Soft Computing (2020) 24:17541–17560
https://doi.org/10.1007/s00500-020-05200-3

FOCUS

A GP-based ensemble classification framework for time-changing streams of intrusion detection data

Gianluigi Folino[1] · Francesco Sergio Pisani[1] · Luigi Pontieri[1]

Published online: 1 August 2020

Abstract
Intrusion detection tools have largely benefitted from the usage of supervised classification methods developed in the field of data mining. However, the data produced by modern system/network logs pose many problems, such as the streaming and non-stationary nature of such data, their volume and velocity, and the presence of imbalanced classes. Classifier ensembles look a valid solution for this scenario, owing to their flexibility and scalability. In particular, data-driven schemes for combining the predictions of multiple classifiers have been shown superior to traditional fixed aggregation criteria (e.g., predictions' averaging and weighted voting). In intrusion detection settings, however, such schemes must be devised in an efficient way, since (part of) the ensemble may need to be re-trained frequently. A novel ensemble-based framework is proposed here for the online intrusion detection, where the ensemble is updated through an incremental stream-oriented learning scheme, correspondingly to the detection of concept drifts. Differently from mainstream ensemble-based approaches in the field, our proposal relies on deriving, though an efficient genetic programming (GP) method, an expressive kind of combiner function defined in terms of (non-trainable) aggregation functions. This approach is supported by a system architecture, which integrates different kinds of functionalities, ranging from the drift detection, to the induction and replacement of base classifiers, up to the distributed computation of GP-based combiners. Experiments on both artificial and real-life datasets confirmed the validity of the approach.

Keywords Data streams · Ensemble learning · Genetic programming · Intrusion detection · Cybersecurity

1 Introduction

Cyber-security issues are attracting increasing interest in disparate fields, owing to the severe threats that cyber crime is posing to citizens, companies, and governments (CERT Australia 2012). Intrusion detection tools constitute a valuable solution in this context, for timely recognizing malicious behaviors, and eventually protecting information systems, sensitive information and physical/monetary assets. Basically, a *Intrusion Detection System (IDS)* is a system devoted to automatically detect suspicious activities, witnessing unauthorized accesses (*intrusions*) to a computer system/network, based on a continuous analysis of different kinds of log data (e.g., network traffic's logs, application/system logs, etc.).

Very many proposals have appeared in the last two decades in the literature that leverage classification-oriented data mining techniques for detecting and analyzing intrusions, based on these data. However, such a classification problem poses many challenging issues, which primarily include the big-data nature of the input data (which tend to be generated in large volumes and at fast paces), the non-stationarity of their distribution (due to the fact that attack patterns tend to evolve over the time and, hence, lead to frequent concept drifts), as well as class imbalance (intrusions are usually far less frequent than normal behaviors).

While many traditional data mining solutions fail to handle all these issues adequately, ensemble-based classification approaches look a valid solution for this challenging scenario, owing to their flexibility and scalability, as confirmed by the wide diffusion of *ensemble-based intrusion detection techniques* (Folino and Sabatino 2016). In fact, this mainly descends from the fact that ensemble schemes are easy to

Communicated by Yaroslav D. Sergeyev.

✉ Gianluigi Folino
gianluigi.folino@icar.cnr.it

1 ICAR-CNR, Rende, Italy

implement efficiently, on top of parallel/distributed architectures, and that ensemble models can be trained to pay more attention to important/rarer attack classes.

A delicate crucial aspect in the design of an ensemble-classification model pertains the combiner function, i.e., the function used to fuse the predictions of the (base) classifiers in the ensemble. Learning-based combiners, capable of adapting the combination logics to the data at hand, were shown to surpass the traditional strategy of simply resorting to a fixed (non-trainable) aggregation criterion, such as averaging the base models' predictions, or performing some kind of weighted voting. However, when applied to (non-stationary) intrusion detection data, such an approach might turn to be unfeasible, seeing as (part of) the ensemble might need to be re-trained frequently.

In this work, a comprehensive ensemble-based framework for the online classification of fast non-stationary data streams (like those arising in typical cybersecurity contexts) is proposed, which leverages a *Genetic Programming* (*GP*) method for automatically deriving an expressive combiner function, defined in terms of non-trainable aggregation sub-functions. The framework, named *Evolutionary Ensemble-based Stream Classification for Intrusion Detection* (or simply $E2SC4ID$ for the sake of conciseness), has been devised to timely adapt to concept drifts by suitably updating the ensemble model (in terms of both its underlying base classifiers and its combiner function), according to a continuous learning-and-prediction approach. This approach is supported by an ad hoc system architecture that integrates different kinds of functionalities, ranging from drift-detection, to the induction/replacement of base models, to the efficient GP-based computation of the ensemble's combiner function. Notably, using non-trainable functions as building blocks for generating candidate solutions (i.e., possible combiner functions for the ensemble) allows us to evaluate the fitness of these solutions efficiently, without needing to perform costly training steps. Combined with the exploitation of a distributed GP platform (Folino et al. 2003), this property makes the proposed framework particularly apt to deal with non-stationary data, even in contexts where strict processing-time requirements exist.

Our proposal is technically different from previous appro aches to employing evolutionary/GP techniques for the discovery of ensemble-based classifiers. In fact, many of these approaches have only focused on the construction or selection of the base classifiers (de Oliveira et al. 2009; Folino et al. 2008). On the other hand, most of the few proposals (De Stefano et al. 2014; Sylvester and Chawla 2005; Acosta-Mendoza et al. 2014) addressing the generation of expressive combiner functions suffer from two serious limitations: either (i) assume the data distribution to be stationary, or (ii) cannot adapt to concept drifts in a sufficiently timely manner, as they need to evaluate the fitness of the generated individuals over the training dataset.

We pinpoint that this manuscript is a revised and extended version of a paper (Folino et al. 2019) published at conference *NUMTA 2019*, and it builds up on some ideas and basic solutions presented in Folino et al. (2016b). Differently from these two (short) papers, the continuous learning-and-classify approach proposed here (in Sect. 4) has been devised to work in a fully online and incremental fashion, on a per-instance basis, with the help of novel ad hoc data structures (supporting drift-detection and classifier-induction tasks). Moreover, compared to these two previous works, a more systematic presentation of related research in the field of ensemble-based intrusion detection and of background concepts is provided here, as well as a more comprehensive experimental analysis (including wider ranges of competitors, evaluation metrics, drift-detection methods, and classifier-replacement strategies).

The rest of the paper is structured as follows. After providing an overview of related work in Sect. 2, some basic concepts and notation are introduced in 3, in order to have a convenient formal basis for presenting our online classification framework. The latter is described in detail in Sect. 4, which contains an algorithmic description of the proposed learning-and-prediction stream processing strategy. The conceptual system architecture adopted to implement the framework is illustrated in Sect. 5, which also provides some major implementation-related details. Section 6 discusses experiments performed on several benchmark datasets, which offer empirical evidence for our proposal's validity. Final remarks and future work are discussed in Sect. 7.

2 Related works

Machine learning and data mining methods were widely used to support the development of cybersecurity solutions, especially in intrusion detection systems (Buczak and Guven 2016). However, traditional methods of this kind tend to suffer from serious limitations, which make them unsuitable for many real-life cybersecurity applications: they cannot handle concept drifts, they cannot work in an incremental way, and they cannot handle big volumes of data and streaming data.

A more promising area of research for this field concerns information fusion and ensemble-based approaches (Folino and Sabatino 2016). Indeed, classification techniques leveraging these approaches tend to work well not only when big data are supplied as input, but also when few training data are available. In addition, they can easily exploit the advantages of distributed environments such as parallel, GPGPU architectures, and P2P and Cloud computing architectures. Finally, they can easily model different abstractions or parts of a network, i.e., some models can be trained on some parts or on some levels of the network and finally combined together, to ensure better predictions.

Differently from our approach, most of these techniques employ simple combiner functions or weighting schemes for merging the predictions of multiple base models in the ensemble. For instance, Gao et al. (2019) propose an adaptive ensemble learning model employing several traditional base learners such as DT, SVM, logistic regression, k-NN and DNN. By using statistical tests, the five most promising classifiers are chosen and exploited to accurately tune the base model parameters. Then, a weighted voting schema is used to provide the final classification. The experimentation shows an important improvement in terms of detection accuracy.

Similar limitations affect the two works (Perdisci et al. 2009; Borji 2007) that introduced the idea of selecting the combiner function from a predefined range of non-trainable functions. In particular, Perdisci et al. (2009) proposed to discover an ensemble of SVMs to detect intrusion attacks in a software network. To this end, a clustering-based algorithm is first exploited to reduce the dimension of the feature space. Then, one SVM for each representation of the payload is trained in order to model the normal network traffic. The combination logics of the ensemble is defined as a non-trainable function, chosen among common aggregation functions (returning the average, product, minimum and maximum of the base models' predictions, respectively). Experiments performed on the DARPA and GATECH datasets showed this approach to achieve good performances, especially in terms of small false-positive rate.

