Genetic Boosting Classification for Malware Detection

Alejandro Martín
Departamento de Ingeniería Informática
Universidad Autónoma
de Madrid, Spain
Email: alejandro.martin@uam.es

Héctor D. Menéndez
Computer Science Department
University College London,
United Kingdom
Email: h.menendez@ucl.ac.uk

David Camacho
Departamento de Ingeniería Informática
Universidad Autónoma
de Madrid, Spain
Email: david.camacho@uam.es

***Abstract*—In the last few years, virus writers have made use of new obfuscation techniques with the ultimate aim of hindering malware from being detected and making the detection more complicated. Strategies to reverse this trend involve executing potentially malicious programs and monitor the actions they perform in runtime, what is known as dynamic analysis.**

I. Introduction

Malware detection can be considered as a very complex task of great importance to determine whether a program has malicious or benign intentions. To discern the nature of a program, it is needed to analyse different data which can be extracted from it. There are two main approaches to obtain these data [10]: to follow a Static Analysis where the Malware detection process is performed based on the information contained in the executable or a Dynamic Analysis, consisting on executing the program and monitor the actions it performs [5].

This research is focused on Static Analysis, whose complexity has been magnified in the last few years because of the emergence of new concealment strategies such as packing, polymorphism or metamorphism [24]. These strategies aim to hide the malicious code inside the software. Combining this fact with the impossibility of manually classify all the programs that are generated daily by a security expert, powerful systems are required to automatize this task, but also providing a higher accuracy than a human. Due to the wide range of forms that Malware can adopt and the difficulties to differentiate it from Benign-ware, it is extremely complicated to arise accuracy rates close to 100%.

The model we present is based on a Static Analysis of API calls. Inside an executable, it is indicated a list of system calls needed. When the program is executed, the Operating System reads these data to allow it to use these API calls. We have designed a method that combines a Clustering algorithm with a Boosting Genetic algorithm composed of several classifiers in order to improve the detection. This idea is motivated by the different Malware families that currently exist and the different behaviours that they have. Malware from similar families will be closer in the "behavioural space" defined by the static API calls data. Therefore, the clustering algorithm will easily identify regions where these similarities are remarkable. Due to Malware aims to imitate benign-ware as a concealment strategy, we use the Genetic Boosting Algorithm to take advantage of both: a multiple learners approach (boosting system with several different classifiers), and the genetic optimization which estimates the relevance of the classifiers inside the regions through a voting system, using a multi-objective approach that also aims to reduce the false positive rate. To test the model proposed, we have used a benchmark database with information about the use of API calls by an important number of Malware and Benign-ware programs.

The remainder of the present publication is structured in several sections, starting with a study of the related work in the next section, followed by the description of the model proposed, which allows to detail the experimental setup and the experimentation in the next two sections and close with some conclusions.

II. Related Work

Malware detection has focused a broad research for more than a decade. Its growing complexity, the many different forms and intentions in can adopt, and also the development of new concealment strategies to hide or distort its true purposes necessitate the field to be constantly updated. On one side, Malware designers develop new techniques to obfuscate and complicate both a static and dynamic analysis. On the other side, new techniques are being developed to detect when a program has malicious intentions, even when this concealment strategies are used to hide its real purposes.

A. Malware obfuscation techniques

Obfuscation techniques are numerous and have been used in the last years creating different categories [24], each more complex than the last. The considered first category is Malware encryption, which is based on dividing executables into two parts, the decryptor loop and an encrypted main body [14]. The malign code is inside the main body, therefore only when the program is executed the malign code is decrypted, making it difficult to know its existence before. However, signature detection is a technique capable of detecting when a program contains malicious code, since that program will always have the same data (although part is encrypted).

Oligomorphic Malware tried to tackle this problem producing changes in its own decryptor loop, but signature detection was still able to detect this type with high precision, generating

a signature per decriptor loop. The next step was represented by polymorphic malware. In essence, polymorphism is based on mutating the binary code to generate a big number of different decyptor loops (around billions), whose signatures overcome the databases [13]. The most feasible form of detecting a polymorphic malware requires its execution (dynamic analysis). For these reasons, controlled environments called Sandboxes were developed to monitor Malware execution in a safe system. The following action to improve polymorphism was metamorphism, where the whole code changes modifying the program flow but keeping its semantic. The specific techniques beyond these types of Malware include Dead-Code Insertion to insert instructions that are not used, Register Reassignment, a technique that changes registers to change the code in appearance only, among many others [2], [19].

