An Incremental Ensemble Evolved by using Genetic Programming to Efficiently Detect Drifts in Cyber Security Datasets

Gianluigi Folino
Institute of High Performance Computing and Networking (ICAR-CNR)
Rende, Italy
folino@icar.cnr.it

Francesco Sergio Pisani
Institute of High Performance Computing and Networking (ICAR-CNR)
Rende, Italy
fspisani@icar.cnr.it

Pietro Sabatino
Institute of High Performance Computing and Networking (ICAR-CNR)
Rende, Italy
pietro.sabatino@icar.cnr.it

ABSTRACT

Unbalanced classes, the ability to detect changes in real-time, the speed of the streams and other peculiar characteristics make most of the data mining algorithms not apt to operate with datasets in the cyber security domain. To overcome these issues, we propose an ensemble-based algorithm, using a distributed Genetic Programming framework to generate the function to combine the classifiers and efficient strategies to react to changes in data. After that the base classifiers are trained, the combining function of the ensemble, based on non-trainable functions, can be generated without any extra phase of training, while the drift detection function adopted, together with a strategy for replacing classifiers, permits to respond in an efficient way to changes.

Preliminary experiments conducted on an artificial dataset and on a real intrusion detection dataset show the effectiveness of the approach.

1. INTRODUCTION

In the last few years, cyber security problems are gaining interest, as cyber crime seriously threatens national governments and the economy of many industries [3]. In this domain, computer and network technologies have intrinsic security weaknesses, i.e., protocol, operating system weaknesses, etc. In addition, computer network activities, human actions, etc. generate large amounts of data, hard to handle without using ad-hoc designed algorithms and distributed machines.

Intrusion Detection Systems (IDS) cope with the issue of detecting *unauthorized accesses* to computer systems and computer networks. An *intrusion* can be defined as an attempt by an outsider to gain access to the target system (local or network system). Data Mining methods and algorithms can support the detection phase of known attacks and indeed, from this research trend, a plethora of proposals appeared in recent years. Despite this, classical *sequential algorithms* are not suitable to capture in real time new trends and changes in *streaming data*, which may denote a network intrusion, as they assume that data are static and not changing due to external modifications. In this specific application scenario, *ensemble-based algorithms* (e.g., [13]) supply some specific characteristics of great interest in the context of intrusion detection, and, as a consequence, several *ensemble-based intrusion detection techniques* have been proposed recently [9]. Among well-recognized characteristics of such class of Data Mining algorithms, some that make them particularly suitable for supporting intrusion detection systems are the following ones: (*i*) they can be easily implemented in *parallel/distributed architectures*; (*ii*) they can improve the accuracy of a *weak learner*; (*iii*) they can be specialized for the detection of *a particular class*; (*iv*) they are particularly suitable to the special case of *unbalanced datasets*. However, building the ensemble could be computationally expensive as, when new data arrives, it is necessary to restart the training phase.

To overcome these issues, we extend the approach presented in [8], in order to handle data streams, to detect in an efficient way changes in the data and to build in an incremental way the ensemble of classifiers used to recognize the attacks. The above-cited approach, named CAGE-MetaCombiner (CMC for short), is a distributed intrusion detection framework, based on a well-known distributed GP tool [6], which is used to generate the combining function of the ensemble, composed by non-trainable functions. The main characteristic of the non-trainable functions is that they can be evolved without any extra phase of training and, therefore, they are particularly apt to handle concept drifts, also in the case of real-time constraints. In this work, we added to the framework a drift detection module, based on a detection function and on a clever strategy to replace the classifiers in the ensemble, which permits to respond in an efficient way to changes in the data.

Evolutionary algorithms have been used mainly to evolve and select the base classifiers composing the ensemble [4][7] or adopting some time-expensive algorithms to combine the ensemble [18]; however a limited number of papers concerns the evolution of the combining function of the ensemble by using GP and, to the best of our knowledge none of these works is able to cope with drifts. Chawla et al. [19] propose an evolutionary algorithm to combine the ensemble, based on a weighted linear combination of classifiers predictions, using many well-known data mining algorithm as base classifiers, i.e. J48, NBTree, JRip, etc. Yan Wang et al. [21] use multiple ensembles to classify incomplete datasets. Their strategy consists in partitioning the incomplete datasets in multiple complete sets and to train the different classifier on each sample. Then, the predictions of all the classifiers could be combined according to the ratio between the number of features in this subsample and the total features of the original dataset. A similar approach could

GECCO'16 Companion, July 20-24, 2016, Denver, CO, USA

DOI: http://dx.doi.org/10.1145/2908961.2931682

be included in our system. In [1], the authors develop a GP-based framework to evolve the fusion function of the ensemble both for heterogeneous and homogeneous ensemble. The approach is compared with other ensemble-based algorithms and the generalization properties of the approach are analyzed together with the frequency and the type of the classifiers presents in the solutions.

The rest of the paper is structured as follows. Section 2 shows how to use the GP tool to evolve the combining function of the ensemble. Section 3 illustrates the software architecture used in the system and shows the drift detection techniques employed and the strategies for the replacement of the classifiers of the ensemble. Section 4 presents some experiments conducted to verify the effectiveness of the approach and to compare it with other similar approaches. Finally, Section 5 concludes the work.

2. USING NON-TRAINABLE FUNCTIONS IN A GP-BASED ENSEMBLE

This section introduces the ensemble of classifiers and the "non-trainable combiner functions" and illustrates how the GP tool is used to evolve the combiner function of the ensemble.

2.1 Background: ensemble and non-trainable functions

Ensemble permits to combine multiple (heterogeneous or homogeneous) models in order to classify new unseen instances. In practice, after a number of classifiers are built usually using part of the dataset, the predictions of the different classifiers are combined and a common decision is taken. Different schemas can be considered to generate the classifiers and to combine the classifiers in the ensemble, i.e. the same learning algorithm can be trained on different datasets or/and different algorithms can be trained on the same dataset. In this work, different algorithms are used on the same dataset in order to build the different classifiers/models.