A heterogeneous ensemble of classifiers (discovered with the help of ANN, SVM, decision tree and k-NN methods) were used in Borji (2007) in order to identify attacks in the DARPA dataset. In addition to the classical average or majority-voting functions, a further kind of combiner function was considered there, which rely on estimating the probabilities that a pattern assigned to a given data class actually belongs to that class or to other classes.

The rest of this section focuses on works that have used ensemble-based methods together with evolutionary algorithms, similarly to our proposal. Notice, however, that in many of these works evolutionary algorithms were only employed for the construction/selection of the base classifiers (de Oliveira et al. 2009; Folino et al. 2008) and not, as in our technique, for generating the combiner function.

Sindhu et al. (2012) proposed to adopt an ensemble of shallow neural networks and compared their approach with the AdaBoost algorithm and with other approaches. In order to select the optimal subset of the features from the dataset, a genetic algorithm was used. Experiments performed on the KDD'99 dataset showed that the proposed ensemble performs better than the AdaBoost algorithm and a neural network-based ensemble with respect to different metrics, i.e., true-positive and false-positive rate, precision, recall and F-measure.

In Folino et al. (2016a), the authors introduced a meta-ensemble based on a non-trainable function and evolved by an evolutionary-based algorithm to classify intrusions. Experiments, performed on the KDD and on the ISCX dataset, demonstrate the goodness of the approach to cope with unbalanced classes, by using specialized classifiers. In addition, the good performance was achieved in terms of low false-alarm rate; however, the approach suffers from some limitations in recognizing all the kinds of attack.

In Kumar (2020), different Pareto-optimal sets of base models are combined through some majority methods, and evaluated on the well-known NSL_KDD and ISCX-2012 datasets. The approach has the advantage of providing classification trade-offs for cybersecurity administrators and managers. However, the training phase and the generation of the weights of the combiner functions (based on multi-objective optimization and neural networks) are computationally expensive.

In Aburomman and Reaz (2017), particle swarm optimization (PSO) methods were employed to generate the weights of the combiner function for 12 classifiers. Different ways to generate the weights were empirically compared on five randomly selected subsets of the KDD99 datasets, eventually demonstrating that the new method gives better accuracy than weighted majority algorithm (WMA).

3 Background and formal framework

This section introduces some basic concepts and notation that concern: (i) the kinds of data and classification models considered in our work (Sect. 3.1); (ii) ensemble-based classification and the specific combining strategy, relying on "non-trainable" combiners, that underlies our framework (Sect. 3.2). The section also presents (in Sect. 3.3) background concepts regarding Genetic Programming (GP), and the specific GP tool exploited in the prototypal implementation of our approach for finding an optimal combiner.

3.1 Data tuples and classifiers

Conceptually, a data *tuple* (a.k.a. instance) is as pair (x, y), where x is a *feature vector*, storing m distinguished data fields, and y is a label representing the class of x, chosen among a given set $\Omega = \langle \omega_1, \omega_2 \ldots, \omega_c \rangle$ of class labels. For the sake of presentation, let us assume that all feature vectors belong to $\mathbb{R}^m$, and that an additional fictitious class label $\perp$ is used to denote *unlabelled* data tuples, i.e., any tuple for which the actual associated class label is unknown.

A *classifier* (or predictor) h is a model encoding a (classification) function of the form $h : \mathbb{R}^m \rightarrow [0, 1]^c$, which maps any feature vector $x \in \mathbb{R}^m$ to a vector of c *support degrees*, one for each of the classes that are referred to in Ω. Specifically, the j-th component of the resulting vector $h(x)$, denoted hereinafter by $h(x)[j]$, represents the degree of support assigned by h to the hypothesis that x belongs to the j-th class ω_j, for any $j \in [1 \ldots c]$—the larger the support, the more likely x is estimated to belong to ω_j. Without loss of generality, we can assume that all the c degrees, returned by any classifier h, are in the interval [0, 1] and that they all sum up to 1, i.e., $h(x)[j] \in [0, 1]$ for $j \in [1..c]$ and $\sum_{j=1}^{c} h(x)[j] = 1$.

A classifier can be discovered from a given (training) set $S = \{(x_i, y_i) \mid i = 1, \ldots, N \text{ and } y_j \neq \perp\}$ by resorting to one of the many standard (propositional) classifier-induction algorithms available in the literature, and then used to predict the class of novel (unlabelled) tuples.

3.2 Ensembles and non-trainable functions

Ensemble methods permit to combine multiple (heterogeneous or homogeneous) classifiers, in order to obtain a more expressive/robust scheme for classifying unseen instances. In practice, after a number of classifiers have been built (using training data), for each new unlabelled feature vector x, the predictions of the different classifiers for x are combined so that a common decision can be eventually taken on the class of x. Different methods can be adopted 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 to build the different classifiers/models.

A different approach is followed by the popular *boosting* method introduced by Schapire (1990) and Freund (1995); 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" (Schapire 1995)), 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 variety of datasets. However, it has the drawback of needing to repeat the training phase for a number of rounds, which could be too time-consuming on large datasets.

The applications and data characterizing domains like cybersecurity pose strong efficiency requirements, which do not permit to re-train the base classifiers. On the contrary, non-boosting-based ensemble strategies do not need any further phase of training, when the predictions of the base classifiers can be combined without using the original training set. The majority vote is a classic example of this kind of combiner function. Some types of combiner, known as *non-trainable* combiners (Kuncheva 2004), 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.

Let us define next some more basic concepts and notation, in order to ease the presentation of our approach.

In an ensemble consisting of g classifiers, every time a new feature vector is to be classified, a collection of $g \times c$ support degrees are returned for x by the classifiers, which need to be integrated in some way. The following definition introduces a concept for representing such a collection of classification degrees uniformly and conveniently.

Definition 1 (*Decision Profile*) Let $x \in \mathbb{R}^m$ be a feature vector, $L = h_1, \ldots, h_g$ be a list of g classifiers, all trained over c given classes labels $\Omega = \omega_1, \ldots, \omega_c$. Then the *decision profile* $H^L(x)$ of L for x is a matrix containing all the degrees of support that are assigned, relatively to x, by classifiers of L to the classes (referred to) in Ω:

$$H^L(x) = \begin{bmatrix} H^L_{1,1}(x) \ldots H^L_{1,j}(x) \ldots H^L_{1,c}(x) \\ H^L_{i,1}(x) \ldots H^L_{i,j}(x) \ldots H^L_{i,c}(x) \\ H^L_{g,1}(x) \ldots H^L_{g,j}(x) \ldots H^L_{g,c}(x) \end{bmatrix}$$

where the element $H^L_{i,j}(x)$ is the degree of support $h_i(x)[j]$ assigned to j-th class by the i-th classifier. □

Let us now introduce a concept concerning the combination of base classifiers' predictions.

Definition 2 (*Combiner*) Let $L = h_1, \ldots, h_g$ be a list of g classifiers, trained over the class labels $\Omega = \omega_1, \ldots, \omega_c$. Then, a *combiner* for L is a function $F : [0, 1]^{g \times c} \rightarrow [0, 1]^c$ that returns an overall vector of support degrees for from those stored (per row) in any decision profile $H^L(x)$ returned by L, for some $x \in \mathbb{R}^m$, by specifically deriving the support degree of the j-th class from the values in the j-th column of $H^L(x)$. By composing L and F, an (ensemble) classifier $\mu^{L,F} : \mathbb{R}^m \rightarrow [0, 1]^c$ can be defined, which maps any feature vector $x \in \mathbb{R}^m$ to $F(H^L(x))$. The j-th component of $\mu^{L,F}(x)$, denoted as $\mu^{L,F}(x)[j]$, represents the combined support degree of class ω_j for x. □

As an instance, a simple instantiation of the combiner function F consists in averaging, for each class, the predictions of the base classifiers, so that the support degree of the j-th class is computed as: $\sum_{i=1}^{g} H_{i,j}(x)$.

Clearly, when using the ensemble-like classifier $\mu^{L,F}$ as a hard classifier (as done in our framework), the class label that will be eventually assigned to a feature vector x is the label ω_{j^*} that gets the highest support on x, i.e., such that $j^* = \arg\max_j\{\mu^{L,F}(x)[j]\}$.