B. Malware detection using Machine Learning

There are two approaches to analyse and detect if a program is Malware: anomaly-based detection and signature-based detection [10], [22]. Anomaly-based detection techniques make a decision based on information about what a benign program should do. In contrast, signature-based detection techniques use information about programs already considered as Malware to classify new samples. In both approaches two techniques can be used: Static Analysis and Dynamic Analysis [17].

The former is based on the analysis of the executable without executing the program (thus preventing any damage), analysing the information included in the executable and its structure. In Dynamic Analysis, the information used to discern whether a program is malicious is extracted during its execution, for example the actions performed in the file system or the system calls performed. Both types of analysis have their pros and cons. Static Analysis is faster, since there is not need to execute the program and wait until an event occurs. Furthermore, its data can be accessed directly, although it can be modified by the developer in order to hide its true intentions. Dynamic Analysis is a more complicated and time-consuming process as it must execute the program as well as analyse it. Furthermore, some malicious programs are able to detect when they are inside a virtual machine and, hence, they behave as benign-ware hiding themselves. In these cases, Machine Learning has proven to be successful at designing Malware detection tools even when the most modern obfuscation techniques are used [16], [7], [15].

There are two main Machine Learning perspectives for Malware detection which are related to two classical Machine Learning approaches: Clustering, with the aim of finding groups with common characteristics blindly, and Classification, training a system with datasets where each instance is already labelled in order to discriminate new samples in the future. Both methods have also been combined to improve the accuracy [16]. This research follows this research line, combining these two approaches to compose a complex model able to achieve high accuracy levels using Static Analysis.

Within the scope of Static Analysis, it can be addressed using different methods which are related to the specific data extracted from the executables [12], [20]. For example, byte sequences or Opcodes can be used to find patters or also the information contained in the header. Inside this part of an executable, it is written a list of API calls, needed to load the program by the Operating System, information that we use in the model presented.

An example of the use of static API calls combined with Machine Learning algorithms can be found in [18], where authors use a RandomForest classifier to improve the detection accuracy. We have used this benchmark dataset as a starting point to improve these results using Genetic Boosting. Other methods which use API calls to detect whether a program is Malware or not involve the extraction of N grams [21] or the generation of API calls groups [8].

C. Evolutionary Computation and Genetic Algorithms

Within the scope of Artificial Intelligence, there is a specific field called Evolutionary Computation, focused on giving solutions to optimization problems using a metaheuristic search and based on the Darwin's rules. The type of algorithms which are beyond its scope form the so-called Evolutionary Algorithms. In addition, these algorithms can be subdivided in different approaches: Genetic Algorithms (GA), Multi-Objective Optimization (MOGA), Co-Evolutionary Algorithms (COEA) and Evolutionary Strategies (ES), among others [1]. This research focuses on using a Genetic Algorithm (which are inspired from the natural selection) for Multi-Objective Optimization to lead the Boosting process, following the SPEA2 algorithm [26].

D. Genetic Boosting

The origins of Boosting date back to the late eighties [25]; AdaBoost is considered the most important implementation. The main idea is to combine multiple learners to specialize the learning process [4]. While existing evidence show the difficulty of achieve high accuracy rates on Malware classification, it is justified the use of more complex classification models, as a Boosting algorithm, to overcome these rates. In this work, we apply Genetic Algorithms (GA) as a method for Boosting optimization. As the literature shows, they can be used to adjust the confidence factors of the classifiers and also the number of weak classifiers [3], [6], improving the accuracy but also decreasing the classification time. Further, Boosting has been applied to Malware classification [11].

III. Genetic Boosting Classification Model for Multiple Regions

The model we propose consists of different steps, as it can be seen in Fig. 1 and Algorithm 1, which contain a general scheme of the architecture and the step by step process. Each sample of the input data states a list of static API calls performed by a Malware or Benign-ware executable according to the header in their binary files. This makes it possible to create a n-dimensional space where each coordinate

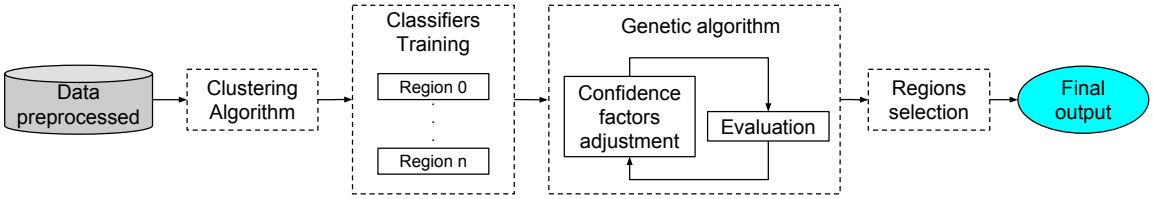

Fig. 1. General diagram of the model.

defines the number of invocations of a specific API call or a group of them. This space is divided in regions, in order to create groups of similar samples, separating them from others with a completely different behaviour. This division allows to create independent and specific classification models in each region, taking particular behaviours into consideration, aiming to improve the final accuracy of the model in contrast with solutions that consider the space as a whole.