Let $S = \{(x_i, y_i)|i = 1, \ldots, N\}$ be a training set where x_i, called example or tuple or instance, is an attribute vector with m attributes and y_i is the class label associated with x_i. A predictor (classifier), given a new example, has the task to predict the class label for it.

Ensemble techniques build g predictors, each on a different training set, then combine them together to classify the test set. As an alternative, the g predictors could be built using different algorithms on the same/different training set.

The largely used boosting algorithm, introduced by Schapire [15] and Freund [16], follows a different schema; in order to boost the performance of any "weak" learning algorithm, i.e. an algorithm that "generates classifiers which need only be a little bit better than random guessing" [16], the method adaptively changes the distribution of the training set depending on how difficult each example is to classify.

This approach was successfully applied to a large number and types of datasets; however, it has the drawback of needing to repeat the training phase for a number of rounds and that could be really time-consuming for large datasets. The applications and the datasets in hard domains, as cyber security, have real-time requirements, which do not permit to re-train again the base models. On the contrary, ensemble strategies do not need any further phase of training, whether the functions used can be combined without using the original training set. The majority vote is a classical example of this kind of combiner function. Some types of combiner have no extra parameters that need to be trained and consequently, the ensemble is ready for operation as soon as the base classifiers are trained. These are named non-trainable combiners [13] and could be used as functions in a genetic programming tree.

Before describing the GP framework used, here, we introduce some definitions useful to understand how the algorithm works.

Let $x \in R^N$ be a feature vector and $\Omega = \{\omega_1, \omega_2 ..., \omega_c\}$ be the set of the possible class labels. Each classifier h_i in the ensemble outputs c degrees of support, i.e., for each class, it will give the probability that the tuple belongs to that class. Without loss of generality, we can assume that all the c degrees are in the interval $[0, 1]$, that is, $h_i : R^N \rightarrow [0, 1]^c$. Denote by $H_{i,j}(x)$ the support that classifier h_i gives to the hypothesis that x comes from class ω_j. The larger the support, the more likely the class label ω_j. A non-trainable combiner calculates the support for a class combining the support values of all the classifiers. For each tuple x of the training set, and considering g classifiers and c classes, a Decision Profile matrix DP can be build as follow:

$$DP(x) = \begin{bmatrix} H_{1,1}(x) & \ldots & H_{1,j}(x) & \ldots & H_{1,c}(x) \\ H_{i,1}(x) & \ldots & H_{i,j}(x) & \ldots & H_{i,c}(x) \\ H_{g,1}(x) & \ldots & H_{g,j}(x) & \ldots & H_{g,c}(x) \end{bmatrix}$$

where the element $H_{i,j}(x)$ is the support for j-th class of i-th classifier.

The functions used in our approach simply combine the values of a single column to compute the support for $j-th$ class and can be defined as follow:

$\mu_j(x) = F[H_{1,j}(x), H_{2,j}(x), ..., H_{g,j}(x)]$

For instance, the most simple function we can consider is the average, which can be computed as: $\mu_j(x) = \frac{1}{g}\sum_{i=1}^{g} H_{i,j}(x)$

The class label of x is the class with maximum support μ.

2.2 Using GP to generate the combining function

In this work, we adopt as GP engine, the CellulAr GEnetic programming (CAGE) [6], running both on distributed-memory parallel computers and on distributed environments, based on the fine-grained cellular model. The generation of the population, the different operators (i.e., crossover, mutation, etc.) are the same defined in the classical GP introduced by Koza [12]. Differently from classical models in which the GP tool is used to evolve the base classifiers, in our approach, the classifiers (with an associated weight previously computed on the training set) are the leaves of the tree, while the combiner functions are placed on the nodes. In practice, GP is used to evolve the combiner function, which the ensemble will adopt to classify new tuples. Implicitly, the function selects the more suitable classifiers/models to the specific datasets considered.

In particular, the functions chosen to combine the classifiers composing the ensemble are non-trainable functions and are listed in the following: average, weighted average, multiplication, maximum and median. They can be applied to a different number of classifiers, i.e. each function is replicated with a different arity, typically from 2 to 5. The choice of this set of functions is due to the fact that most of the papers adopting non-trainable functions use these combiners and obtain good experimental results [13]. The only function we do not include in this set was the product, which obviously does not perform well in the multi-class case. in the following, we specified better the functions used:

The **average** function, used with an arity of 2, 3 and 5, is defined as: $\mu_j(x) = \frac{1}{g}\sum_{i=1}^{g} H_{i,j}(x)$.

The **maximum** function returns the maximum support for 2, 3 and 5 classifiers and can be computed as: $\mu_j(x) = \max_i \{H_{i,j}(x)\}$.

The **median** function (arity 3 and 5) can be computed as: $\mu_j(x) = median_i \{H_{i,j}(x)\}$.

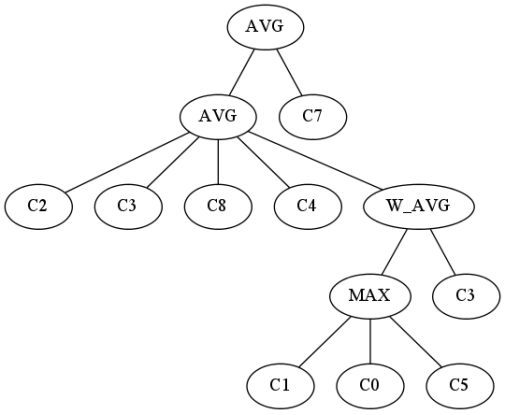

Figure 1: An example of the combiner function generated from the GP tool.