3.3 Genetic programming basics and the CAGE tool

Genetic Programming (GP) is a sub-class of Evolutionary Algorithms, inspired by the evolutionary theories of Darwin,

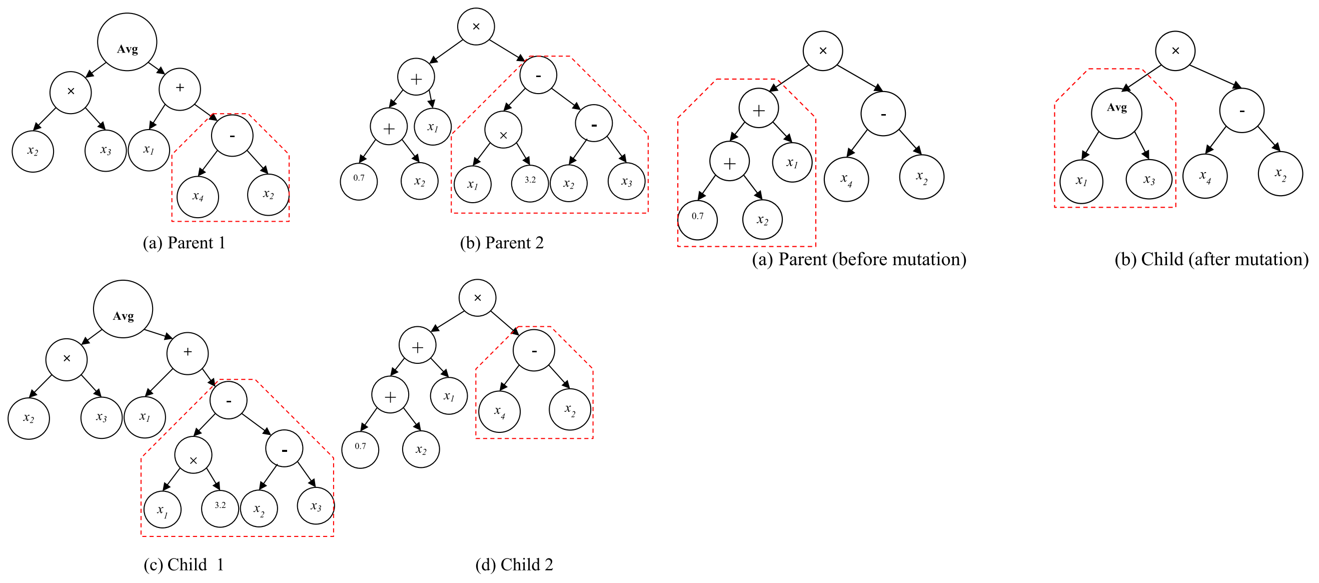

Fig. 1 An example of GP crossover (left) and mutation (right). In GP crossover two random subtrees of the parents are selected and swapped and generate two new individuals. Here, the function set contains Avg, $+$, $-$ and $\times$, and the terminal set contains problem variables and some random numbers. In GP mutation, a random subtree of the parent is selected and substituted with a new random subtree

that it is meant to find solutions to a computational problem by evolving a population of solutions (*individuals* or *chromosomes*) for a number of rounds (*generations*). The individuals of a standard GP approaches are trees that encode some kind of program. The internal nodes of the tree are functions and the leaves are typically the problem variables, constants or random numbers.

The initial population of GP is a set of trees generated randomly. During the evolutionary process, the individuals are evolved until an optimal solution (or a good approximation of it) is found, or a maximum number of generations is reached. The evolution is driven by a function of *fitness*, which is chosen for the particular problem to be solved and represents the goodness of a solution/individual.

Similarly to other evolutionary algorithms, for each generation, two genetic operators (crossover and mutation) are performed on some individuals, chosen randomly on the basis of their fitness: individuals with better fitness have more chance to be chosen. The crossover operator swaps two random subtrees of two individuals (parents) and generates two new individuals (children), whereas the mutation operator, which is performed on a single individual, transforms a random subtree and generates a new individual. Figure 1 shows an example of the crossover and mutation operator.

The newly generated individuals are added to the populations and compete with other individuals based on their fitness, i.e., the better individuals have more chance to survive. This process leads to find better solutions along the evolution of the process.

The CAGE tool In order to evolve the combiner function of an ensemble, our framework employs the *CellulAr GEnetic programming* (CAGE) tool (Folino et al. 2003) as its underlying GP engine.

This particular choice was mainly taken for efficiency and scalability reasons. The tool, based on the fine-grained *cellular model*, can run indeed on top of both distributed-memory parallel computers and distributed environments.

The overall population of the GP algorithm is partitioned into subpopulations of the same size. Each subpopulation can be assigned to one processor running a standard (*panmictic*) GP algorithm. 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 main difference between a cellular GP model and the panmictic come from the fact that the former employs a decentralized neighbor-based selection mechanism for evolving a population. Indeed, in the cellular model each individual is associated with a spatial location and a small neighborhood, and it can only interact with its neighbors.

In the prototypal implementation of our framework, the logics underlying the generation process and the genetic operators are the same as in the classical GP formulation introduced by Koza (1992), and summarized before.

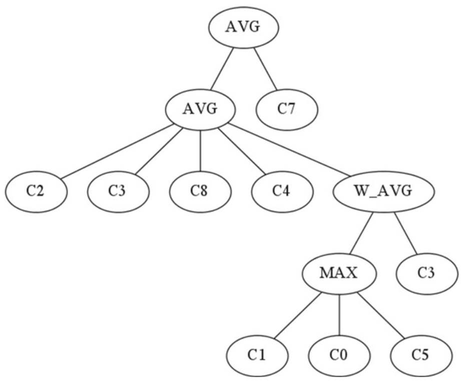

Fig. 2 An example combiner function F_E that could be generated with our approach for the classifier ensemble E

4 Proposed framework: stream-processing scheme

This section describes our approach to the ensemble-based classification of streaming intrusion detection data, which is meant to support a continuous learning-and-prediction processing flow over such data.

The proposed stream processing flow is illustrated in detail in Sect. 4.2 in the form of an algorithm, named *Evolutionary Ensemble-based Stream Classification for Intrusion Detection* (or simply $E2SC4ID$ for short).

For the sake of readability, in Sect. 4.1 we introduce some major data structures that are used in our approach, in order to maintain updated high-level information on both the data distribution and the discovered classification models.

4.1 Data structures

Ensemble model and associated GP-based combiner The proposed ID framework relies on processing a stream $D = d_0, d_1, \ldots$ of log data, containing both labelled and unlabelled data tuples, with the help of an ensemble model $E = \langle B_E, F_E \rangle$, which consists of two components:

– a list B_E of base classifiers, say $h_1, \ldots, h_g$, such that each h_j encodes a function mapping any data tuple d of D to the space of the reference intrusion-related classes (cf. the notation introduced in Sect. 3);
– a combiner F_E, taking the form of a GP tree (described later on), which allows for mapping the attribute vector x of any tuple $d \in D$ to a vector $\mu^{B_E, F_E}(x) \in [0, 1]^c$ of overall predictions, based on the predictions (namely, $h_1(x), \ldots, h_g(x)$) returned for x by the base classifiers in B_E (see Definition 2 for details).

The combiner function F_E in the proposed ensemble model is encoded as a GP tree, where each leaf corresponds to one of the base classifiers in B_E, while the other nodes may be associated with non-trainable aggregation functions chosen among the following ones: *average, weighted average, maximum* and *median*. The choice of using these particular non-trainable functions descends from the fact that they were widely shown to work well in practice (see, e.g., the extensive experimental analysis discussed in Kuncheva (2004)[1]). For the sake of concreteness, Fig. 2 shows pictorially one of the combiner functions generated by our approach in the experimentation described in Sect. 6.

In more detail, in any GP-tree combiner, every leaf w represents the general result that would be obtained by applying some of the base classifiers (i.e., the one associated with w, chosen among those in B_E) to a generic feature vector x. Let us employ $w(x)$ to denote the result of this operation, which is a vector of per-class support degrees, and $w(x)[j]$ to denote the j-th component of this vector (i.e., the support degree assigned to the j-th class).

Every non-leaf node v represents the result of the (possibly partial) computation encoded by the subtree rooted in v, which consists in applying the aggregation function associated with v to the results returned by the children of v, in a recursive fashion (until arriving at a leaf w, which is computed as specified before). Let $v(x)$ denote the result produced by the sub-tree rooted in v when (its leaves are) applied to a generic feature vector x. Then, $v(x)$ is computed by aggregating the results returned by the children of v by using the aggregation function associated with v itself. For example, for a non-leaf node v associated with the *average* function, it is: $v(x)[j] = \frac{1}{|\text{children}(v)|} \sum_{v' \in \text{children}(v)} v'(x)[j]$, for $j \in \{1, \ldots, c\}$. Similar definitions apply to the other aggregation functions (namely, *median, weighted average, and maximum*). In particular, the *weighted average* function is a variant of the average function where each base classifier is associated with a weight, previously computed on the training set.