A clustering algorithm takes the Training dataset to divide the space and returns a list of centroids, one per region, which allows to allocate the samples corresponding to the Validation and Test dataset to the closest region. In order to create an accurate model in each region, a set of classifiers is trained, which are later combined with a Boosting process leaded by a Genetic Algorithm. The final output is generated according to a parameter called *Accuracy Threshold*, which allows to ensure a minimum accuracy when an output is produced.

Algorithm 1 Local Genetic Boosting Classification Algorithm

1: **procedure** MAIN($nReg, threshold$)
2: $regsTrain, centroids \leftarrow$ KMEANS($Train, nReg$)
3: $regsVal \leftarrow$ ALLOCATE($Val, centroids$)
4: $regsTest \leftarrow$ ALLOCATE($Test, centroids$)
5: $classifiers \leftarrow$ TRAIN($regsVal$)
6: Generate solution per each classifier and region:
7: $predictions \leftarrow$ EVALUATE($regsVal$)
8: $\mathcal{P} \leftarrow$ GENBOOSTING($predictions$)
9: $ind \leftarrow$ SELECTINDIVIDUAL($\mathcal{P}$)
10: $regsSelected \leftarrow$ SELECTREGS($threshold$)
11: EVALUATESOLUTION($ind, regsSelected$)
12: **end procedure**

A. Data Partition

Given a dataset composed of instances whose attributes determine the number of uses of a specific API call, it is divided into three different datasets to perform different tasks in the algorithm. These parts are the Training dataset, the Validation dataset and the Test dataset. The reasons for creating a Validation dataset rely on the necessity of evaluating the classification models of each region with new examples and adjust accordingly their weights in the Boosting process. The Test dataset is used at the end of the execution to evaluate the whole classification model.

B. Clustering algorithm

As it has been described earlier, the model takes advantage from the use of different regions to create different specialised classification models and thus improving the final accuracy. In order to create these regions, it uses the K-means algorithm, where the Euclidean distance serves as a metric to calculate distances between points. To ensure that the regions generated have enough samples for the training process, all of them must be composed of at least 3 samples. If this condition is not satisfied when the process is over, the clustering algorithm is executed again with a new random initialisation until a valid solution is found.

C. Classifiers Training

Once the clustering algorithm finishes, a set of classifiers is trained in each region. The combination of multiple learners allows to ensure a high accuracy in each region, without the constraints of a single model with a global approach. The execution of the classification algorithms was performed with the Weka library [9], which is written in Java. The reasons for choosing this library lie in the variety and number of classifiers provided. In each region, 17 different classifiers were trained, involving decision trees like RandomForest or RandomTree, Bayesian algorithms like NaiveBayesMultinomial, regression models like the Logistic algorithms or meta-classifiers like RandomCommittee.

D. Genetic Boosting approach

This combination of classifiers can be addressed with a Boosting process, where all the classifiers work together to complement their outputs and provide a more accurate solution.

In a Boosting approach, each classifier is associated with a confidence factor, which is related with its Accuracy level. To explain how it operates, we focus on a classical boosting algorithm: AdaBoost, where these factors are decided evaluating a set of samples sequentially with each classifier, increasing its confidence factor if the sample is correctly classified or decreasing it otherwise. The final output for each instance is calculated evaluating it with all the classifiers and applying their factors to their outputs, i.e., each possible label will have a value associated as the sum of the confidence factors of the classifiers that returned that label. At the end, the label with the higher cumulative value is defined as the output for the particular instance. In contrast, the Boosting process of our model is addressed by a Genetic Algorithm. This decision is

due to the optimization power of Genetic Algorithms and also to their capacity to generate varied solutions.