In order to better clarify, how the tree is built, in Figure 1, an example of tree generated from the tool is illustrated. As for the fitness function, it is simply computed as the error of the ensemble on the validation set, i.e. the ratio between the tuples not correctly classified and the total number of tuples. However, in the particular case of unbalanced datasets, a weighted fitness is adopted. In practice, if a tuple belonging to a minority class is misclassified, the fitness function is penalized by a weight equal to the ratio between the total number of tuples and the total number of tuples belonging to that class (to avoid really high weights, if the weight exceeds the threshold value of 10, it is fixed to this threshold). For the tuple belonging to the majority class, the penalty weight is fixed to 1, as in the case of balanced datasets.

3. THE SOFTWARE ARCHITECTURE OF THE FRAMEWORK

This section introduces a general architecture for processing and classifying intrusion detection datasets and details the main module of this architecture used to classify the attacks and the clever strategy adopted to detect the drift and to replace the classifiers composing the ensemble.

3.1 A general architecture for IDS

In Figure 2, a general architecture for an innovative intrusion detection system is described. Typically, the stream of data in input can originate from different sources: network traffic coming from a particular interface or coming from a router, system logs, application logs concerning the software installed on the system, etc. After the preprocessing phase, the features are extracted and the data stream is ready for further analysis. The next stage of the elaboration is performed by the module called Change Detector. It performs an analysis of the data stream seeking qualitative deviations from the normal behavior. Indeed, as it must perform a real time analysis, it typically divides the data stream in time windows of prefixed duration and then some functions are computed on the values of aggregated features coming from the window considered. The set of the values of these functions captures qualitative characteristics of the data stream; if an anomaly is detected in these sequences, or in the initial training phase of the system, the module Model Generator is activated in order to generate new models for the analysis of the data stream, performed by the Distributed Ensemble module. These new models are used to update the ensemble by using some replacement strategies or simply by adding them to the current ensemble. Finally, the overall ensemble is used to classify the new incoming tuples and to generate some alerts to be processed in some way.

3.2 The ensemble-based classifier

Following the schema illustrated in Subsection 3.1, our ensemble-based framework, which must operate in order to quickly detect and manage drift, must be composed by some main modules: a drift (change) detector, a generator of classifiers (models), a module to replace old/inefficient/overlapping classifiers. In addition, our framework also includes a module to generate the combiner function of the ensemble by using a genetic programming approach.

The tool used to evolve the combining function is a distributed GP implementation, named CellulAr GEnetic programming (CAGE) [6], running both on distributed-memory parallel computers and on distributed environments. The tool is based on the fine-grained cellular model. The overall population of the GP algorithm is partitioned into subpopulations of the same size. Each subpopulation can be assigned to one processor and a standard (panmictic) GP algorithm is executed on it. Occasionally, migration process between subpopulations is carried out after a fixed number of generations. For example, the n best individuals from one subpopulation are copied into the other subpopulations, thus allowing the exchange of genetic information between populations. The model is hybrid and modifies the island model by substituting the standard GP algorithm with a cellular GP (cGP) algorithm. In the cellular model each individual has a spatial location, a small neighborhood and interacts only within its neighborhood. The main difference in a cellular GP, with respect to a panmictic algorithm, is its decentralized selection mechanism and the genetic operators (crossover, mutation) adopted.

This tool is used to evolve the combiner functions and obtain an overall combiner function, which the ensemble will adopt to classify new tuples. Implicitly, the function selects the classifiers/models more apt to the particular datasets considered. More details on how the combiner function of the ensemble is evolved are reported in [8]. Briefly, as nodes of the GP tree, some non-trainable functions are chosen, better specified in the experimental section, while the leafs of the tree are the different classifiers selected in the previous phase (see Figure 1) and the fitness function is simply the error of the ensemble computed on the validation set. It is worth to remember that no extra computation on the data is necessary to build the GP trees, as the validation set is only used to verify the correct class is assigned and consequently to compute the fitness function.

The model generator is built on the well-known Massive Online Analysis (MOA) toolbox[1], adopted to handle the data streams and also used for the implementation of the classifier algorithms and for the drift detection strategies described in the Subsection 3.3. Finally, the module to replace old/inefficient/overlapping classifiers adopts the strategies also described in the Subsection 3.3.

The way in which the overall algorithm works is better illustrated in the pseudocode of Figure 3. For the sake of simplicity, we suppose that the detection method is based on a window of prefixed length of tuples; however, the methodology can be applied to most of the drift detection methods present in the MOA framework. It is beyond the aim of this paper to compare different strategy of drift detections. We refer to [11], for a comprehensive comparison and on the basis of the experiments reported in this paper, we choose the detection algorithm used in our experiments.

An infinite stream of tuples is the input of the model generator and they are analyzed in windows of pre-fixed length n. On each

[1]MOA release 2014.11; http://moa.cms.waikato.ac.nz/.