Drift detection model An important kind of auxiliary data structure that is employed in our stream processing approach is a *drift-detection model*. Such a model constitutes indeed one of the main parameters of algorithm $E2SC4ID$, which could be instantiated with any of the techniques available in the literature for the detection of concept drifts. In our framework, a drift detection model DM is meant to encapsulate the statistics and computation logics that are exploited to detect a concept drift, and to provide an abstract Boolean method $DM.update\&check(x, y, \hat{y})$, which performs the following actions: (i) updating the internal statistics of the model to accommodate any newly arrived labelled tuple (x, y) and

[1] The only aggregation function, among those tested in Kuncheva (2004), that has not been integrated in our framework is the *product*, which was actually shown to not perform well enough in the general case of multi-class classification settings.

```
Input:
    an input data stream $D$, consisting of a continuous flow of data tuples of the form $(x,y)$, where $y$ may be either the label of
    a real class or the special symbol $\perp$ (in the case of a non classified tuple);
    an output data stream $O$ representing a modified version of $D$ where each tuple of $D$ is re-assigned the class predicted for it;
Output:
    an ensemble model $E = \langle B_E, F_E \rangle$ (as defined in Section 4.1);
Parameters:
    a list $learners$ of classifier-induction algorithms to be used as base learners; // see Sect. 5.4 for the details on these algorithms
    maximal number $l$ of base classifiers to select at each learning session (with $l < |learners|$);
    maximal number $maxC$ of base classifiers that can be kept in $B_E$;
    fixed number $n$ of tuples per data chunk (playing as "window size");
    a drift detection model $DM$ (implementing the method $DM.update\&check$ for incremental concept-drift detection);
    a temporally-ordered list $InitializerData$ of tuples to be used for initializing the models $E$ and $DM$.
Method:
        // Part I: Variables' initialization
    a1 $Buffer := InitializerData$;
    a2 $DM.initialize(Buffer)$; // the implementation of this step depends on the specific type of DM employed
    a3 Induce an ensemble model $E$ by performing the Steps 11-18 of procedure processTuple; // this eventually empties Buffer
    a4 $drift :=$ false; // boolean used by procedure processTuple to flag the occurrence of concept drifts
        // Part II: Tuple-wise incremental learning and classification
    a5 do
    a6     read a new tuple $(x,y)$ from $D$ (as soon as it arrives);
    a7     $(\hat{y}, E, Buffer, DM, drift) :=$ processTuple$((x,y), E, Buffer, DM, drift, learners, l, maxC, n)$;
    a8     write tuple $(x,\hat{y})$ onto the output stream $O$;
    a9 until ($D$ has ended or the algorithm has been interrupted by the user);
   a10 return $E$;
```

Fig. 3 Algorithm $E2SC4ID$ (pseudo code)

the class $\hat{y}$ predicted for it;[2] and (ii) checking whether there has been a drift in the recent portion of data stream ending with this tuple. The value returned by this method is `true` iff such a drift has been estimated to occur.

Any drift-detection model DM is also expected to provide a method $DM.initialize$ for initializing the model, based on a given set of tuples (to be passed as input to the method).

Clearly, the actual implementation of model DM depends on the particular drift-detection strategy chosen by the user. More details on this respect are provided in Sect. 5.3, which briefly illustrates the alternative drift-detection models that have been integrated in the current implementation of our framework,

4.2 Algorithm E2SC4ID

Both the base classifiers and the ensemble model E are built through a continuous learning-and-prediction strategy, where incoming labelled tuples are used as training examples, and unlabelled ones are classified with E as soon as they arrive. This data-processing scheme is described in Fig. 3 in the form of an algorithm, taking the same name, i.e. $E2SC4ID$, as the entire framework proposed in this work.

The algorithm takes, as its main input, a data stream D providing a continuous flow of data tuples (to be classified or used as training examples). All these tuples are assumed to be pairs of the form (x, y), where y can represent either the actual class of the feature vector x (in case this class is known a priori) or the special symbol $\perp$ (in the case the class of x is unknown, and it must be assigned by the model). A further auxiliary output stream O is taken as input, which is just meant to represent a "classified version" of D. Stream O is indeed populated by the algorithm, which inserts a modified version $(x, \hat{y})$ of each data tuple (x, y) in D, as soon as a prediction $\hat{y}$ has been made for x.

Basically, the algorithm incrementally discovers an ensemble model $E = \langle B_E, F_E \rangle$ (of the form described in Sect. 4.1) from the input stream D. As explained in detail later on, this stream is split conceptually into a sequence $W_0, W_1, \ldots, W_i, W_{i+1}, \ldots$ of non-overlapping windows, named hereinafter *chunks*, which all contain the same fixed number n of labelled tuples.[3] More precisely, denoting by $D^{Lab} = d_0^{Lab}, d_1^{Lab}, \ldots$ the "filtered" version of D featuring only the labelled tuples of D, each chunk W_i consists of the following n (temporally contiguous) labelled tuples: $\{d_j^{Lab} \mid j \in \mathbb{N}^0,\ j \geq i \times n,\ j < (i+1) \times n\}$. Such a temporal segmentation of the input data stream serves the goal of building up a series of training sets W_i for the induction of new base classifiers, such that: (i) each of them is large enough (and hence representative enough of the behavior of system in the respective time window), but (ii) it is unlikely that many of these training sets are too heterogeneous internally (i.e., feature each too many different attack patterns).

[2] In fact, our MOA-based implementations of such models only takes account for the class labels, and disregard the feature vector x.

[3] The choice of using fixed-size windows is mainly for the sake of concreteness and of presentation. In fact, our approach can be easily extended to deal with other data-segmentation schemes.

```
Input:
  a tuple $(x, y)$, where $y$ may be either a real class label or the special symbol $\perp$ (in the case of an unlabelled tuple);
  an ensemble model $E = \langle B_E, F_E \rangle$, where $B_E$ is the list of base classifiers and $F_E$ is the combiner function;
  a list Buffer of tuples (recent tuples, kept in temporal order);
  a drift detection model DM (implementing the method DM.update&check for incremental concept-drift detection);
  a boolean drift-detection flag drift;
  a list learners of classifier-induction algorithms to be used as base learners; // see Sect. 5.4 for details on these algorithms
  maximal number $l$ of base classifiers to select at each learning session (with $l < |learners|$);
  maximal number maxC of base classifiers that can be kept in $B_E$;
  fixed number $n$ of tuples per data chunk (playing as "window size");
Output:
  the class predicted for $x$ (only for debug purposes when the input tuple is already labelled);
  updated versions of the ensemble model $E$, tuple list Buffer, and drift-detection model DM (in the actual implementation of this
  procedure, $E$, Buffer and DM are all passed by reference and modified directly, without returning any copy of them as output);
  the novel value of flag drift;
Method:
 1 // Phase I: Classify $x$ using the current version of $E$
 2 Build the decision-profile matrix $H^{B_E}$ of $B_E$ for $x$ // cf. Def. 1
 3 Compute a global support degree $\mu_j^{B_E,F_E}(x)$ for each class $\omega_j$ by applying combiner $F_E$ to the $j$-th column of $H^L$; // cf. Def. 2
 4 Compute the predicted class label $\hat{y}$ of $x$ as follows $\hat{y} := \arg\max_j\{\mu(x)[j]\}$;
 5 if $(y \neq \perp)$ then // a labelled tuple is arrived
 6   Insert $(x, y)$ into Buffer;
 7   drift := drift or DM.update&check$(x, y, \hat{y})$; // updates the state of DM and returns a drift detection flag (see Sec. 5.3)
 8   if $(|Buffer| = n)$ then // a novel chunk of training data has become available
 9     if (drift) then
10       drift := false;
         // Phase II: Extend the list of base classifiers in the ensemble model E
11       Partition Buffer into a training set $Train$ and a validation set $Valid$;
12       Build a set NewModels of classifiers by running each of the methods in learners on $Train$;
13       Insert into $B_E$ the $l$ classifiers in NewModels that achieve the highest $l$ accuracy scores on $Train$;
14       if $(|B_E| > maxC)$
15         Select and remove $maxC - |B_E|$ classifiers in $B_E$; // according to some chosen replacement criterion (see Sec. 5.2)
16       end if
         // Phase III: Update the combiner of the ensemble model E
17       Compute the decision profile $H^{B_E}(x')$ of all tuples $(x', y') \in Valid$, and store them all into a list DP; // cf. Def. 1)
18       $F_E$ := computeGPcombiner(DP, Valid); // compute a novel GP-based combiner by using the CAGE tool (see Sec. 5.1)
19     end if
20     Empty Buffer; // a new data chunk will start being formed at next run of the algorithm
21   end if
22 end if
23 return ($\hat{y}$, E, Buffer, DM, drift);
```

Fig. 4 The incremental stream-processing procedure `processTuple` used in algorithm $E2SC4ID$ (pseudo code)

The algorithm consists of two parts, which encode respectively: (I) the initialization of the ensemble model E and of other auxiliary variables (Steps a1–a4); and (II) a tuple-wise stream-processing scheme, which relies on iteratively calling an ad hoc auxiliary procedure, named `processTuple`, on each of the tuples appearing in the input stream D (Steps a5–a10).