E. Genetic Boosting

The previous step divided the space in regions and set the classifiers inside the regions. In this step we need to combine the information of the classifiers in a sensible way, in order to make it complementary. For this reason, we have designed the Genetic Boosting Algorithm, which performs this task taking advantage of GAs optimization abilities. We use a GA to define a confidence factor for each classifier. In the GA each individual represents a possible assignment of weights (or confidence factors) to classifiers. Algorithm 2 shows the pseudo-code of how it operates. The first individuals are randomly initialised and, over the generations of the GA, they evolve by applying them the classical Genetic Algorithms operations: Selection, Reproduction, Crossover and Mutation. In this evolution, there are two objectives to pursue: to maximize the accuracy and to minimise the number of false positives. The evaluation function, detailed in Algorithm 3 describes how these two values are calculated. The predictions are compared with the real labels in the validation dataset to calculate the accuracy and the number of false positives. The prediction procedure is explained below.

At the end, it is expected to arise solutions where the classifiers are successfully combined to achieve high accuracy rates. The GA has been implemented with the ECJ library written in Java. Specifically, the multi-objective SPEA2 algorithm was selected to evolve the population.

Algorithm 2 Genetic Boosting

```
1:  procedure GenBoosting(predictions)
2:      P ← RandomPopulation(size, N * R)
3:      for i ← 0, Generations do
4:          for each p ∈ P do
5:              obj1, obj2 ← calcFitness(p, predictions)
6:          end for
7:          Apply crossover, mutation and selection on P
8:      end for
9:      return P
10: end procedure
```

1) Encoding: The individuals (Figure 2) have $n \cdot k$ positions (the number of regions and the number of classifiers respectively), which represent the confidence factors of the classifiers for all the regions. Each region R_i is represented sequentially in the individual and it is composed of a list of values defining the confidence factor R_iW_j for each of the k classifiers of that region. Each allele, which represents the confidence factor of a specific classifier, is a decimal number in the range of 0 to 1.

2) Operations: The operations performed by the Genetic algorithm are four:

- **Selection**: We have performed an Elitism selection where the l best individuals of a population are passed to the next generation.

Algorithm 3 Calculate Fitness

```
1:  procedure CalcFitness(indv, predictions, centroids)
2:      sumCorrect, falsePositives ← 0
3:      for i ← 0, numSamples do
4:          prediction ← Classify(indv, val[s], centroids)
5:          if prediction == val[s] then
6:          sumCorrect++
7:          if val[s] == 0 and prediction == 1 then
8:          falsePositives++
9:      end for
10:     acc ← sumCorrect/S
11:     return acc, falsePositives
12: end procedure
```

- **Reproduction**: The chromosomes which are chosen for the reproduction are elected using a Tournament operator.
- **Crossover**: The crossover applied is a uniform crossover, where each gene is crossed with a defined probability.
- **Mutation**: A random based mutation where any allele is randomly assigned to a different decimal value.

Algorithm 4 Classify sample

```
1:  procedure Classify(factors, sample, centroids)
2:      region ← getRegion(sample, centroids)
3:      predictions ← Predict(sample, region)
4:      malware ← factors[region] * predictions
5:      benign ← factors[region] * Swap01(predictions)
6:      prediction ← max(malware, benign)
7:      return prediction
8:  end procedure
```

3) Fitness Function and Evaluation : While the classifiers are trained with the Training dataset, its evaluation is performed with the Validation dataset. This is due to the necessity of using new data to evaluate each classifier, but also because the accuracy reached by the classifiers in the training dataset is close to 100%, avoiding to find differences between them and obtain different confidence factors.

The dual Fitness Function tries to maximize on one hand the validation accuracy and also to minimise the number of False Positives. The last one is a very important parameter in evaluating a Malware detection system, which means to avoid misclassification of benign-ware as Malware. This factor is related with the feasibility of the model to be used in a real environment.

These two Fitness values are calculated comparing the prediction produced by the model for each sample in the Validation dataset and the real labels. The prediction procedure for one instance is shown in Figure 2 and its pseudo-code is written in Algorithm 4. First, it is needed to access the region R_i where the sample is placed and evaluate the instance with each classifier C_j of that region. On the other hand, the weights of the classifiers of R_i are extracted from the individual, located in WR_i (which consists of different positions, one for each classifier). Then, for each label is calculated a