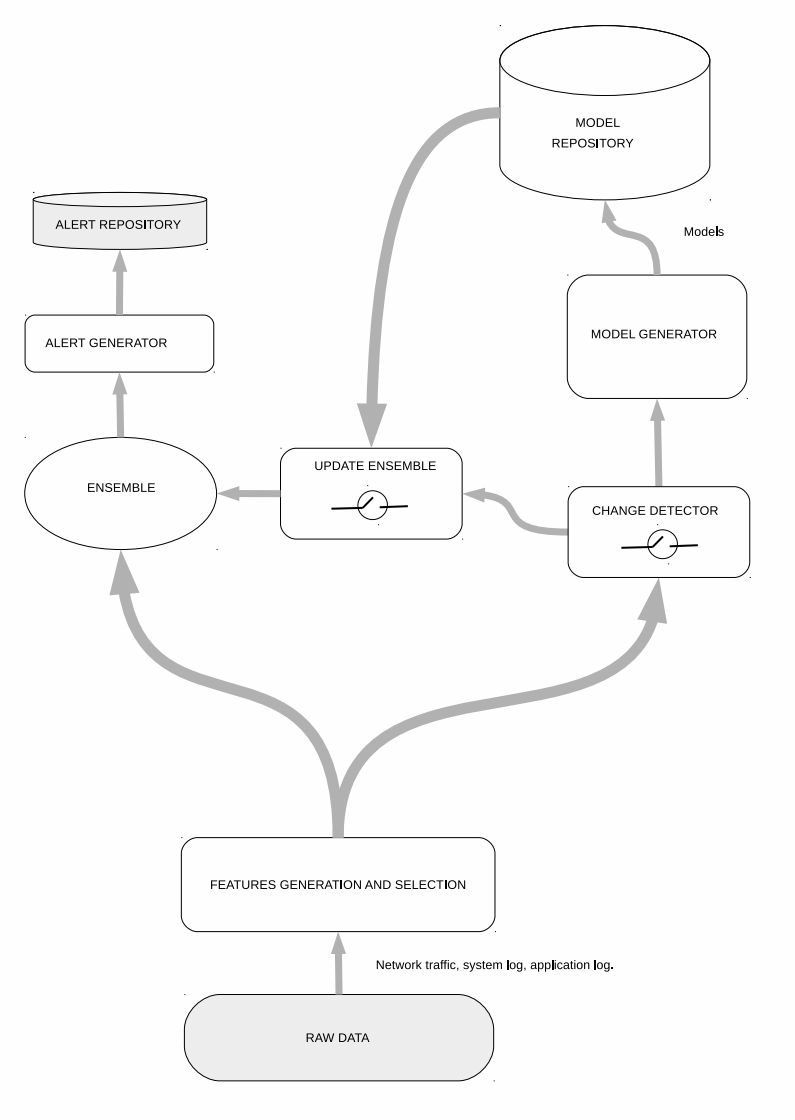

Figure 2: A general architecture for classifying large streams of attacks and normal connections.

window, a function verifies whether a drift occurred; only in the positive case, the labelled tuples of the current window are partitioned equally into training and validation set. The training set is used to train the base classifier algorithms selected among the available in the MOA tool and better specified in the experimental section. Among them, the l classifiers having the better accuracy on the training set are chosen and added to the current ensemble. A replacing strategy is used to remove some classifiers from the ensemble, whether the maximum number of elements is reached. Afterwards, the GP tool is run to generate the combiner function, which will be applied to the ensemble, by using the validation set.

Finally, if no drift is detected, the new incoming tuples are classified using the following schema: for each tuple x, for each possible class j, the supports are computed using the formula: $\mu_j(x) = \sum H_{ij}(x)$. and the final classification is obtained by using the formula $class(x) = argmax_j(\mu_j(x))$

3.3 Drift detection and replacement strategies

In this work, we are not particularly interested in developing a new drift detection strategy, but we used that included in the MOA tool. In particular, on the basis of the experiments reported in [11], which analyze the performance of the most important drift detection strategies, we choose STEPD and ADWIN strategies.

Statistical Test of Equal Proportions (STEPD) [14] is a drift detection algorithm based on the accuracy. It computes two statistics: the overall accuracy from the beginning of the stream and the accuracy of the model computed on a testing window W. If the difference between the accuracy computed on W and the overall accuracy is greater than a threshold, then a drift is detected and the model must be rebuilt/updated. In the implementation described in the above-cited paper, the instances stored in memory are used to update the classifier when a drift is detected. Our algorithm updates the models by using the instances of the last window. The window size is fixed, and it contains the target instance (when STEPD detects drift) and its neighbors. The threshold value P is computed as the percentile of the standard normal distribution in order to obtain

Let α be the maximum number of base classification algorithms used.
Let ł be the number of base classification selected (with $l < \alpha$) .
Let M be the maximum number of classifiers composing the ensemble.
Let *wind* the size of the window examined.
Given $T_1, T_2, \ldots T_\infty$, the infinite tuples composing the stream, analyze in windows of size n,
named $\{T_{w_1}, T_{w_2}, \ldots, T_{w_\infty}\}$.
where $T_i = \{T_{i_1}, T_{i_2}, \ldots, T_{i_m}\}$ is a tuple with associated a number of attributes m and a class.
The current ensemble $E = \{C_1, C_2, \ldots, C_M\}$
for each (T_{w_i}**)**
  if (drift_detection_function (T_{w_i}))
    Partition the tuples composing T_{w_i} in a training set and a validation set: $Train_i$ and $Valid_i$
    Train α different classification algorithms on $Train_i$
    Select the l classifiers obtaining the best accuracy on the training set.
    Add these classifiers to the ensemble E
    if ($\mid E \mid > M$)
      prune the ensemble E, removing $M - |E|$ classifiers, by using a strategy of pruning.
    end if
    Build l decision profile matrixes, one for each of the classifiers, $DP_1, DP_2, \ldots, DP_l$ using the validation set,
    one for each classifier of dimension $k \times c$, where k is the number of tuples and c the number of classes.
    Run the distributed GP tool on the validation set $Valid_i$ in order to obtain the combiner function of the ensemble.
    Obtain an Ensemble E, a combiner function FC ,
  else
    Build the decision profile matrix DP of the entire ensemble E,
    where each element $H_{i,j}(x)$ is the support that classifier h_i gives to the hypothesis that the tuple x comes from class ω_j.
    Compute for each class j: $\mu_j(x) = \sum H_{ij}(x)$.
    Compute the class by using the formula $class(x) = argmax_j(\mu_j(x))$.
  end if
end for each

Figure 3: The pseudo-code of the algorithm.

the observed significance level. STEPD uses three parameters: the window value to detect recent changes (we are using 20 instances, as in original paper) and the significance levels α_w and α_d. In practice, STEPD stores the instances in its memory when $P < \alpha_w$ and it resets all the variables (i.e., clear its memory, reset the window accuracy, etc.) when $P < \alpha_d$. As suggested in the original paper, we use a value of 0.03 for α_d and 0.08 for α_w.

