header correspond to the DOS executable header fields. IG results showed that these characteristics are not specially relevant, having a maximum IG value of 0.23, corresponding to a reserved field, which intuitively may not be a relevant field. 15 values range from 0.10 to 0.16, and the rest present a relevance bellow 0.10.
- **File header block (23).** This header block (also named COFF header) is present in both image files (executable files), and object files. From a total of 23 characteristics, 14 have an IG value greater than 0, and only 2 of them have an IG value greater than 0.01: the number of sections (0.3112) and the time stamp (0.1618).
- **Optional Header Block (71).** This block is present in image files but not in object files, and contains data about how the executable must be loaded into memory. The data directory is located at the end of this structure and provides the address and size for very useful data structures. 37 features have an IG value over 0, but the most relevant ones are: the address of entry point (0.5111), the Import Address Table (IAT) size (0.3832) and address (0.3733) (relative to the number of imported DLLs), the size of the code (0.3011), the base of the data (0.2817), the base of the code (0.2213),the major linker version (0.1996), checksum (0.1736), the size of initialised data (0.1661), the size of headers (0.1600), the size of relocation table (0.1283) and the size of image (0.1243).
- **Section characteristics (33).** From the 33 characteristics that conform this group, 22 have an IG value greater than 0. The most significant ones are: the number of non-standard sections (0.7606), the number of executable sections (0.7127); the maximum raw data per virtual size ratio (0.5755) ($rawSize/virtualSize$, where $rawSize$ is defined as the section raw data size and $virtualSize$ is the section virtual size, both expressed in bytes),

the number of readable and executable sections (0.5725) and the number of sections with a raw data per virtual size ratio lower than 1 (0.4842).

- **Section of entry point characteristics (29).** This group contains characteristics relative to the section which will be executed once the executable is loaded into memory. 26 characteristics have an IG value greater than 0, from which 11 have a significant relevance: the characteristics field in its raw state (0.9757), its availability to be written (0.9715), the raw data per virtual size ratio (0.9244), the virtual address (0.7386), whether is a pointer to raw data or not (0.6064), whether is a standard section or not (0.5203), the virtual size (0.4056), whether it contains initialised data (0.3721), the size of raw data (0.2958) and its availability to be executed (0.1575).
- **Entropy values (24).** We have selected 24 entropy values, commonly used in previous works [20], from which 22 have an IG value greater than 0, and 9 have a relevant IG value: max section entropy (0.8375), mean code section entropy (0.7656), mean section entropy (0.7359), file entropy (0.6955), entropy of the section of entry point (0.6756), mean data section entropy (0.5637), header entropy (0.1680), number of sections with an entropy value greater than 7.5 (0.7445), and number of sections with an entropy value between 7 and 7.5 (0.1059).

After the extraction, every feature has to be normalised: each value is divided by the maximum value for that feature in the whole dataset. In this way, each executable is represented as a vector of decimal values that range from 0 to 1. Finally, each feature value is multiplied by its relevance IG value to obtain the final vector that will be used in the next steps. These weights are used to compute a better distance measure among samples and to reduce the amount of features selected, given that only 151 of them have an IG value greater than 0.

2.2 Data reduction

Dataset reduction is a step that has to be faced in very different problems that have to work with large datasets. In our work [21], the experiments were performed with a base of 900 not packed executables, which means that every sample analysed had to be compared 900 times to classify it as packed or not. Now, we propose a data reduction algorithm based on partitional clustering. Cluster analysis divides data into meaningful groups [23]. These techniques usually employ distance measures to compare instances in datasets to make groups with those which appear to be similar. We can identify several types of clustering, but most common ones are hierarchical clustering and partitional clustering. The first approach generates clusters in a nested style, which means that the clusters generated from the dataset are related hierarchically. In contrast, partional clustering techniques create a one-level (unnested) partitioning of the data points [23]. We are interested in this last technique to validate our initial hypothesis: it is possible to divide a big set of executables that represent normality (i.e., not packed executables) into a reduced set of representations.

```
input  : The original dataset 𝒱, the distance threshold for each cluster
         threshold, and the minimum number of vectors in each cluster
         minimumvectors
output: The reduced dataset ℛ
// Calculate the distance from each vector (set of executable
   features) to the rest of vectors in the dataset.
foreach {v_i | v_i ∈ 𝒱} do
    foreach {v_j | v_j ∈ 𝒱} do
        // If a vector v_j's distance to v_i is lower than the specified
           threshold, then v_j is added to the potential cluster 𝒜_i,
           associated to the v_i vector
        if distance(v_i,v_j) ≥ threshold then
            𝒜_i.add(v_j)
// In each loop, select the potential cluster with the highest
   number of vectors
while ∃𝒜_i ∈ 𝒜 : |𝒜_i| ≥ minimumvectors and ∀𝒜_j ∈ 𝒜 : |𝒜_i| ≥ |𝒜_j| and i ≠ j
do
    // Add the centroid vector for the cluster to the result set
    ℛ.add(centroid(𝒜_i))
    // Discard potential clusters associated to vectors v_j ∈ 𝒜_i
    foreach {v_j | v_j ∈ 𝒜_i} do
        𝒜.remove(𝒜_j)
        𝒱.remove(v_j)
    // Remove vectors v_j ∈ 𝒜_i from the clusters 𝒜_k remaining in 𝒜
    foreach {𝒜_k | 𝒜_k ∈ 𝒜} do
        foreach {v_j | v_j ∈ 𝒜_k and v_j ∈ 𝒜_i} do
            𝒜_k.remove(v_j)
// Add the remaining vectors to the final reduced dataset
foreach {v_j | v_j ∈ 𝒱} do
    ℛ.add(v_j)
```

Fig. 2. QT Clustering based dataset reduction algorithm.