These two phases are illustrated in Sect. 4.2.1 (which also describes the parameters and variables used by the algorithm), and 4.2.2 (which also describes the core procedure `processTuple`), respectively.

4.2.1 Parameters, variables and initialization phase

The algorithm is assumed to be provided with the following parameters, complementing the two data streams D and O: (i) the list *learners* of classifier-induction algorithms to be used for training new base classifiers (in the prototype implementation of our approach, these algorithms were chosen among those available in the MOA tool, as discussed in Sect. 5.4); (ii) the maximal number l of base classifiers to be selected at each learning session; (iii) the maximal number *maxClass* of base classifiers that can be in the ensemble; (iv) the size n fixed for all the chunks; (v) a *drift-detection model DM* (of the kind described in Sect. 4.1, and equipped with suitable implementations of the incremental drift-detection method $DM.update\&check$ and of method $DM.initialize$); and (vi) a temporally ordered list *InitializerData* of labelled tuples for initializing both models E and DM.

Notice that in the experiments described in Sect. 6, we always set parameter *InitializerData* as to contain the first chunk (i.e., the oldest n tuples) of the dataset at hand.

The first part of the algorithm (Steps a1–a4) is devoted to initialize the ensemble model E, the drift-detection model DM, and two auxiliary variables, which all need to be passed in each of the calls of procedure `processTuple` that will be made in the rest of the algorithm (Steps a5–a10). These auxiliary variables are: (i) a Boolean variable *drift*, used in procedure `processTuple` to flag a detected occurrence of concept drift, which is simply initialized to `false`; and (ii) a list *Buffer* of tuples, devoted to maintain the (labelled) tuples of the current data chunk (ordered according their respective times of arrival), which is initially set as empty.

Both E and DM are initialized by exploiting the contents of parameter *InitializerData* (actually copied in *Buffer* in the algorithm, just for the sake of presentation). In particular, case of the ensemble model E, this is specifically performed by running the Steps 11–18 of the procedure `processTuple` shown in Fig. 4 (see Sect. 4.2.2 for a detailed description of these steps). The initialization of the DM depends instead on the chosen kind of drift-detection model (and on its associated implementation of method $DM.initialize$).

4.2.2 Tuple-wise processing via procedure `processTuple`

The second part of Algorithm E2SC4ID (Steps a5–a10 in Fig. 3) amounts to applying procedure `processTuple` iteratively to each of the tuples (x, y) that appear in the input stream D. Even though such a stream-processing loop might go on forever, we here assume that it can be stopped by the user or it ends naturally when no more tuples are expected to arrive from D.

Essentially, every execution of procedure `processTuple` takes as input the current tuple (x, y) that needs to be processed, together with the variables *E, DM, Buffer* and *drift* and the constant parameters (namely, *learners, l, maxC* and n) described in the previous subsection.

As a result, the procedure returns the label $\hat{y}$ of the class predicted for x, which is used as the class label of x within the modified version of the tuple that is inserted in the auxiliary stream O (Step a8 in Fig. 3). Every invocation of the procedure `processTuple` also returns an updated version of the variables *E, DM, Buffer* and *drift*, in order to allow the algorithm to support an incremental processing/learning scheme, by keeping (across different executions of the procedure) updated information on both the raw data processed and the (drift-detection and ensemble-classification) models discovered.

The computation steps of procedure `processTuple` are summarized in Fig. 4. The former four steps of the procedure are devoted to predict a class label $\hat{y}$ for x, which will be eventually returned as output by the procedure in the final step. This is done by: (i) first computing, for x, the decision profile H^{B_E} of the base classifiers currently associated with the ensemble E (Step 2); then (ii) deriving an overall support-degree vector μ^{B_E,F_E} from H^{B_E} with the help of the combiner F_E of the ensemble E (Step 2); and (iii) finally selecting the class achieving the highest (combined) support degree (Step 3).

Steps 6–20, executed only when x is a labelled tuple, are meant to update the global parameters *Buffer* and DM as to take account for the information conveyed by x and its associated class label y, as well as to possibly update the flag *drift* and the ensemble E in case a concept drift has been detected by DM (through the invocation of its method *update&check* in Step 7). In fact, in such a case the update of the ensemble E is carried out in Steps 10–18 only when a sufficient number (namely, n) of training tuples have been collected in *Buffer*; otherwise this operation is postponed to a subsequent execution of the procedure (i.e., as soon as the size of *Buffer* has grown to n), while leaving the flag *drift* unchanged.

Let us now focus on the case when *Buffer* contains all the n tuples of a chunk, say W_i, and, in addition, *drift* $=$ `true`, meaning that a concept drift has been detected by DM either at the current execution of the procedure or in a previous execution (where there were no sufficient tuples to complete the chunk). In such a case, the (labelled) tuples of *Buffer* are randomly split into two equally sized sets: a training subset $Train$ and a validation set $Valid$. The former set is used to induce a number of novel base classifiers by using the classifier-induction procedure listed in *learners*. Among these newly discovered classifiers, the l ones having the best accuracy (on the same training set $Train$) are chosen and added to list B^E of the ensemble. In case this list has grown over the maximum allowed size *maxC*, *maxC* $- |B_E|$ base classifiers are removed from the list, so that the size constraint for the ensemble is satisfied. The combination of the insertion and removal operations, performed throughout Steps 13–16 in such a case, can be regarded as a *replacement* operation, where a selection of the base classifiers of E (among those present in B_E before Step 13) is substituted with "better" classifiers among those that have been trained on the current chunk W_i. More details on this respect can be found in Sect. 5.2, which specifically illustrates three alternative replacement strategies that have been integrated in the current prototype implementation of the framework.

At this point, a novel GP-tree combiner F_E is derived for E, in Steps 17–18, by running a GP procedure, named `combineGPcombiner`, that evolves a population of GP-trees, and selects an individual that allows the ensemble to maximize its classification accuracy over the tuples of $Valid$. Indeed, as explained in more detail in Sect. 5.1, the fitness adopted in this procedure to test each candidate combiner T is defined in terms of the classification errors that T makes over the labelled tuples in $Valid$.

For the sake of efficiency, in Step 17 the procedure preliminary applies the base classifiers of E to all the tuples in $Valid$, and stores the resulting tuple-wise vectors of per-class support degrees into a list *DP* of decision profiles (with a distinct decision profile for each validation tuple). This way, it is not necessary to re-compute of all these support degrees for every candidate GP-tree combiner T generated in Step 18. Indeed, in order to evaluated the fitness of such a combiner T it is sufficient to compare the ground-truth class label y' of each tuple (x', y') in $Valid$ with the class that T eventually predicts for x' by combining the predictions of the base classifiers (already stored in *DS*). Further details on the actual implementation of function `combineGPcombiner` are given in Sect. 5.1.

Remark The combiner function F_E that is returned by function `computeGPcombiner` may well encode a combination scheme that only uses a subset of the base classifiers in B_E, as clarified in Sect. 5.1. This allows us to possibly operate a more careful (fitness-driven) selection of the base classifiers than the preliminary coarse-grain greedy selection that was performed in Step 15, mainly for the sake of computational efficiency (owing to the fact that reducing the number of base classifiers to be combined, allows function `combineGPcombiner` to focus on a smaller search space).

5 Proposed framework: system architecture

This section is devoted to present the conceptual system architecture that has been adopted to implement the proposed framework, in order to eventually enable for a concrete continuous application of algorithm $E2SC4ID$ (defined in Fig. 3) to a stream of real-life intrusion detection data.

This architecture, sketched in Fig. 5. features several functional modules that are in charge of supporting the main kinds of data-stream processing tasks involved in the approach, which primarily include: the detection of concept drifts, the incremental chunk-wise update of a classifier ensemble, the application of the classifier ensemble to new data tuples, and the detection and handling of (either real or presumed) intrusion attacks.

The system ingests a stream of data coming, in a continuous fashion, from different sources, such as network traffic logs (e.g., coming from a network's router or interface), system logs, application logs. In order to turn such a stream of data into the form of a series of (labelled/unlabelled) tuples considered in our framework, some data pre-processing module need to be put into place, e.g., supporting some ad hoc formatting, filtering, and feature engineering/selection tasks. This module is not shown in the figure, where the data stream is assumed to consist of (pre-processed) tuples. Clearly, the implementation of such a module strongly depends on the data source and the application domain. As a matter of fact, the current version of our prototype system includes pre-processing functionalities specifically designed for the case of network traffic data in the standard `pcap` (packet capture) format.