The ADaptative WINDdowing method [2] keeps a sliding window W with the most recent examples and compares the distribution on two sub-windows of W. When the difference of the average value of the two sub-windows is greater than a threshold, then the older sub-window is dropped and a change in the distribution of examples is assigned. However, this idea is computationally intensive and requires a huge amount of memory. Therefore, the current implementation is based on a different data structure. The input records are stored in exponential histograms, a data structure that maintains an approximation of the number of 1 contained in a sliding window of length W using logarithmic memory and update time. The length W is update to fit stream variations. The input of the algorithm are real numbers in the interval [0,1] and to detect drift, only values of 0 and 1 are processed: wrong predictions are marked as 1, good one as 0. A confidence parameter δ with value of 0.002 is used to control the false positive rate, i.e. if the expected value of distribution remains constant within W, the probability that ADWIN shrinks the window at this step is at most δ.

As for the strategies for replacing the classifiers composing the ensemble with the new generated by the algorithm, we developed three strategies, named old, best and wheel selection. In the training phase, the dataset is partitioned into two parts: 50% of the instances are selected to train classifiers and the remaining instances compose a validation set to build the ensemble model. The old strategy replaces 1/3 of classifiers in the ensemble with the new generated. The classifiers are removed considering the insertion time in the ensemble model, i.e. the oldest are removed. The new classifier is chosen by considering the accuracy result on the validation set. The best strategy works as the previous one, but it removes classifiers having the worst accuracy on the validation set and it inserts the best ones. As in the previous case, only a 1/3 of classifiers are replaced. Finally, the wheel selection strategy is a selection algorithm based on the wheel mechanism. When a drift is detected, all the classifiers in the repository are trained on the training dataset. Then, classifiers are replaced with a roulette wheel selection algorithm. The performance on validation set is used as a probability, then 1/3 classifiers are selected with probability p_i using formula $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$ where f_i is the accuracy on the validation set.

4. EXPERIMENTAL SECTION

Two set of experiments were performed, by using an artificial dataset and an intrusion detection dataset, in order to verify the goodness of the approach. In the next two subsections, the datasets and the parameters used in the experiments are illustrated.

4.1 Parameter Settings

All the experiments were performed on a Linux cluster with 16 Itanium2 1.4 GHz nodes, each having 2 GBytes of main memory and connected by a Myrinet high performance network. No tuning phase has been conducted for the GP tool, but the same parame-

ters used in the original paper were used, listed in the following: a probability of crossover equal to 0.7 and of mutation equal to 0.1, a maximum depth equal to 7, a population of 120 individuals and 500 as number of generations. All the results were obtained by averaging 30 runs.

Among the many metrics for evaluating classifier systems, in this paper we choose recall and precision, because they give an idea of the capacity of the system in individuating the attacks and in reducing the number of false alarms; indeed, recall represents the proportion of correctly predicted attack cases to the actual size of the attack class (a value of 100% indicate we detect all the attacks, however, we can individuate also a large number of false attacks); precision represents the proportion of attack cases that were correctly predicted relative to the predicted size of the attack class (a value of 100% indicates that no false alarms were signaled, however a large number of alarms could be not detected).

The AUC metric is the value of the area under the ROC curve. The ROC curve is computed comparing the false positive rate and the true positive rate. The first term measures the capacity to correctly detect attacks (i.e. recall). The second term measures the rate between the false alarm signaled above all normal connections processed. Computing the area, we have a number to describe the goodness of classifier. An AUC close to 1 means an optimal recognition rate.

The different classifiers composing the ensemble are trained on the same training set. In practice, for each window of the stream, 50% is used to train the base classifiers, the remaining 50% is is used as validation set. The validation part is used by the evolutionary algorithm to evolve the combination function. The maximum number of classifiers is fixed to 20 and the number of classifiers composing the ensemble are 10.

CAGE-MetaCombiner (CMC for short) uses many learner as base classifiers, then it makes a selection using a strategy chosen among best, old and wheel selection as described in the previous section.

The algorithms used as base classifiers in the experiments are based on the WEKA implementation[2] and are listed in the following: J48 (decision trees), JRIP rule learner (Ripper rule learning algorithm), NBTree (Naive Bayes tree), Naive Bayes, 1R classifier, logistic model trees, logistic regression, decision stumps and 1BK (k-nearest neighbor algorithm).

4.2 Description of the datasets

The performances of CAGE-MetaCombiner are evaluated on an artificial dataset and on a real dataset of the cyber security domain. To generate the artificial dataset, we use the HyperPlane generator available in MOA. This generator is very popular as benchmark for drift detection algorithms. It was originally used in (Hulten et al. 2001) [10]. It generates data for a binary classification problem, taking a random hyperplane in d-dimensional Euclidean space as a decision boundary. The user can customize the number of attributes generated, the attributes used to generate drift (the others can be considered as irrelevant), the magnitude of changes and the percentage of noise to add to the data.

DARPA and KDD are two very popular datasets used in the classification in the IDS domain, see Travallaee et al. [20]; however, they have been thoroughly criticized for being unable to provide a realistic scenario. To overcome this issue, we choose to conduct our experiments on the ISCX IDS dataset from the Information Security Centre of Excellence of the University of New Brunswick [17]. This dataset is the result of capturing seven days of network traffic in a controlled testbed made of a subnetwork placed behind a firewall. Normal traffic was generated with the aid of agents that simulated normal requests of human users following some probability distributions extrapolated from real traffic. Attack were generated with the aid of human operators. The result is a fully labelled dataset containing realistic traffic scenarios. Indeed, the dataset consists of standards `pcap` (packet capture) files one for each day containing the relative network traffic, as illustrated in Table 1. Different days contain different attack scenarios, ranging from HTTP Denial of Service, DDos, Brute Force SSH and attempts of infiltrating the subnetwork from the inside. The main characteristics of this dataset are summarized in Table 1.