Quality Threshold (QT) clustering algorithm was proposed by Heyer et al. [24] to extract useful information from large amounts of gene expression data. K-means is a classic algorithm for partitional clustering, but it requires to specify the number of clusters desired. In contrast, QT clustering algorithm does not need this specification. Concretely, it uses a similarity threshold value to determine the maximum radial distance of any cluster. This way, it generates a variable number of clusters that meet a quality threshold. Its main disadvantage is the high number of distance calculations needed. Nevertheless, in our case, this computational overhead is admissible because we only have to reduce the dataset once, (we employ an static representation of normality that only varies from platform to platform).

Our algorithm, shown in Fig. 2, is based on the concepts proposed by Heyer et al. [24], but it is adapted to our data reduction problem and it is implemented iteratively, instead of recursively.

Let $\mathcal{A} = \{\mathcal{A}_0, \mathcal{A}_1, ..., \mathcal{A}_n\}$ be the set of potential clusters. For each vector v_i in the dataset $\mathcal{V}$, there is potential cluster $\mathcal{A}_i \in \mathcal{A}$. A potential cluster $\mathcal{A}_i$ is composed of the set of vectors at a distance respect to v_i not higher than the *threshold* previously specified.

Once the potential clusters are calculated, we select the cluster with the highest number of vectors as a final cluster. Then, we calculate its centroid, defined as $c = x_1 + x_2 + \cdots + x_k/k$ where $x_1, x_2, \cdots, x_k$ are points in the feature space. The resultant centroid is added to the final reduced dataset. Afterwards, each vector v_j present in the selected cluster $\mathcal{A}_i$ is removed from the original dataset $\mathcal{V}$ (as they will be represented by the previously calculated centroid). Moreover, the potential clusters $\mathcal{A}_j \in \mathcal{A}$ associated to each vector v_j previously removed are also discarded. When there are not more clusters available with a number of vectors higher than the parameter $minimumvectors$, the remaining vectors in $\mathcal{V}$ are added to the final reduced dataset and the algorithm finishes and returns the resulting reduced dataset. The final result is a dataset composed of one centroid representing each cluster and all the vectors that were not associated to any cluster by the QT clustering algorithm (i.e., outliers).

2.3 Anomaly Detection

The features described represent each executable as a point in the feature space. Our anomaly detection system analyses points in the feature space and classifies executables based on their similarity. The analysis of an executable consists of 3 different phases:

- Extraction of the features from the executable file.
- Computation of calculated values.
- Measurement of the distance from the point representing the executable file to the points that symbolise normality (i.e., not packed executables) that conform the knowledge base.

As a result, any point at a distance from normality that surpasses an established threshold is considered to be an anomaly and thus, a packed executable. In this study, we have considered 2 different distance measures:

- **Manhattan Distance.** This distance between two points x and y is the sum of the lengths of the projections of the line segment between the points onto the coordinate axes:

$$d(x, y) = \sum_{i=0}^{n} |x_i - y_i|$$

where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.

– **Euclidean Distance.** This distance is the length of the line segment connecting two points. It is calculated as:

$$d(x, y) = \sum_{i=0}^{n} \sqrt{x_i^2 - y_i^2}$$

where x is the first point; y is the second point; and x_i and y_i are the ith component of first and second point, respectively.

In previous work [21] we noticed that the cosine similarity measure, (i.e., a distance measure computationally more expensive), does not produce better results.

Since we have to compute this measure with a variable number of points representing not packed executables, a combination metric is required in order to obtain a final distance value which considers every measure performed. To this end, we employ 3 simple rules:

– **Mean rule.** Select the average distance value.
– **Max rule.** Select the highest distance value.
– **Min rule.** Select the lowest distance value.

In this way, when an executable is analysed, the final distance value calculated depends on the distance measure and the combination rule selected.

3 Empirical Validation

To evaluate the performance of our method, we have conducted an experiment consisting of 2 phases: firstly, we reduce the set of vectors corresponding to not packed executables that represent normality, and secondly we start the anomaly detection step to measure accuracy and efficiency.

3.1 Experimental Configuration

The experiment designed to evaluate this system was performed using an executable collection comprising 1,000 packed and 1,000 not packed executables. Initially, 1,000 goodware executables were extracted from a common Microsoft Windows XP installation, and 1,000 malicious executables were gathered from the website VxHeavens [25]. All the executables where analysed with PEiD to assure that they were not packed. To generate the packed dataset, we employed 1,000 not packed executables (500 benign and 500 malicious) and we packed them using 10 different packing tools with different configurations: Armadillo, ASProtect, FSG, MEW, PackMan, RLPack, SLV, Telock, Themida and UPX. The not packed dataset was comprised of the remaining 1,000 executables.