The rest of this section is devoted to illustrate the different modules of the proposed ensemble-based IDS:

1. Module *Combiner Generation*, which is devoted to generate a new combiner for the ensemble, by suitably implementing the procedure `computeGPcombiner` of algorithm $E2SC4ID$ (see Fig. 3). This module, which is based upon the GP tool *CAGE*, is described in detail in Sect. 5.1.
2. Module *Classifier Replacement*, which implements the logics for selecting the "victim" base classifiers, among those forming the current ensemble, that will be replaced with newly discovered classifiers. In general, such a selection could be performed on the basis of accuracy and/or redundancy/diversity criteria. Details on the replacement methods implemented in our prototype system are presented in Sect. 5.2.
3. Module *Drift Detection*, which is in charge of carrying out an incremental real-time analysis of the data stream's distribution, and possibly detect relevant changes that witness a recent occurrence of a concept drift. The module implements different Drift-Detection models (in order to instantiate parameter DM of algorithm $E2SC4ID$), which allow for incrementally extracting and analyzing aggregated statistics supporting the detection of concept drifts. More details on this respect are given in Sect. 5.3, which describes the three drift-detection models that have been integrated in the prototype system.
4. Module *Classifier Induction*, which is responsible for discovering novel base classifiers from the current data chunk, in order be possibly insert an optimal subset of them into classifier ensemble. To this end, the module implements a number of machine-learning algorithms for supervised classification, which are listed in Sect. 5.4.

The remaining module in the figure, namely *Classification & Alert Generation*, is devoted to classify every incoming unlabelled tuple, using the ensemble model E, and to generate and register an alert whenever an intrusion attack is recognized, based on the result of this classification.

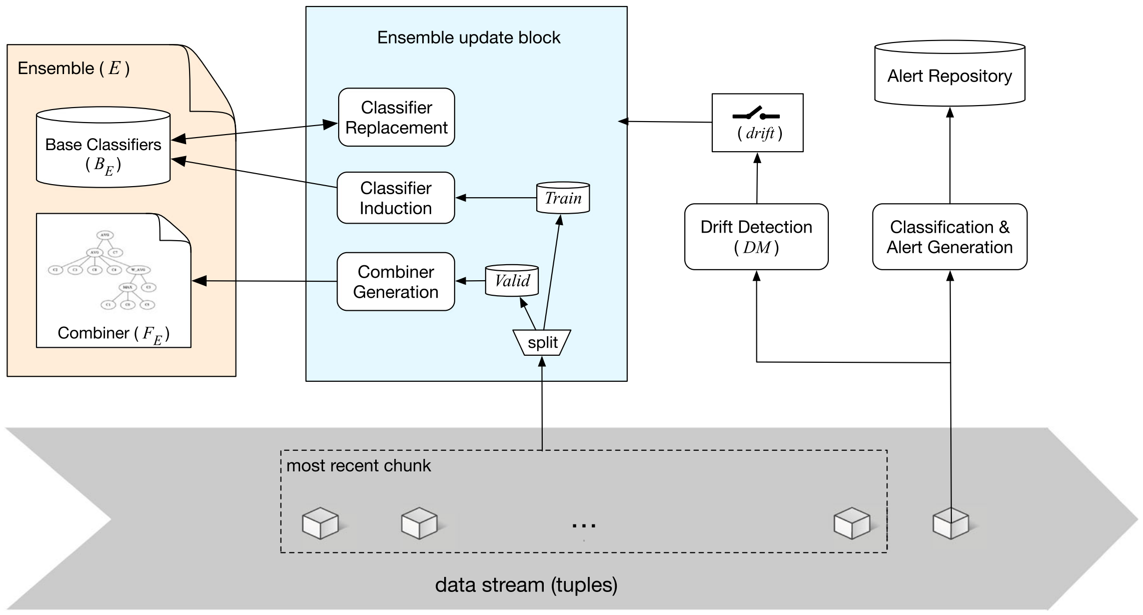

Fig. 5 System Architecture. Please notice that the input data are assumed to be already put into the form of tuples, which are all depicted as small cubes. For the sake of presentation, it is assumed that a drift has been detected correspondingly to the current tuple (i.e., the rightmost cube), and that a chunk (formed by labelled tuples arrived before) is available for updating the ensemble model. Some objects refer the names (in italics) of related variables of algorithm $E2SC4ID$

5.1 Combiner generation

As described in Sect. 4, the revision of the classifier ensemble E (which is at the basis of our approach, and is triggered by drift-detection events) is performed in three phases: (i) discovering novel base classifiers from the most recent data chunk; (ii) extending the list B_E of the base classifiers composing E with a selection of classifiers discovered in the previous phase, while possibly discarding some older elements of B_E according to a replacement strategy; and (iii) generating a new, optimal, combiner F_E for E through a GP-based evolutionary computation scheme. In what follows we only focus on the last phase, which is directly supported by module *Combiner Generation*, and refer the reader to Sects. 5.4 and 5.2 for a more detailed discussion of the former two phases.

In order to find an optimal combiner F_E for the updated collection B_E of base classifiers, the module implements a GP search procedure, which is briefly described below.

Basically, every individual in this search, representing a candidate combiner function for the ensemble E, is encoded as a GP tree of the form described in Sect. 4.1. Let us recall that, in such a tree, each leaf corresponds to one of the base classifiers in B_E, while any internal node may be associated with a non-trainable aggregation function chosen among *average, weighted average, maximum* and *median*. In order to limit the representation complexity of the candidate GP-tree solutions to be explored (and thus restrict the search space), the number of children of every non-leaf node v (i.e., the arity of the aggregation function associated with v) can be only chosen in the set $\{2, 3, 5\}$.

A randomly created initial population of such GP trees is made evolve with the help of a distributed instantiation of the GP tool *CAGE* (see Sect. 3.3 (Folino et al. 2003)), according to the accuracy-based fitness measure described below.

Fitness measure: definition and (efficient) computation The quality of each candidate GP tree T, generated throughout the evolutionary search procedure, is evaluated with a fitness score that measures the accuracy that would be obtained when using T to combine the predictions returned, over the tuples of the validation set *Valid*, by base classifiers in B_E. This fitness score can be computed as the opposite of the ratio between the tuples of *Valid* that are misclassified and the total number of tuples in *Valid*. As a safer alternative for the case unbalanced datasets (which are not so rare in intrusion detection scenarios), a weighted scheme can be adopted to compute the fitness of a GP tree T, where (i) each misclassified tuple belonging to a "small"/minority class ω is associated with a weight that coincides with the ratio between the total number of tuples in *Valid* and the number of *Valid*'s tuples that belong to class ω (in case this weight exceeds the threshold value of 10, it is turned into 10, in order to prevent generating excessively high rescaling factors); while (ii) each misclassified tuple belonging to a "big"/majority class is associated with a fixed unitary weight, as in the standard case of balanced datasets.

Let us recall that, since T is composed of non-trainable aggregation functions, no extra computations are needed on other training data. Moreover, the pre-computation of the decision profiles of all the tuples in $Valid$ allows for computing the fitness score of any candidate T in a fast way.

5.2 Model replacement

As specified before, in our framework the ensemble model is bound to contain a fixed number m of base classifiers at most (cf. Fig. 3). In order to adhere to this constraint, the Model Replacement module implements three alternative strategies for selecting a number of base classifiers to be removed, and thus making room for newly generated classifiers: *Random*, *Best* and *Wheel*, respectively.

Before describing these strategies, let us recall that in our approach, once detecting a concept drift, a collection *NewModels* of novel base classifiers is discovered from the most recent data chunk, by only using a subset *Train* of the data chunk (consisting of a randomly sampled half of the labelled tuple in the chunk), while keeping the remaining example tuples in a validation set *Valid* that is mainly used to search a novel combiner for the ensemble.

Let us denote as B_E^{old} the base classifiers that composed the ensemble model E before building the new classifiers *NewModels*. When a drift is detected, all the base classifiers (be them new classifiers of *NewModels* or existing classifiers of B_E^{old}) are trained again on the novel data chunk.

The three model replacement strategies are briefly explained in what follows:

- The *Random* strategy replaces 1/3 of the base classifiers in B_E^{old} with newly generated ones. The classifiers are removed on the basis of their insertion time: older classifiers are removed first. These classifiers are then replaced with a subset of those in *NewModels*, chosen on the basis of their accuracy on the validation set (clearly, more accurate classifiers are preferred).
- The *Best* strategy works removes a 1/3 of the base classifiers in B_E^{old}, as in the previous case, but selecting those with the worst accuracy scores over the validation set. These classifiers are replaced with the best ones in *NewModels*, still in terms of accuracy over the validation set.
- The *Wheel* strategy works according to a roulette-wheel mechanism. Then, 1/3 of these classifiers is chosen for removal through a sampling process where $p_i = \frac{f_i}{\sum_{j=1}^{N} f_j}$ is used as the probability of selecting the i-th classifier, and f_i is the accuracy of this classifier on the validation set.