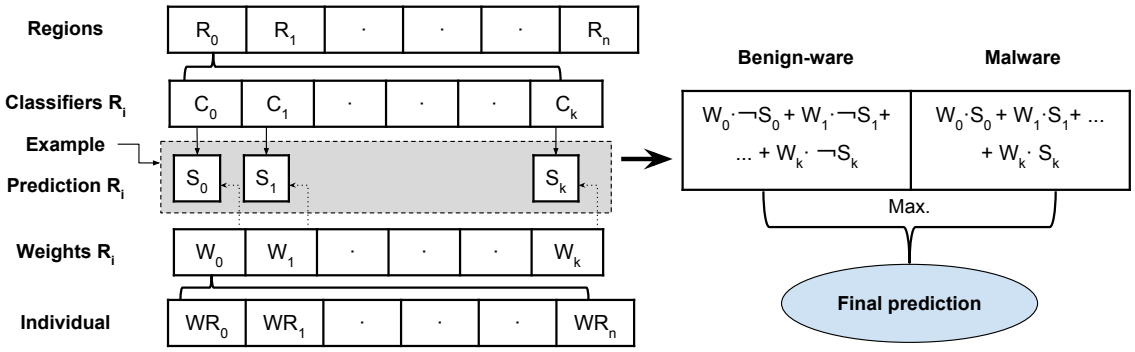

Fig. 2. Label prediction procedure

sum of weights W_j of the classifiers that predicted that label. For example, in order to calculate the total weight gained by the Malware label, the output of each classifier (which will be "1" for Malware samples) is multiplied by its weight. The sum of these values is equals to the sum of the weights of the classifiers that resulted in "1". Calculating the weight gained by the Benign-ware label requires changing zeros to ones and ones to zeros in the outputs to only take into account those which are equals to "0." The label with the maximum sum is the final prediction of the algorithm.

4) Solution selection: A multi-objective Genetic Algorithm produces several solutions. Those ones that are not dominated by any other form the Pareto frontier. We choose only one solution to finally test the model, taking into account primarily the accuracy, which allows to compare with other algorithms which are also focused on maximizing this objective.

5) Regions selection: The creation of regions will produce different classifications models with different accuracy levels. If the performance of each region is quantified, it is possible to know if the output for a new sample is reliable depending on the region where it is placed. To achieve this, each region is evaluated with the instances of the Validation dataset belonging to that region, and the accuracy reached can be used to measure its reliability. Then, it is possible to only take into account those regions that exceed a specific threshold, thereby discarding those that are expected to produce poor results.

When classifying a new instance, first it will placed in the region concerned. If the accuracy in the Validation dataset for this region was above the threshold, it is classified with its classification model. If that region does not reach the threshold, no output is produced. This procedure will reduce inevitably the number of instances which are given a label, but the accuracy will increase for those that are classified.

If no output is delivered, the instance in question cannot be classified reliably and a more detailed analysis (for example a Dynamic Analysis) is needed to produce an output accurately.

IV. Experimental setup

This section describes the dataset used, the preprocessing techniques which have been applied over the data and also the parametrization of the Genetic Algorithm and the classifiers. All of this will allow to explain the experimentation and the results obtained in the next section.

A. Datasets

This work is focused on a Static Analysis of Windows binaries. We have used a public dataset [18] with information of 31,869 already detected as malicious programs and 2,951 checked as benign. These data are basically extracted from tables (included in the binary file) that inform the DLLs it needs, and also the specific calls which will be invoked in the execution.

The dataset comprises 44,605 attributes, each one related with a particular API call. The binary value of the attributes indicate if the executable imports each API call. This information needs a preprocessing step before being used to train a classification model. The authors of the dataset generated three different set applying different preprocessing techniques: DLL, Cat+ and Close datasets.

DLL dataset reduces the number of attributes grouping them by the DLL they belong to. Close dataset uses the Clospan algorithm [23] to find close API calls that can be combined to decrease the number of dimensions. Finally, the Cat+ dataset creates a new form to represent each sample. The 95 first attributes are related with a taxonomy defined by Microsoft where each column is a category representing a subset of the API calls. The next 955 attributes are a list of the top discriminative API calls, obtained using the Fischer score. The number of samples of this dataset is reduced to 11,987.

Cat+ dataset is the most suited to train a classification model because it provides more information that the Close and DLL datasets. The authors of the dataset assert that they achieve a 98% Accuracy using a Random Forest classifier with a parameter of 100 trees. However, this high percentage is biased due to two factors. First the dataset is unbalanced, only 24% of the samples are benign, so allocating the same label to the whole dataset will provide high, but fictitious, accuracy values. Second, we found that the 45% of the samples was duplicated. Although this can be attributed to the fact that different samples have a very similar behaviour, the existence of the same sample in the training and test datasets leads to inconclusive results. In the next section it will be described the data processing performed to address these two problems.

B. Data Preprocessing

As it has been described in the Datasets Section, the Cat+ dataset is the most appropriated to train a classification model, since it provides further information about each samples, leading to the best results.

First, all the duplicated samples were removed. This decreased the amount of samples to 6,647. In a second step, a resample filter was applied over the examples pursuing a uniform distribution of the labels, obtaining a new balanced dataset with around 2100 samples of each label.