The experimental method used was 10-fold cross validation [26], dividing the whole dataset into 10 different divisions. In this way, each fold is composed of 900 not packed executables as knowledge base and 1,100 testing executables, from which 100 are not packed and 1,000 are packed executables.

In order to test the dataset reduction algorithm proposed, 4 experimental configurations were selected for each distance measure. The threshold parameter values for our QT clustering based algorithm were selected by empirical observation. In particular, the thresholds for Manhattan distance were set as the double of the thresholds selected for Euclidean distance. While Manhattan distance sums the lengths of the projections of the line segment between the points onto the different coordinate axes, the Euclidean distance measures the line between two points, that is always shorter. Table 1 shows the results obtained in the process. Reduction ratio varies from 76.12% for Euclidean distance and threshold 0.25 to 99.88% for both Euclidean and Manhattan distance and an infinite threshold (in practice, this threshold is set to the maximum value allowed for a 64-bit double variable). The result obtained for this configuration is a unique centroid of the whole dataset that represents the arithmetic mean vector, or a single representation of normality. In this case, selection rules do not influence the final result because it is only performed one single comparison for each sample.

Table 1. Number of vectors that conform the reduced dataset for the different reduction parameters. The initial dataset is in all cases comprised of 900 not packed vectors.

Distance measure	**Quality threshold**	**% Average reduction**	**Number of vectors in each fold**									
			1	2	3	4	5	6	7	8	9	10
Euclidean	0.25	76.12%	217	215	214	218	216	208	216	219	206	220
	0.50	95.35%	44	42	39	44	41	44	42	42	39	41
	1.00	99.12%	7	8	8	8	8	8	8	8	8	8
	∞	99.88%	1	1	1	1	1	1	1	1	1	1
Manhattan	0.50	83.63%	153	148	150	149	151	145	147	143	141	146
	1.00	95.42%	41	41	41	43	40	42	40	43	38	43
	2.00	98.98%	7	10	11	10	8	10	9	10	8	8
	∞	99.88%	1	1	1	1	1	1	1	1	1	1

3.2 Efficiency results

During our experimental evaluation, we measured the times employed in each different phase. In this way, we can distinguish 3 different phases in the experiment:

- **Feature extraction and normalization.** The first step in the experiment was to extract the characteristics from the executables and to calculate values such as entropy or size ratios. Once extracted, these features were normalised to obtain a value ranging from 0 to 1 for each point in the feature space. This stage was performed in a virtual machine to keep all malware samples isolated from the host system and to prevent any possible infection. The virtual machine used was VMWare[27], hosted in an Intel Core i5 650 clocked

at 3.20 GHz and 16 GB of RAM memory. The guest machine specification was the following: 1 processor, 1 GB of RAM memory and Windows XP SP3 as operative system. Fig. 3 shows the time required by the feature extraction and normalization process for each file. This step took an average time of 28.57 milliseconds for each file analysed (93.66 μseconds/KB). Although the extraction of certain features such as PE executable header fields should require a similar amount of time for all the executables, some other values such as entropy are calculated using all the bytes present in the file: the higher the file size, the higher the time it takes to analyse it. Once extracted, feature vectors were saved into CSV files for further use.

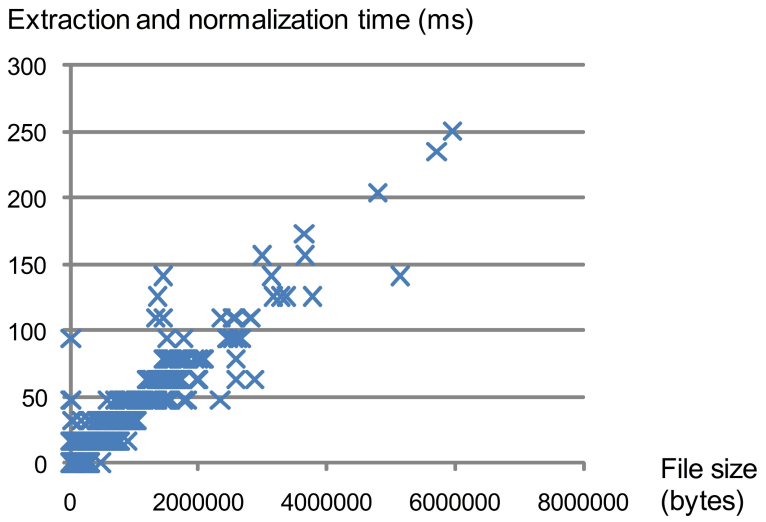


Fig. 3. Time required to extract and normalize the selected features from each executable file. The X axis represents the file size, expressed in bytes, while the Y axis shows the time taken by the extraction process, expressed in milliseconds.

– **Data reduction.** The second step was data reduction. In this phase, we reduced the original datasets, composed of 900 vectors for each fold, which were previously saved into CSV files. In this way, we used 8 different configurations to reduce each different dataset: Euclidean distance (0.25, 0.50, 1, and ∞) and Manhattan distance (0.50, 1, 2, and ∞). This stage was conducted directly in the host machine. Fig. 4 shows the time employed in the data reduction phase. It can be observed that times do not vary considerably for the different thresholds used for each distance measure. This occurs because the operations that take a higher processing overhead are the distance measure calculations, and the algorithm proposed in Fig. 2 calculates all the distances between points before starting the clustering step. Consequently,

the data reduction algorithm performs the same heavy calculations independently of the threshold specified. The average processing time consumed to reduce each fold is 97.83 seconds for Euclidean distance and 65.05 seconds for Manhattan distance. Note that this process, in spite of being very time consuming, is executed only once and does not interfere in the performance of the system.