[2]http://www.cs.waikato.ac.nz/ml/weka

4.3 Experiments on the artificial dataset

In order to evaluate the effectiveness of the replacement strategies, of the drift detection techniques used and, in general, of all the approach, we conducted our experiments by varying the size of the evaluation window of the stream. In addition, our approach is compared with the well-known incremental algorithm HoeffdingTree using the same drift detection algorithm of our approach and with a boosted version of the HoeffdingTree. As for the window size, we want to remark that it is the number of records used to train the base classifiers and to build the base classifier. The metrics of recall, precision and AUC are computed for each window and averaged over all the windows.

In all the tables of this section and of the next one, we report in bold the values that **are not** statistically significant using the Friedman test (note that we made this choice, as most of the differences are significative). The critical value of the Friedman test [5] is obtained from a chi-square distribution with two degree of freedom and a significancy level of 5%. The Friedman test is a non-parametric statistical test and it is used to detect differences across multiple test. The null hypothesis of this test is that the median value of all the populations is equal.

In Tables 2, 3 and 4, are shown respectively the precision, the recall and the AUC metrics concerning the HoeffdingTree algorithm (classic and boosted version) and the different replacement strategies used in our approach: old, best and wheel. The experiments show that the size of the evaluation window does not affect significantly the performance of the algorithms. Our approach is slightly better than the others and the better strategy of replacement is that replaces the worst trees with the best ones. This behavior can be referred to nature of the generated dataset. The same conclusions can be drawn for the AUC metric.

4.4 Experiments on the real dataset

Generally, in the intrusion detection datasets, most of the instances represent normal connections, while the attacks represent the minority classes, often with different orders of magnitudo. Therefore, the ISCX dataset, described in the Subsection 4.2, which is representative of this situation, was used for the experiments of this section.

Analyzing Table 1 and the relative dataset, it is evident that the attacks are grouped in a small range of windows and, for different days, different kinds of attack can be observed. It worth to notice that drifts (changes in the data) can be detected when a new type of attack appears for the first time. Therefore, this dataset is appropriate in order to test our approach.

In Tables 5, 6 and 7 the experimental results for the ISCX dataset are shown and the behavior of our approach is examined with the different replacement strategies.

Day	Description	Size of the pcap file (GB)	Number of Flows	Percentage of Attacks
Day 1	*Normal traffic without malicious activities*	16.1	359,673	0.000 %
Day 2	*Normal traffic with some malicious activities*	4.22	134,752	1.545 %
Day 3	*Infiltrating the network from the inside* & Normal traffic	3.95	153,409	6.395 %
Day 4	*HTTP Denial of Service & Normal traffic*	6.85	178,825	1.855 %
Day 5	*Distributed Denial of Service using an IRC Botnet*	23.4	554,659	6.686 %
Day 6	*Normal traffic without malicious activities*	17.6	505,057	0.000 %
Day 7	*Brute Force SSH + Normal activities*	12.3	344,245	1.435 %

Table 1: Main characteristics of the ISCX IDS dataset.

In this case, the wheel selection strategy has the better performance for both the precision and recall metrics. The same behavior is observable in Table 7 for the AUC metric.

Table 2: Precision for CMC and HoeffdingTree (classic and boosted version) (HyperPlane dataset).

	Precision			
	1k	2k	5k	10k
STEPD				
HT boosted	**87.66** ± 2.12	87.66 ± 2.93	87.80 ± 1.97	87.97 ± 0.68
HoeffdingTree	**87.48** ± 1.75	87.56 ± 0.27	87.64 ± 2.97	87.86 ± 2.81
CMC wheel	**87.68** ± 1.52	87.87 ± 2.61	87.39 ± 1.26	87.49 ± 2.98
CMC best	**87.68** ± 2.72	90.01 ± 0.81	93.28 ± 0.74	85.52 ± 0.99
CMC old	**87.47** ± 2.61	89.78 ± 1.78	90.28 ± 2.35	85.55 ± 1.27
ADWIN				
HT boosted	84.83 ± 0.05	85.00 ± 1.54	83.52 ± 0.33	83.52 ± 0.78
HoeffdingTree	86.44 ± 0.46	85.99 ± 0.07	86.25 ± 0.06	85.41 ± 0.59
CMC wheel	87.68 ± 2.46	85.48 ± 2.19	83.83 ± 0.92	84.72 ± 1.38
CMC best	87.68 ± 2.53	90.01 ± 0.41	91.92 ± 2.71	85.52 ± 0.21
CMC old	87.13 ± 1.71	89.29 ± 1.41	90.63 ± 1.77	84.23 ± 2.06

Table 3: Recall for CMC and HoeffdingTree (classic and boosted version) (HyperPlane dataset).

	Recall			
	1k	2k	5k	10k
STEPD				
HT boosted	87.30 ± 2.69	87.37 ± 0.85	87.53 ± 2.07	87.59 ± 0.58
HoeffdingTree	87.25 ± 0.50	87.51 ± 2.66	87.55 ± 2.90	87.82 ± 1.04
CMC wheel	87.33 ± 0.32	87.48 ± 0.71	88.59 ± 1.17	87.98 ± 1.17
CMC best	91.32 ± 1.31	91.40 ± 2.67	85.86 ± 0.69	87.55 ± 0.15
CMC old	90.56 ± 2.77	89.01 ± 2.89	85.48 ± 0.83	87.32 ± 0.43
ADWIN				
HT boosted	84.11 ± 1.40	84.16 ± 2.46	83.47 ± 1.85	83.77 ± 2.66
HoeffdingTree	86.48 ± 1.62	86.39 ± 1.62	85.45 ± 2.94	84.67 ± 1.70
CMC wheel	91.32 ± 0.84	85.36 ± 1.72	84.41 ± 1.59	83.79 ± 1.35
CMC best	91.32 ± 1.55	91.40 ± 1.16	91.66 ± 1.06	87.55 ± 1.92
CMC old	89.72 ± 2.75	89.80 ± 2.76	89.66 ± 2.13	85.55 ± 2.48

Table 4: AUC metric for CMC and HoeffdingTree (classic and boosted version) (HyperPlane dataset).