C. Genetic Algorithm Parametrization

The execution of a Genetic Algorithm involves choosing a set of parameters. After a Grid Search, we decided to create an initial population of 120 individuals, which allows to have enough diversity to cover a large part of the search space. These population evolves through 40 generations, which is a number that enables the algorithm to always converge. The crossover and mutation probabilities were fixed to 20% and 10% respectively, and the elitism was defined to take the best 10 individuals in each generation.

D. Classifiers

All the classifiers used, implemented in the Weka library, take their default parameters, but the seed (in those algorithms that applies) was changed every time that the algorithm was invoked.

V. Experimentation

This section addresses a series of experiments in order to test the proposed model. At the same time, it will be analysed how Windows executables behave when they are separated according to the use of system API calls they perform and if the approach taken allows to improve the results compared to other research.

A. Correlation between Validation and Test dataset

Having several regions containing different samples implies the emergence of different behaviours. Therefore, each classification model will behave in a different form in each region of the space, generating different accuracy levels. A previous assessment of each region with the results obtained from the boosting algorithm in the validation dataset, allows to reject those that are below a threshold. For this purpose, it is necessary that the accuracy in the validation dataset relates this value for the test dataset, in order to anticipate how the model will behave with unseen examples.

Figure 3 shows how the accuracy evolve in both datasets in four executions with 7, 13, 20 and 26 regions. If a very small number of elements is placed in an specific region, the accuracy reached here will not ascertain a similar performance in test samples; therefore, a minimum threshold of 20 elements in each region has been established, omitting these regions. In almost all regions for all the executions, both values are on a similar trend, where all the points are very close between themselves. Increasing the number of regions reduces the number of regions that exceeds the threshold.

TABLE I
Results in validation and test dataset depending on the threshold used

Validation dataset			Test dataset		
ACC thold	**Regions over thold**	**Acc in regions selected**	**Acc in regions selected**	**Instances in selected regions**	**False Positives**
90%	7	93.03%	92.85%	74.88%	1.14%
91%	4	95.87%	96.83%	35.49%	0.3%
92%	4	95.87%	96.83%	35.49%	0.3%
93%	2	97.48%	97.66%	25.38%	0.1%
94%	2	97.48%	97.66%	25.38%	0.1%
95%	1	100.00%	100.00%	15.27%	0%
96%	1	100.00%	100.00%	15.27%	0%
97%	1	100.00%	100.00%	15.27%	0%
98%	1	100.00%	100.00%	15.27%	0%
99%	1	100.00%	100.00%	15.27%	0%
100%	1	100.00%	100.00%	15.27%	0%

These results prove that the accuracy reached in the Validation dataset is a reliable factor to predict the accuracy in samples belonging the test dataset, enabling to discard those regions which are expected to lead to poor results.

B. Model evaluation

As previously stated, the model proposed filters the regions according to the accuracy reached in the Boosting process, discarding those with no promising results. To put this into practice, different accuracy thresholds have been established and the regions producing results below this value are flagged. The main idea of this process is to increase the accuracy but decreasing in the other hand the number of instances which are classified. Figure 4 shows how the number of instances classified decreases when the accuracy threshold raises in an execution with 12 regions. In order to classify the whole set of instances, a low threshold must be selected. In contrast, if it is required the maximum achievable accuracy, the threshold should be fixed close to 95% or 100%, which implies classifying 15% of instances.

Another important parameter that affects the percentage of instances classified or, in other words, the amount of instances that are classified over a minimum threshold is the number of regions. Figure 5 illustrates this, where different thresholds above 93% and the results from executions with 2 to 30 regions are presented (which is shown by the colour of the bars). Again, the percentage of instances classified decreases with a higher threshold. However, the number of regions affects significantly to the results. With a high threshold, a high number of regions must be selected, while when the threshold falls, a low number of regions produce better results.