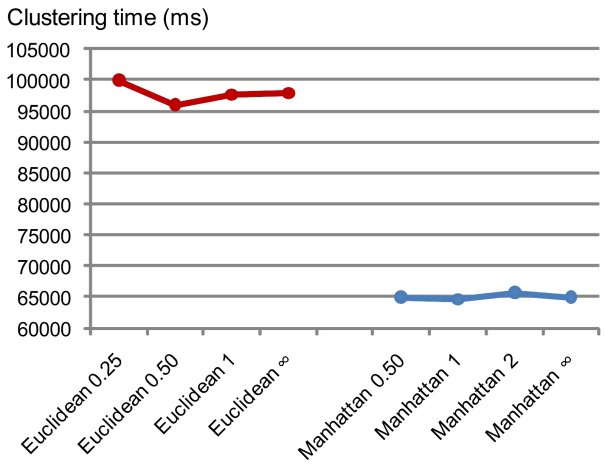


Fig. 4. Time required to reduce the original dataset composed of 900 not packed executables. The X axis shows the different experimental configurations selected for the data reduction step. The Y axis shows the time required by each clustering process performed, expressed in milliseconds.

– **Sample comparison.** Finally, the last step was the comparison of samples. For each experimental configuration employed in the data reduction stage, the samples under test (1,000 packed samples and 100 not packed samples) were compared against the reduced dataset. The total number of comparisons depends exclusively on the number of vectors present in the reduced datasets, so it is straightforward that the time employed in this step is inversely proportional to the threshold value used in the clustering process. As the previous phase, the sample comparison process was performed in the host machine. Fig. 5 shows the average time employed by the comparison step for each executable file. It can be noticed that the time required for comparison is lower when fewer vectors are utilised. For Euclidean distance the average comparison time varies from 25.62 ms for a 0.25 clustering threshold value, to 0.13 ms for an ∞ threshold (single vector representation). In the case of Manhattan distance, performance overhead is lower due to the simplicity of the calculations, and varies from 11.62 ms for a 0.50 clustering threshold value, to 0.08 ms for an ∞ threshold.

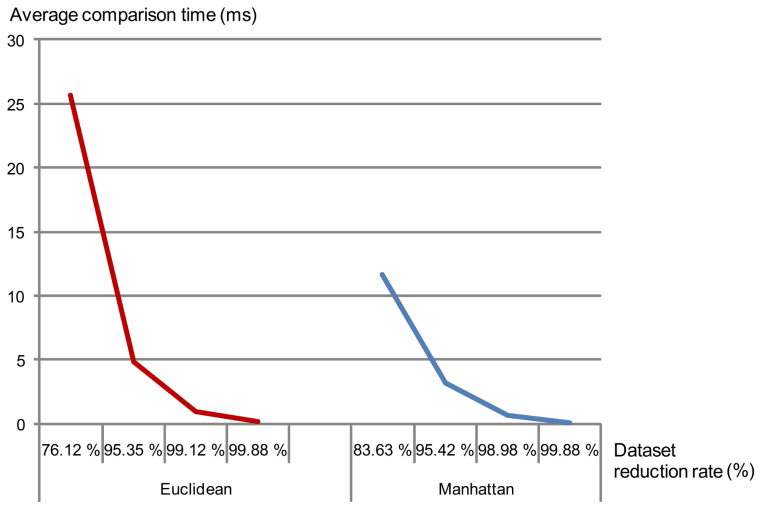


Fig. 5. Time required by the comparison phase for each reduced dataset. The X axis represents the reduction rate for each dataset once the clustering step was applied. The higher the reduction rate, the lower the number of vectors utilised for comparison. The Y axis represents the average comparison time for each executable file, expressed in milliseconds.

Subsequently, once the reduced datasets are obtained, the analysis of an executable file depends on extraction, normalization and comparison time. The times obtained highlight the conclusion that our system is able to compute between 1,000 and 2,000 executables in a minute.

3.3 Efficacy results

Hereafter, we extracted the selected features from the executables and reduced the dataset using the 2 different distance measures and 4 different threshold values (resulting into 8 different reduced datasets). Afterwards, we employed the same 2 distance measures and the 3 combination rules described in Section 2.3 to test the datasets and obtain a final measure of deviation for each testing executable. For each measure and combination rule, we established 10 different thresholds to determine whether an executable is packed or not, and selected the one which conducted to the best results in each case in terms of False Negative Rate and False Positive Rate.

We evaluated accuracy by measuring False Negative Rate (FNR), False Positive Rate (FPR), and the Area Under the ROC Curve (AUC).

In particular, FNR is defined as:

$$FNR(\beta) = \frac{FN}{FN + TP}$$

where TP is the number of packed executable cases correctly classified (true positives) and FN is the number of packed executable cases misclassified as not packed software (false negatives).