	AUC			
	1k	2k	5k	10k
STEPD				
HT boosted	0.87 ± .029	0.88 ± .013	0.88 ± .012	0.88 ± .025
HoeffdingTree	0.87 ± .010	0.88 ± .022	0.88 ± .015	0.88 ± .001
CMC wheel	0.89 ± .019	0.89 ± .023	0.90 ± .002	0.89 ± .018
CMC best	0.89 ± .003	0.91 ± .001	0.90 ± .006	0.86 ± .019
CMC old	0.89 ± .013	0.90 ± .012	0.89 ± .004	0.86 ± .025
ADWIN				
HT boosted	0.86 ± .014	0.84 ± .010	0.83 ± .015	0.83 ± .029
HoeffdingTree	0.86 ± .078	0.86 ± .029	0.86 ± .007	0.85 ± .025
CMC wheel	0.89 ± .030	0.85 ± .010	0.84 ± .025	0.82 ± .015
CMC best	0.89 ± .016	0.91 ± .014	0.92 ± .026	0.86 ± .028
CMC old	0.88 ± .094	0.90 ± .012	0.92 ± .013	0.85 ± .022

In Tables 8, 9 and 10 are shown respectively the recall and precision and the AUC metrics concerning the comparison between the HoeffdingTree algorithm (classic and boosted version) and our approach (CMC) with the wheel selection strategy. We use the ADWIN algorithm as drift detection because it has equivalent performance (better in some cases) in comparison with STEPD. Furthermore, considering only the AUC metric, ADWIN has a slight advantage on the performance of the STEPD algorithm for the ISCX dataset. Note that, the HoeffdingTree algorithm updates more frequently its model, and therefore, also for small windows, it can improve quickly the predictive capacity. However, our approach has performance close to the HoeffdingTree for small windows and it performs sensibly better when the window size grows. Furthermore, comparing the performance for both the version of the HoeffdingTree, a performance degradation of the ensemble-based algorithm can be observed. This behavior does not affect our approach, probably because when a drift is detected, it updates/replaces the models and re-weights the classifiers (by recomputing the combination function).

Table 5: Precision for different replacement strategies for CMC (ISCX dataset).

	Precision			
	1k	2k	5k	10k
STEPD				
CMC wheel	83.46 ± 0.29	89.40 ± 0.31	87.75 ± 2.24	89.97 ± 0.11
CMC best	79.19 ± 0.19	82.31 ± 2.65	74.30 ± 0.36	84.67 ± 2.29
CMC old	70.99 ± 0.95	75.98 ± 0.32	75.25 ± 0.59	78.98 ± 0.29
ADWIN				
CMC wheel	83.46 ± 0.81	87.59 ± 1.07	92.42 ± 0.77	88.28 ± 2.71
CMC best	57.98 ± 2.01	61.90 ± 0.79	65.90 ± 0.15	58.67 ± 1.92
CMC old	56.75 ± 2.73	63.65 ± 0.12	63.83 ± 1.99	59.80 ± 2.67

Table 6: Recall for different replacement strategies for CMC (ISCX dataset).

	Recall			
	1k	2k	5k	10k
STEPD				
CMC wheel	88.39 ± 2.75	85.47 ± 1.62	85.10 ± 2.71	80.41 ± 2.38
CMC best	65.98 ± 2.51	66.56 ± 2.40	82.85 ± 0.33	71.70 ± 2.96
CMC old	59.34 ± 1.63	62.35 ± 0.26	63.35 ± 1.09	67.98 ± 0.16
ADWIN				
CMC wheel	88.39 ± 2.51	88.25 ± 2.38	82.44 ± 2.44	80.79 ± 0.08
CMC best	67.44 ± 1.82	67.62 ± 0.60	73.90 ± 1.56	62.82 ± 0.69
CMC old	63.54 ± 2.82	64.34 ± 2.12	64.90 ± 0.71	61.98 ± 2.41

Table 7: AUC metric for different replacement strategies for CMC (ISCX dataset).

	AUC			
	1k	2k	5k	10k
STEPD				
CMC wheel	0.84 ± .019	0.86 ± .018	0.87 ± .007	0.87 ± .018
CMC best	0.72 ± .029	0.79 ± .016	0.86 ± .013	0.84 ± .008
CMC old	0.59 ± .009	0.60 ± .002	0.60 ± .007	0.62 ± .017
ADWIN				
CMC wheel	0.84 ± .014	0.87 ± .017	0.89 ± .007	0.89 ± .002
CMC best	0.51 ± .026	0.51 ± .018	0.55 ± .019	0.52 ± .015
CMC old	0.53 ± .003	0.53 ± .015	0.53 ± .001	0.52 ± .017

Table 8: Precision for the comparison among our approach, the Hoeffding tree (classical and boosted version) on ISCX dataset.

	Precision			
	1k	2k	5k	10k
ADWIN				
CMC wheel	83.46 ± 0.28	87.59 ± 0.03	92.42 ± 2.20	88.28 ± 2.69
HT boosted	84.72 ± 2.67	81.79 ± 2.82	79.50 ± 0.14	75.29 ± 2.82
HoeffdingTree	89.22 ± 1.67	87.51 ± 2.23	87.23 ± 1.80	87.48 ± 1.51

Table 9: Recall for the comparison among our approach, the Hoeffding tree (classical and boosted version) on ISCX dataset.