A second very important aspect shown in the plot is the existence of bars with a drastic change compared to their neighbours. This is due to the clustering algorithm, because of its random initialisation, but also because the samples distribution in the space. An increment of just 1 cluster can cause a completely different division of the space, thereby

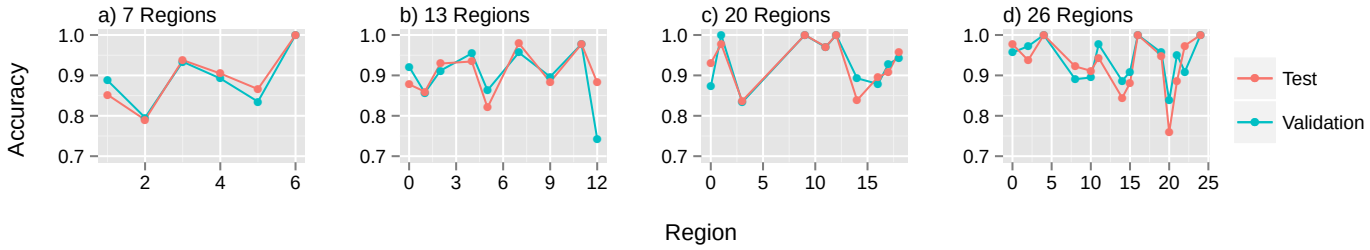

Fig. 3. Correlation between Validation and Test dataset with different number of regions

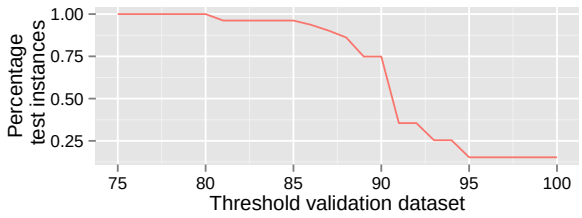

Fig. 4. Percentage test instances classified depending on the threshold used

affecting the final results. In this respect, the number of regions will be subjected to the output of the clustering algorithm.

The final results in the test dataset depending on the threshold used are shown in Table I. As it can be seen in the table, the method presented reaches a 100% accuracy over the 15% of the test instances.

Regarding the number of False Positives, the only region that was selected with a threshold above 95% did not produced any False Positive since the accuracy was 100%. With a threshold fixed 90%, only a 1.14% of the benign samples was labelled as Malware.

C. Discussion

As mentioned in the previous paragraphs, a local approach focusing on the specific characteristics of each region of the space helps to improve the accuracy if it is compared with a global approach. At the same time, creating a different classification model depending on the region of the space entails the existence of completely different accuracy levels. We consider that if one of these regions is expected to produce poor results, it is better to not deliver any output, since a deeper analysis is needed to classify the instances with enough confidence. The decision of what is considered as enough confidence lies in many factors, our models allows to modify this value and produce therefore different results.

The number of regions is also a key parameter in the algorithm because it defines the degree of granularity. A high value will generate more specialised regions and classifiers, while a lower value will be associated with a greater generalisation level. When the number of clusters increases, regions with very high accuracy level arises, allowing to classify instances with high reliability, as it was shown in Figure 5.

VI. Conclusion

The new obfuscation techniques developed recently make it necessary to develop new complex techniques able to reach high accuracy rates. The model we present takes a multi-region approach to create specialised classification models focused on particular types of Malware. In each region, a strong classifier is built combining several classifiers with a Boosting process guided by a Genetic Algorithm. In a second step each region is evaluated and those that are predicted to have a low accuracy level are discarded. With this technique, it can be adjusted an accuracy threshold in order to only deliver an output if it is reliable. The results of the experiments demonstrate that the model is able to achieve a 100% accuracy in a subset of the input data. Our future work involves increasing the percentage of instances classified, by combining the method presented with a Dynamic Analysis to be used with those instances that could not be classified.

Acknowledgment

This work has been supported by the next research projects: TIN2014-56494-C4-4-P, CIBERDINE S2013/ICE-3095, SeMaMatch EP/K032623/1 and Airbus Defence & Space (FUAM-076914 and FUAM-076915).

References

[1] Thomas Bäck. *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford university press, 1996.

[2] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[3] Zhang Dezhen and Yang Kai. Genetic algorithm based optimization for adaboost. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 1, pages 1044–1047. IEEE, 2008.

[4] Pedro Domingos. A few useful things to know about machine learning. *Communications of the ACM*, 55(10):78–87, 2012.

[5] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys (CSUR)*, 44(2):6, 2012.

[6] AhmedSharaf ElDen, MA Mustafa, Hany M Harb, and AbdelH Emara. Adaboost ensemble with simple genetic algorithm for student prediction model. *International Journal of Computer Science & Information Technology*, 5(2):73–85, 2013.

[7] Ivan Firdausi, Charles Lim, Alva Erwin, and Anto Satriyo Nugroho. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pages 201–203. IEEE, 2010.