As well, FPR is defined as:

$$FPR(\alpha) = \frac{FP}{FP + TN}$$

where FP is the number of not packed executables incorrectly detected as packed while TN is the number of not packed executables correctly classified.

Finally, the AUC is defined as the area under the curve formed by the union of the points representing FPR and TPR for each possible threshold in a plot where the X axis represents the FPR and the Y axis represents the TPR. To calculate the AUC we used the points corresponding to the 10 thresholds selected. The lowest and the highest thresholds were selected in such a way that they produced a 0% FNR and a 0% FPR respectively. The rest of thresholds were selected by equally dividing the range between the first and the last threshold. The area under the curve formed by that points was calculated dividing it into 9 trapezoidal subareas and computing them independently:

$$AUC = \sum_{i=0}^{i=9} \left((x_{i+1} - x_i) \cdot y_i + \frac{(x_{i+1} - x_i) \cdot (y_{i+1} - y_i)}{2} \right)$$

Table 2 shows the obtained results. To simplify the results presented, we only show the performance that corresponds to the best possible threshold for each configuration. Despite Euclidean distance is more time consuming than Manhattan distance, both distance measures achieve similar results for each dataset configuration. In particular, our anomaly-based packed executable detector is able to correctly detect more than 99 % of packed executables while maintaining the rate of misclassified not packed executables lower than 1 %. As it can be observed, mean combination rule presents slightly better results both for FNR and FPR.

Nevertheless the most important issue to consider is data reduction. We propose 4 different data reduction configurations for each distance measure. We can observe in Table 2 that results slightly get worse when a higher threshold is applied (higher reduction rate). Fig. 6 shows 6 different plots for each distance measure and selection rule. Each plot shows 4 ROC curves corresponding to the 4 different reduced datasets. We can observe that in most of the cases the ROC curves show inferior results as the threshold increases (and thus, the number of vectors to compare with, decreases). Fig. 8 represents this evolution. In each case, as the number of vectors is reduced, the system looses accuracy. Nevertheless, when the executables are compared against the mean vector, the results obtained improve and in some occasions are even better than the ones achieved for the less reduced dataset (Euclidean distance with Max selector, in Fig. 6(c), and Manhattan distance with Max and Min selectors in Fig. 6(d) and Fig. 6(f)). This behaviour is more noticeable for Max and Min selectors, owing to the fact that this selectors are more sensitive to outlier vectors (i.e., vectors distant from

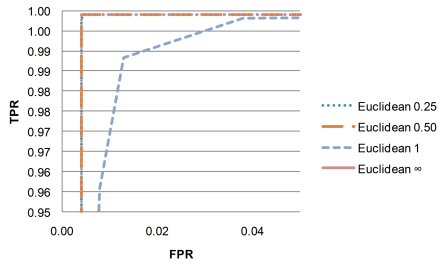
Table 2. Results for the different reduced datasets, combination rules and distance measures. Our method is able to detect more than 99 % of the packed executables while maintaining FPR lower than 1 %.

Dataset		Selection rule	Threshold	FNR	FPR	AUC
Euclidean	**Prev. work**	Mean	1.54000	0.00200	0.00800	0.99676
		Max	2,20000	0,00200	0,01400	0.99887
		Min	0.62000	0.00180	0.01400	0.99815
	0.25	Mean	1.36667	0.00100	0.00500	0.99820
		Max	2.06667	0.01860	0.01000	0.99874
		Min	0.58889	0.00370	0.00700	0.99845
	0.50	Mean	1.46667	0.00100	0.00400	0.99821
		Max	2.02222	0.01720	0.02100	0.99784
		Min	0.64444	0.00560	0.00800	0.99808
	1	Mean	1.42222	0.01170	0.01300	0.99786
		Max	1.97778	0.03420	0.02200	0.99383
		Min	0.70000	0.01090	0.03800	0.99448
	∞	-	1.33333	0.00100	0.00400	**0.99830**
Manhattan	**Prev. work**	Mean	4.05000	0.00160	0.01000	0.99819
		Max	7.40000	0.00820	0.01800	0.99808
		Min	1.55000	0.00180	0.00800	0.99914
	0.50	Mean	3.75556	0.00110	0.00500	0.99898
		Max	6.33333	0.00780	0.01900	0.99829
		Min	1.22222	0.00100	0.00400	0.99925
	1	Mean	3.87778	0.00110	0.00500	0.99921
		Max	6.33333	0.00890	0.01500	0.99850
		Min	1.36667	0.00200	0.00800	0.99858
	2	Mean	3.84444	0.00740	0.01700	0.99853
		Max	5.94444	0.06440	0.04700	0.98612
		Min	1.60000	0.01220	0.02300	0.99782
	∞	-	3.47778	0.00200	0.00500	**0.99901**

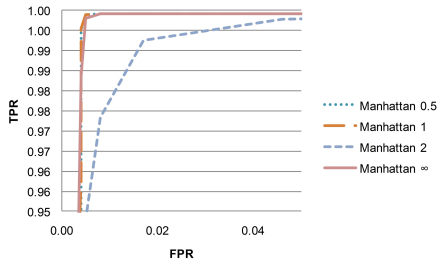
the normality representation) and can affect in a negative way as they alter the distance value obtained. Fig. 7 visually represents this effect. In the clustering process, 3 clusters are generated. Unfortunately, 2 clusters correspond to 2 outlier vectors that do not match with the majority of the not packed vectors. Arrows in Fig.7(b) show how the final distance value is very high for the not packed sample under analysis when Max selector is chosen. At the same time, the packed sample is misclassified if the Min selector is applied, due to the proximity of a not packed sample. In contrast, as Fig. 7(c) shows, mean vector is the representation of the whole dataset and the negative effects caused by distant vectors over the single centroid are smoothed by the rest of the vectors in the group.