5.3 Drift detection

This module analyses the data stream in search of possible changes in the distribution of the target classes (i.e., normal behaviors vs attacks), representing evidence for a concept drift. For the sake of efficiency (and suitability for an online ID setting), such analysis relies on the incremental computation of statistics for every new incoming data window D_i.

Since in this work we were not particularly interested in developing a new drift detection strategy, in the implementation of our approach we resorted to existing drift-detection models, leveraging their respective implementations in the MOA framework. Specifically, based on the thorough experimental analysis of different drift-detection models discussed in Gonçalves et al. (2014), we decided to equip the Drift Detection module with three alternative models, which obtained the lowest number of false alarms while maintaining a good overall accuracy: *DDM* (looking for changes in the classification error's rate), *STEPD* (based on statistically comparing groups recent tuples and of past tuples) and *ADWIN* (looking for distribution changes in sliding windows of variable size). We refer to the above-mentioned paper (Gonçalves et al. 2014) for a comprehensive comparison of different drift-detection methods. More details on these models are provided below:

- *STEPD* ("Statistical Test of Equal Proportions") (Nishida and Yamauchi 2007) is based on computing two statistics on the performances of a given classification model C: the overall (mean) accuracy of C from the beginning of the stream and the (mean) accuracy of C on a test window W. These two measures are compared statistically according to a Chi-square test: if the difference between them is lower than a given significance level, then the null hypothesis that the average accuracy of C has not changed is rejected, and a concept drift is detected. In more detail, three parameters are used in STEPD: the size of the time window expressed as number of data instances, and the significance levels α_w and α_d. STEPD stores the instances in its memory when $P < \alpha_w$ and it resets all the variables (i.e., its memory, the window accuracy, etc.) when $P < \alpha_d$.
- *DDM* ("Drift Detection Method") (Gama et al. 2004) relies on using a binomial distribution to model the probability of a random variable representing the number of classification errors made by a given classifier C in a sample of n examples. Correspondingly to each

Table 1 Main characteristics of the ISCX IDS dataset

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 2 Comparison of the different replacement strategies for the three drift detection algorithms (AWIN, DDM and STEPD) using the AUC and AUC-PR metrics for the Hyperplane dataset using a windows of 1k

Strategy	ADWIN		DDM		STEPD	
	AUC	AUC-PR	AUC	AUC-PR	AUC	AUC-PR
Best	0.923 ± 0.010	0.918 ± 0.014	0.931 ± 0.009	0.919 ± 0.018	0.928 ± 0.014	0.914 ± 0.015
Random	0.926 ± 0.012	0.919 ± 0.015	0.933 ± 0.015	0.922 ± 0.018	0.923 ± 0.014	0.910 ± 0.016
Wheel	$\mathbf{0.939 \pm 0.009}$	0.926 ± 0.011	$\mathbf{0.941 \pm 0.010}$	0.928 ± 0.016	0.931 ± 0.012	0.920 ± 0.014

The best strategy is highlighted in **bold**, provided that it is significantly better than the others according to Nemenyi post hoc test

Table 3 Comparison of the different replacement strategies for the three drift detection algorithms (AWIN, DDM and STEPD) using the AUC and AUC-PR metrics for the ISCX dataset using a windows of 1k

Strategy	ADWIN		DDM		STEPD	
	AUC	AUC-PR	AUC	AUC-PR	AUC	AUC-PR
Best	0.851 ± 0.040	0.796 ± 0.033	0.868 ± 0.004	0.797 ± 0.013	0.788 ± 0.010	0.734 ± 0.011
Random	0.872 ± 0.005	0.813 ± 0.011	0.872 ± 0.026	0.814 ± 0.030	0.794 ± 0.023	0.743 ± 0.029
Wheel	0.874 ± 0.012	$\mathbf{0.827 \pm 0.013}$	$\mathbf{0.892 \pm 0.028}$	0.822 ± 0.039	0.797 ± 0.011	0.746 ± 0.007

The best strategy is highlighted in **bold**, provided that it is significantly better than the others according to Nemenyi post hoc test

Table 4 Comparison of the different replacement strategies for the three drift detection algorithms (AWIN, DDM and STEPD) using the AUC and AUC-PR metrics for the ISCX dataset using a windows of 2k

Strategy	ADWIN		DDM		STEPD	
	AUC	AUC-PR	AUC	AUC-PR	AUC	AUC-PR
Best	0.879 ± 0.008	0.817 ± 0.014	0.855 ± 0.010	0.765 ± 0.016	0.783 ± 0.023	0.694 ± 0.030
Random	0.876 ± 0.006	0.818 ± 0.010	0.853 ± 0.024	0.751 ± 0.039	0.797 ± 0.012	0.730 ± 0.014
Wheel	0.876 ± 0.006	$\mathbf{0.821 \pm 0.006}$	$\mathbf{0.867 \pm 0.009}$	$\mathbf{0.781 \pm 0.016}$	0.808 ± 0.020	0.740 ± 0.024

The best strategy is highlighted in **bold**, provided that it is significantly better than the others according to Nemenyi post hoc test

example d_i in the sample, the *error rate* p_i is computed on the basis of this model, and standard deviation $s_i = \sqrt{p_i \cdot (1 - p_i)/i}$. Since in a stationary regime the error rate of C is assumed to decrease when having more examples, a significant increase in the error rate suggests a drift in the class distribution. This check is performed by DDM by comparing $p_i + s_i$ with $p_{min} + 3 \cdot s_{min}$, where p_{min} and s_{min} are the minimum values of p_i and s_i, respectively, in the current sample. This technique is expected to be good at detecting abrupt changes and (not too slow) gradual changes.

– *ADWIN* (“ADaptative WINDdowing”) (Bifet and Gavalda 2007) maintains a sliding window W with the most recent tuples and compares the class distributions in two sub-windows of W. When the difference of the average value of the two sub-windows is greater than a given threshold, then the older sub-window is dropped and a change in the distribution of examples is detected. The dimension of the windows is increased dynamically as long as no changes are found, or decreased when a change has been detected. A confidence parameter δ is used to control the false-positive rate: if the expected value of the distribution remains constant within W, the probability that ADWIN shrinks the window at this step is at most δ.

For all of the above drift-detection models, the standard setting of the parameters have been used in the prototypal implementation of our framework and in the experimentation [see (Bifet and Gavalda 2007) for details on the specific values of the parameters]. In particular, for STEPD we fixed $\alpha_d = 0.03$ and $\alpha_w = 0.08$, as suggested in Nishida and Yamauchi (2007), and used windows of 20 instances (as in Nishida and Yamauchi 2007). The confidence parameter δ of ADWIN was always set to 0.002.

5.4 Classifier induction

As explained before, this module is responsible for inducing a collection of base models from given training tuples, and it is called when building an initial version of the ensemble and whenever a concept drift is detected on a window D_i (to extract a collection of new base models that capture well the emerging concept). The module is built on the well-known Massive Online Analysis (MOA) toolbox,[4] which allows for efficiently applying different classifier-induction methods to streaming data. The current implementation of the module reuses functionalities for building decision trees, Bayesian models, logistic regression models, as well as k-NN classifiers. In particular it includes the implementation of the following algorithms (which were all used in the experimentation discussed in Sect. 6): *J48* (decision trees), *JRIP* (Ripper rule-learning algorithm), *NBTree* (Naive Bayes trees), *Naive Bayes* (Bayesian net, with class-conditionally independent attributes), *1R* classifier, *Logistic Model Tree*, *Logistic Regression*, *Decision Stumps* and *1BK* (k-nearest neighbor algorithm).

6 Experimental section

This section discusses the results of different suites of experiments that we performed to compare the performance of our approach (referred to as $E2SC4ID$ from now on) to those of several state-of-the-art competitors, and to study the effect of different window sizes and different strategies for both detecting the drifts and replacing the base classifiers. An artificial dataset and a real intrusion detection dataset were used to this aim, which are described in the next subsection together with the parameter setting and evaluation metrics that have been employed in the experimentation.

6.1 Datasets, parameters and metrics

Datasets The performances of both E2SC4ID and the competitors were evaluated on an artificial dataset and a real dataset of the cyber-security domain. The Hyperplane generator available in MOA, very popular as a benchmark for drift-detection algorithms (Hulten et al. 2001), was used to generate the artificial dataset. This generator produces data for a binary classification problem, taking a random hyperplane in a d-dimensional Euclidean space as decision boundary. The user can customize the number of attributes generated and the attributes involved in the drift (respectively 20 and 10 in our experiments), as well as the magnitude of changes and the percentage of noise to be added to the data. In our experiments, we fixed the parameter $sigmaPercentage$ to 10, the percentage of noise to 5% and the total number of tuples to $100,000$. Let us refer to the resulting artificial dataset as ***Hyperplane***, from now on.