	Recall			
	1k	2k	5k	10k
ADWIN				
CMC wheel	88.39 ± 0.69	88.25 ± 1.08	82.44 ± 0.69	80.79 ± 0.26
HT boosted	85.20 ± 1.04	81.35 ± 2.14	70.62 ± 1.15	58.72 ± 1.90
HoeffdingTree	92.66 ± 0.92	87.76 ± 2.86	79.69 ± 0.53	75.62 ± 2.10

Table 10: AUC metric for the comparison among our approach, the Hoeffding tree (classical and boosted version) on ISCX dataset.

	AUC			
	1k	2k	5k	10k
ADWIN				
CMC wheel	$0.84 \pm .022$	$0.87 \pm .003$	$0.89 \pm .001$	$0.89 \pm .007$
HT boosted	$0.82 \pm .021$	$0.79 \pm .007$	$0.75 \pm .009$	$0.69 \pm .003$
HoeffdingTree	$0.89 \pm .014$	$0.88 \pm .001$	$0.85 \pm .012$	$0.83 \pm .013$

5. CONCLUSIONS AND FUTURE WORKS

A framework for classifying streaming intrusion detection datasets based on the ensemble model, is presented. The system evolves a combiner function, which does not need additional phases of training, after the heterogeneous classifiers composing the ensemble are trained. The framework includes a drift detection function to detect changes in the data and a strategy for replacing classifiers, which permits to build the ensemble in an incremental way. Preliminary experiments showed that the framework is apt to cope with large streams of normal connections and attacks and that the wheel selection strategy obtains encouraging results in comparison with other replacement strategies. In future works, we intend to better investigate the ability of the algorithm to handle large real-world datastreams in the cyber security domain.

Acknowledgment

This work has been partially supported by MIUR-PON under project PON03PE_00032_2 within the framework of the Technological District on Cyber Security.

6. REFERENCES

[1] N. Acosta-Mendoza, A. Morales-Reyes, H. J. Escalante, and A. Gago-Alonso. Learning to assemble classifiers via genetic programming. *IJPRAI*, 28(7), 2014.

[2] Albert Bifet and Ricard Gavalda. Learning from time-changing data with adaptive windowing. In *SDM*, volume 7, page 2007. SIAM, 2007.

[3] CERT Australia. Cyber crime and security survey report. Technical report, 2012.

[4] D. F. de Oliveira, A. M. P. Canuto, and M. C. P. de Souto. Use of multi-objective genetic algorithms to investigate the diversity/accuracy dilemma in heterogeneous ensembles. In *International Joint Conference on Neural Networks*, pages 2339–2346. IEEE, 2009.

[5] Janez Demsar. Statistical comparisons of classifiers over multiple data sets. *Journal of Machine Learning Research*, 7:1–30, 2006.

[6] G. Folino, C. Pizzuti, and G. Spezzano. A scalable cellular implementation of parallel genetic programming. *IEEE Transactions on Evolutionary Computation*, 7(1):37–53, February 2003.

[7] G. Folino, C. Pizzuti, and G. Spezzano. Training Distributed GP Ensemble With a Selective Algorithm Based on Clustering and Pruning for Pattern Classification. *IEEE Trans. Evolutionary Computation*, 12(4):458–468, 2008.

[8] Gianluigi Folino, Francesco Sergio Pisani, and Pietro Sabatino. A distributed intrusion detection framework based on evolved specialized ensembles of classifiers. In Giovanni Squillero and Paolo Burelli, editors, *Applications of Evolutionary Computation - 19th European Conference, EvoApplications 2016, Porto, Portugal, March 30 - April 1, 2016, Proceedings, Part I*, volume 9597 of *Lecture Notes in Computer Science*, pages 315–331. Springer, 2016.

[9] Gianluigi Folino and Pietro Sabatino. Ensemble based collaborative and distributed intrusion detection systems: A survey. *J. Netw. Comput. Appl.*, 66(C):1–16, May 2016.

[10] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106. ACM, 2001.

[11] Paulo Mauricio Gonçalves Jr., Silas Garrido Teixeira de Carvalho Santos, Roberto Souto Maior de Barros, and Davi Carnauba De Lima Vieira. A comparative study on concept drift detectors. *Expert Syst. Appl.*, 41(18):8144–8156, 2014.

[12] J. R. Koza. *Genetic Programming: On the Programming of Computers by means of Natural Selection*. MIT Press, Cambridge, MA, 1992.

[13] L. Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-Interscience, 2004.

[14] Kyosuke Nishida and Koichiro Yamauchi. Detecting concept drift using statistical testing. In *Discovery Science*, pages 264–269. Springer, 2007.

[15] R. E. Schapire. The strength of weak learnability. *Machine Learning*, 5(2):197–227, 1990.

[16] R. E. Schapire. Boosting a weak learning by majority. *Information and Computation*, 121(2):256–285, 1995.

[17] Ali Shiravi, Hadi Shiravi, Mahbod Tavallaee, and Ali A. Ghorbani. Toward developing a systematic approach to generate benchmark datasets for intrusion detection. *Computers & Security*, 31(3):357 – 374, 2012.

[18] C. De Stefano, G. Folino, F. Fontanella, and A. Scotto di Freca. Using bayesian networks for selecting classifiers in GP ensembles. *Information Sciences*, 258:200–216, 2014.

[19] J. Sylvester and N. V. Chawla. Evolutionary ensembles: Combining learning agents using genetic algorithms. In *AAAI Workshop on Multiagent Learning*, pages 46–51, 2005.

[20] M. Tavallaee, N. Stakhanova, and A.A. Ghorbani. Toward credible evaluation of anomaly-based intrusion-detection methods. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 40(5):516–524, Sept 2010.

[21] Y. Wang, Y. Gao, R. Shen, and F. Yang. Selective ensemble approach for classification of datasets with incomplete values. In *Foundations of Intelligent Systems*, pages 281–286. Springer, 2012.