The results obtained indicate that it is not necessary to renounce to accuracy in order to improve the efficiency of our anomaly detection approach. Although

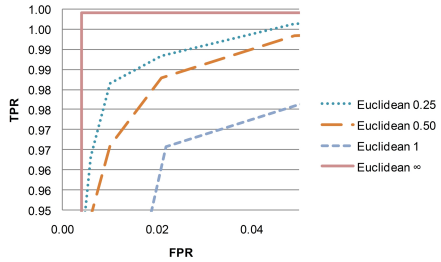
accuracy is reduced when a higher reduction rate is applied, when the samples are compared to a single representation (centroid of the group), results improve. This is the configuration that should be considered to implement an efficient and accurate packed executable filter.



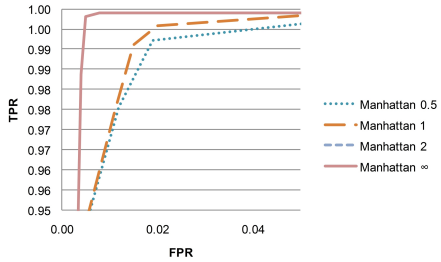
(a) ROC curve for the Euclidean distance and Mean selector.



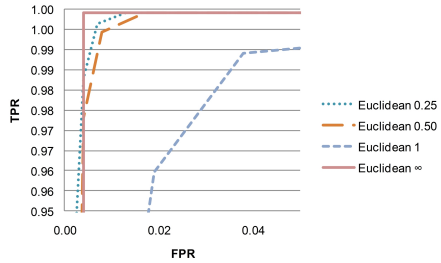
(b) ROC curve for the Manhattan distance and Mean selector.



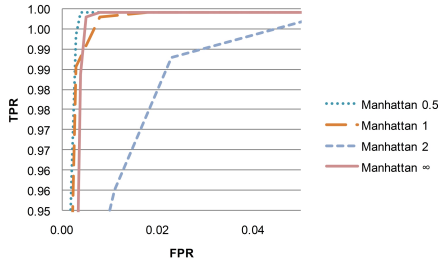
(c) ROC curve for the Euclidean distance and Max selector.



(d) ROC curve for the Manhattan distance and Max selector.

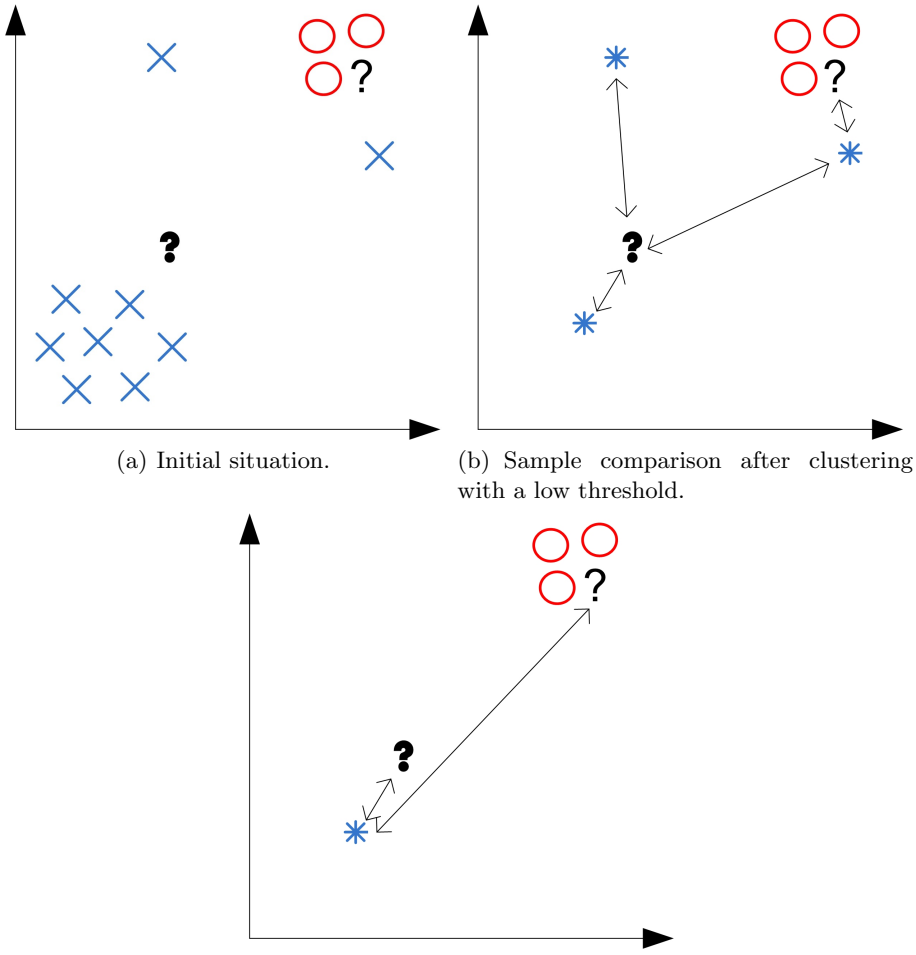


(e) ROC curve for the Euclidean distance and Min selector.