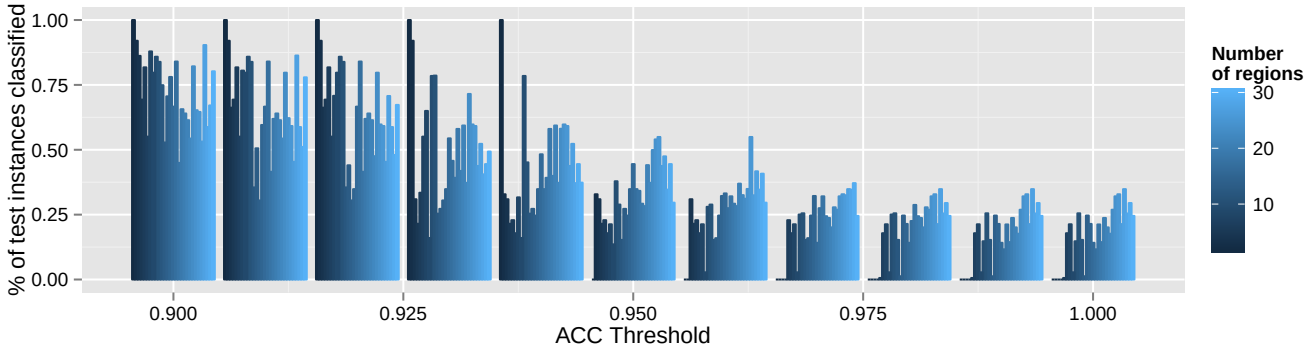

Fig. 5. Percentage instances classified with a minimum accuracy in different execution for different number of regions

[8] A. Fujino, J. Murakami, and T. Mori. Discovering similar malware samples using api call topics. In *Consumer Communications and Networking Conference (CCNC), 2015 12th Annual IEEE*, pages 140–147, Jan 2015.

[9] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian H Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[10] N Idika and A P Mathur. A survey of malware detection techniques. *Purdue University*, 2007.

[11] Pratiksha Natani and Deepti Vidyarthi. Malware detection using api function frequency with ensemble based classifier. In *Security in Computing and Communications*, pages 378–388. Springer, 2013.

[12] Hiran V Nath and Babu M Mehtre. Static malware analysis using machine learning methods. In *Recent Trends in Computer Networks and Distributed Systems Security*, pages 440–450. Springer, 2014.

[13] Philip O'Kane, Sakir Sezer, and Keiran McLaughlin. Obfuscation: The hidden malware. *Security & Privacy, IEEE*, 9(5):41–47, 2011.

[14] Babak Bashari Rad, Maslin Masrom, and Suhaimi Ibrahim. Camouflage in malware: from encryption to metamorphism. *International Journal of Computer Science and Network Security*, 12(8):74–83, 2012.

[15] Konrad Rieck, Thorsten Holz, Carsten Willems, Patrick Düssel, and Pavel Laskov. Learning and classification of malware behavior. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[16] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[17] Imtithal A Saeed, Ali Selamat, Ali MA Abuagoub, and Salman Bin Abdulaziz. A survey on malware and malware detection systems. *analysis*, 3(10):13–17, 2013.

[18] Ashkan Sami, Babak Yadegari, Naser Peiravian, Sattar Hashemi, and Ali Hamze. Malware detection based on mining API calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing - SAC '10*, page 1020, New York, New York, USA, March 2010. ACM Press.

[19] Sebastian Schrittwieser and Stefan Katzenbeisser. Code obfuscation against static and dynamic reverse engineering. In *Information Hiding*, pages 270–284. Springer, 2011.

[20] Asaf Shabtai, Robert Moskovitch, Yuval Elovici, and Chanan Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1):16–29, 2009.

[21] Dolly Uppal, Roopak Sinha, Vishakha Mehra, and Vinesh Jain. Malware detection and classification based on extraction of api sequences. In *Advances in Computing, Communications and Informatics (ICACCI, 2014 International Conference on*, pages 2337–2342. IEEE, 2014.

[22] P Vinod, R Jaipur, V Laxmi, and M Gaur. Survey on malware detection methods. In *Proceedings of the 3rd Hackers Workshop on Computer and Internet Security (IITKHACK09)*, pages 74–79, 2009.

[23] Xifeng Yan, Jiawei Han, and Ramin Afshar. Clospan: Mining closed sequential patterns in large datasets. In *In SDM*, pages 166–177, 2003.

[24] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *2010 International conference on broadband, wireless computing, communication and applications*, pages 297–300. IEEE, 2010.

[25] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC Press, 2012.

[26] Eckart Zitzler, Marco Laumanns, Lothar Thiele, Eckart Zitzler, Eckart Zitzler, Lothar Thiele, and Lothar Thiele. Spea2: Improving the strength pareto evolutionary algorithm, 2001.