(f) ROC curve for the Manhattan distance and Min selector.

Fig. 6. ROC curves for the different experimental configurations. Each figure shows 4 ROC curves corresponding to the different reduced datasets. The scale selected for the X and Y axes has been reduced to 0.00 to 0.04 for X axis (false positive rate) and 0.95 to 1.00 for Y axis (true positive rate), to facilitate legibility and to represent precisely the differences among the different datasets tested. Unfortunately, the curve for Manhattan reduction with a threshold of 2 is out of the scope of the scale shown in the plot in 6(d). Note that in 6(a) and 6(b), some of the curves represented slightly overlap.



(a) Initial situation.

(b) Sample comparison after clustering with a low threshold.

(c) Sample comparison after clustering with an infinite threshold.

Fig. 7. Visual representation of the comparison process in two different scenarios. In 7(a) it is shown the initial situation. Crosses represent not packed executables, circles are packed files and question marks stand for samples to classify. In particular, bold question marks symbolise not packed vectors whereas flat ones are packed vectors. Finally, asterisks represent the centroid vectors generated for each cluster after the clustering process is performed. In 7(b) we show the vectors generated in the clustering process for a low threshold, and the effects over the distance measures obtained in the comparison phase. Similarly, 7(b) shows the unique centroid generated for the clustering with infinite threshold, and the distance measures obtained with this configuration.

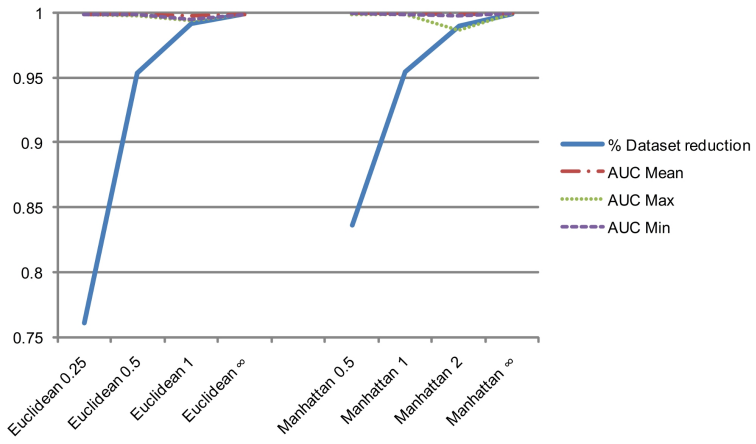


Fig. 8. Dataset reduction rate and accuracy achieved with each reduced dataset. The continuous line represents the increasing reduction rate (the higher the rate, the lower the number of samples in the reduced dataset), while the dotted lines represent the area under the ROC curve (AUC) obtained with each reduced dataset.

4 Discussion and Conclusions

The method proposed in this paper was focused on executable pre-filtering, in order to distinguish between packed and not packed executables. More specifically, it improves our previous work [21] by providing a new method for data reduction that boosts scalability in the anomaly detection process, enabling a much more efficient comparison of executable characteristics. As opposite to other approaches, anomaly detection systems do not need previously labelled data about packed executables or specific packers, as they measure the deviation of executables respect to normality (not packed executables). In contrast to signature scanning methods, this approach is packer independent.

Furthermore, accuracy results are not compromised by the dataset reduction process. It can be observed that the *AUC* varies slightly as the number of vectors in the dataset decreases. Nonetheless, when a single centroid vector is used, results are still sound, or even better than the ones obtained with no reduction. This fact brings us to the conclusion that it is possible to determine a single representation for not-packed executables, and that this single point is sufficiently representative to correctly classify executables as packed or not packed. Although anomaly detection systems tend to produce high false positive rates, our experimental results show very low values in all cases. These results, in addition to the time results presented in section 3.2 show that this method is a valid pre-process step for a generic unpacking schema. Since the main limitation of these unpackers is their performance overhead, a packed executable detector like our anomaly-based method with data reduction can improve their workload, acting as a filter.

Nevertheless, there are some limitations that should be focused in further work. First, the features selected by their IG value for the executable comparison are subject to attacks in order to bypass the filter. Malware writers can avoid certain suspicious field values or can program malware in such a way that the characteristics analysed by the proposed filter are more similar to the ones that correspond to not packed executables. For instance, DLL imports, section number, certain flags and patterns in executable headers can be modified with this intention. In addition, heuristic approaches can be evaded by using standard sections instead of not standard ones, or filling sections with padding data to unbalance byte frequency and obtain lower entropy values.

Secondly, the system is not aimed at identifying the packer used to protect the executable. However, this information is useful for the malware analyst and anti-virus systems in order to apply specific unpacking routines for each packer, avoiding the execution on time consuming dynamic analysis environments.

Finally, the dataset we employed was composed of executables protected with only 10 known packers. Some other packers, as well as custom packers, may implement some of the mentioned evasion techniques to bypass our filter.

In further work we will study different characteristics and alternative representations of executables to obtain an static detection system capable of providing more information about the packer used, if any. In addition, characteristics subject to attacks should be considered, in order to make the system resilient to new techniques employed by malware writers.

5 Acknowledgements

This research was partially supported by the Basque Government under a predoctoral grant given to Xabier Ugarte-Pedrero.

References

1. Kaspersky: Kaspersky security bulletin: Statistics 2008 (2008) Available online: `http://www.viruslist.com/en/analysis?pubid=204792052`.
2. McAfee Labs: Mcafee whitepaper: The good, the bad, and the unknown (2011) Available online: `http://www.mcafee.com/us/resources/white-papers/wp-good-bad-unknown.pdf`.
3. PEiD: PEiD webpage (2010) Available online: `http://www.peid.info/`.
4. Faster Universal Unpacker: (1999) Available online: `http://code.google.com/p/fuu/`.
5. Morgenstern, M., Pilz, H.: Useful and useless statistics about viruses and anti-virus programs. In: Proceedings of the CARO Workshop. (2010) Available online: `www.f-secure.com/weblog/archives/Maik_Morgenstern_Statistics.pdf`.
6. Babar, K., Khalid, F.: Generic unpacking techniques. In: Proceedings of the 2^{nd} International Conference on Computer, Control and Communication (IC4), IEEE (2009) 1–6
7. Data Rescue: Universal PE Unpacker plug-in Available online: `http://www.datarescue.com/idabase/unpack_pe`.

8. Stewart, J.: Ollybone: Semi-automatic unpacking on ia-32. In: Proceedings of the 14^{th} DEF CON Hacking Conference. (2006)
9. Rolles, R.: Unpacking virtualization obfuscators. In: Proceedings of the 3rd USENIX Workshop on Offensive Technologies.(WOOT). (2009)
10. Böhne, L.: Pandoras bochs: Automatic unpacking of malware. PhD thesis (2008)
11. Royal, P., Halpin, M., Dagon, D., Edmonds, R., Lee, W.: Polyunpack: Automating the hidden-code extraction of unpack-executing malware. In: Proceedings of the 2006 Annual Computer Security Applications Conference (ACSAC). (2006) 289–300
12. Kang, M., Poosankam, P., Yin, H.: Renovo: A hidden code extractor for packed executables. In: Proceedings of the 2007 ACM workshop on Recurring malcode. (2007) 46–53
13. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Proceedings of the 2007 Annual Computer Security Applications Conference (ACSAC). (2007) 431–441
14. Sharif, M., Yegneswaran, V., Saidi, H., Porras, P., Lee, W.: Eureka: A Framework for Enabling Static Malware Analysis. In: Proceedings of the European Symposium on Research in Computer Security (ESORICS). (2008) 481–500
15. Danielescu, A.: Anti-debugging and anti-emulation techniques. CodeBreakers Journal **5**(1) (2008) Available online: `http://www.codebreakers-journal.com/`.
16. Cesare, S.: Linux anti-debugging techniques, fooling the debugger (1999) Available online: `http://vx.netlux.org/lib/vsc04.html`.
17. Julus, L.: Anti-debugging in WIN32 (1999) Available online: `http://vx.netlux.org/lib/vlj05.html`.
18. Shafiq, M., Tabish, S., Mirza, F., Farooq, M.: PE-Miner: Mining Structural Information to Detect Malicious Executables in Realtime. In: Proceedings of the 12^{th} International Symposium on Recent Advances in Intrusion Detection (RAID), Springer-Verlag (2009) 121–141
19. Shafiq, M., Tabish, S., Farooq, M.: PE-Probe : Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. In: Proceedings of the 2009 Virus Bulletin Conference (VB). (2009) 1–10
20. Perdisci, R., Lanzi, A., Lee, W.: McBoost: Boosting scalability in malware collection and analysis using statistical classification of executables. In: Proceedings of the 2008 Annual Computer Security Applications Conference (ACSAC). (2008) 301–310
21. Ugarte-Pedrero, X., Santos, I., Bringas, P.G.: Structural feature based anomaly detection for packed executable identification. In: Proceedings of the 4^{th} International Conference on Computational Intelligence in Security for Information Systems (CISIS). (2011) in press.
22. Kent, J.: Information gain and a general measure of correlation. Biometrika **70**(1) (1983) 163–173
23. Kumar, V.: An introduction to cluster analysis for data mining. Computer Science Department, University of Minnesota, USA (2000)
24. Heyer, L., Kruglyak, S., Yooseph, S.: Exploring expression data: identification and analysis of coexpressed genes. Genome research **9**(11) (1999) 1106–1115
25. VX Heavens: Available online: `http://vx.netlux.org/`.
26. Kohavi, R.: A study of cross-validation and bootstrap for accuracy estimation and model selection. In: Proceedings of the International Joint Conference on Artificial Intelligence. Volume 14. (1995) 1137–1145
27. VMware: (2011) Available online: `http://www.vmware.com`.