As most of the popular intrusion datasets used in the past to test IDS systems (e.g., KDD, DARPA, NSL-KDD) do not adequately supply a realistic scenario (Tavallaee et al. 2010), we resorted to a more recent (and more realistic) dataset: the ISCX IDS dataset from the Information Security Centre of Excellence of the University of New Brunswick (Shiravi et al. 2012). Going more into detail, 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. More specifically, the dataset consists of 2,230,620 records, made of standard `pcap` (packet capture) files, one for each day and containing information on the network traffic of that day. As summarized in Table 1, different days contain different attack scenarios, including HTTP Denial of Service, DDos, Brute Force SSH and attempts of infiltrating the subnetwork from the inside. Therefore, this dataset is particularly apt to our aims, as it represents the situation in which new types of attacks emerge over the time.

To turn these traffic data into the tabular form required by our framework, we aggregated them at the level of traf-

[4] http://moa.cms.waikato.ac.nz/.

fic flows, by resorting to the popular `flowcalc tool`[5] used in many previous intrusion detection works (and more suitable for streaming data settings, than other more recent pcap-oriented data extraction tools). The resulting tuples represent different network flows, consisting of both categorical attributes (e.g., the protocol used) and several aggregated measures (e.g., the total number of packets and of bytes exchanged in the flow, aggregate statistics on packets' sizes and inter-arrival times, the size of the first packets in the flow, and possibly statistics on web traffic's content). The dataset obtained this way is referred to as ***ISCX*** hereinafter.

Parameters' setting and Test of Significance The WEKA's[6] implementations of the following machine-learning algorithms were used to induce the base classifiers of our ensembles (i.e., to populate the list *learners* of algorithms in Fig. 3): J48, JRIP, NBTree, Naive Bayes, 1R classifier, logistic model trees, logistic regression, decision stumps and 1BK.

The maximum number *maxC* of base classifiers was kept fixed to 20. However, the initial version of the ensemble model (induced from the first chunk of data) was made to only consists of 10 base classifiers.

No tuning phase was conducted for the GP tool, which was used with the same parameters' setting as in the original paper (Folino et al. 2003): a crossover probability of 0.7, a mutation probability of 0.1, a maximum depth of 7, 120 individuals per population, and 500 generations.

All the experiments were performed on a Linux cluster with 16 Itanium 2 1.4 GHz nodes, each having 2 GBytes of main memory and connected through a high-performance Myrinet network. All the results shown in this section were obtained by averaging those obtained in 30 different trials.

In order to enable for statistically significant comparative analyses, we resorted to a combined application of the Friedman test and Nemenyi test. Basically, the Friedman test (Demsar 2006) is a non-parametric statistical test that allows for assessing whether there are differences in the results of a collection of methods. The null hypothesis of this test is that all the populations of results (corresponding each to the application of one of the methods analyzed) have the same median value. Once the Friedman test rejects the null hypothesis, a post hoc test is required in order to make pairwise comparisons, and detect the couples of methods that are significantly different. In our experimental analysis, the Friedman test was executed for all the measures (columns) of all the tables in Sects. 6.2 and 6.3 (i.e., Tables 2, 3, 4, 5 and 6), with a critical value obtained from a Chi-square distribution with two degrees of freedom and a significance level of 5%. The p value was corrected for multiple hypotheses by using the Holm methodology (García and Herrera 2009) and, to verify the differences between each couple of methods, as post hoc test, we adopted the Nemenyi test (Demsar 2006).

Evaluation metrics Two different quality metrics have been used to evaluate the discovered classification models: (i) *AUC* (area under the ROC curve), which quantifies the area under the curve relating the false-positive rate and the true-positive rate of the classification model under evaluation; and (ii) *AUC-PR* (area under the curve of precision–recall), which measures the area under the curve relating the precision and recall scores obtained by a classification model.

These measures were computed according to a binary classification setting, featuring only two classes of instances: negative instances (representing normal behaviors) and positive instances (representing anomalous/attack behaviors).

Notably, the latter metrics is often regarded as a more suitable metrics for IDS settings featuring imbalanced classes, owing to the fact that it does not depend on the true-positive rate—and hence it is less biased in scenarios where the normal connections are far more than the malicious ones. However, AUC-PR is more benevolent towards classification models that raise many false alarms. In the light of these considerations, we eventually decided to adopt both metrics in our empirical analysis.

6.2 Studying the effect of different replacement and drift-detection strategies

A first suite of experiments were carried out to evaluate the effectiveness of our approach in a number of different operating modes, corresponding to different instantiations of the classifier-replacement strategy and of the drift-detection model (*DM*). This allows us to study the impact, on the quality of the classification model returned, of these two important (parametric) aspects of the proposed framework.

To this end, we considered 9 different configurations of algorithm $E2SC4ID$, resulting from combining each of the three replacement strategies introduced in Sect. 5.2 (namely, *Wheel*, *Random* and *Best*), with one of the three drift-detection models (namely, ADWIN, DDM and STEPD) described in Sect. 5.3. Furthermore, we considered two different values, namely 1k and 2k, for the size of the chunking window (i.e., the number of instances per data chunk) –these two values correspond to setting $n = 1000$ and $n = 2000$, respectively, in the algorithm $E2SC4ID$ of Fig. 3. For a more complete study of the window size's effect, we refer the reader to Sect. 6.4.

The AUC and the AUC-PR scores obtained by our approach on the Hyperplane dataset are shown in Table 2 , for each of the above-mentioned combination of drift-detection and replacement options. As the performance of our approach on this dataset did not differ significantly when varying the size of the windows (i.e., chunks), we prefer to report these scores only for the case of 1k-sized windows. By contrast,

[5] http://mutrics.iitis.pl/flowcalc.

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

Table 5 Comparison with the competitors on dataset ***Hyperplane***: AUC and AUC-PR results obtained when using each of the drift-detection strategies (ADWIN, DDM and STEPD), and a fixed window size of 1k

Algorithm	ADWIN		DDM		STEPD	
	AUC	AUC-PR	AUC	AUC-PR	AUC	AUC-PR
RandHTree	0.570	0.556	0.632	0.610	0.621	0.610
HoeffTree	0.641	0.622	0.641	0.622	0.683	0.663
Meta-TAC	0.640	0.621	0.653	0.631	0.680	0.651
RHTBoost	0.725	0.709	0.856	0.811	0.858	0.849
HTBoost	0.859	0.807	0.875	0.802	**0.930**	**0.917**
E2SC4ID	**0.939**	**0.926**	**0.941**	**0.928**	**0.931**	**0.920**

The algorithm(s) performing significantly better than all the other ones (according to Nemenyi post hoc test) are highlighted in **bold**

Table 6 Comparison with the competitors on dataset ***ISCX***: AUC and AUC-PR results obtained when using each of the drift-detection strategies (ADWIN, DDM and STEPD), and a fixed window size of 1k

Algorithm	ADWIN		DDM		STEPD	
	AUC	AUC-PR	AUC	AUC-PR	AUC	AUC-PR
RandHTree	0.729	0.740	0.822	0.815	0.734	0.727
HoeffTree	0.771	0.781	0.813	0.810	0.822	0.824
Meta-TAC	0.783	0.798	0.819	**0.835**	0.824	0.826
RHTBoost	0.860	**0.825**	**0.891**	**0.844**	**0.928**	**0.885**
HTBoost	0.864	0.798	0.869	0.758	0.890	0.783
E2SC4ID	**0.874**	**0.827**	**0.892**	0.822	0.797	0.746

The algorithm(s) performing significantly better than all the other ones (according to Nemenyi post hoc test) are highlighted in **bold**

the AUC and AUC-PR scores obtained on the ISCX dataset are reported for a window size of 1k and of 2k, in Tables 3 and 4, respectively.

For all the experiments in this subsection, when the Friedman test detects differences, we highlight the best strategy in bold, on the basis of the results of the Nemenyi post hoc test. Obviously, if the Friedman test fails, no algorithm is highlighted, as the differences are not significant.
From analyzing the tables, it is evident that the Wheel replacement strategy, when combined with the ADWIN and DDM drift detection methods, outperforms significantly (or in a few cases is not significantly different) the other ones in terms of both AUC and AUC-PR, on both the artificial dataset and the real dataset. When employing STEPD, on the contrary, no significant differences can be observed among all the methods.

The degradation of the performance due to using a window of 2k does not appear particularly relevant here. However, we pinpoint that the effect of increasing the window size will be analyzed in more detail in Sect. 6.4.

6.3 Comparison with state-of-the-art algorithms

In accordance with the results of the previous subsection, the Wheel replacement strategy appears to be the best choice. Thus, let us focus on this particular configuration of our approach, in our empirical comparison with existing (competitor) approaches to online classification.

To this end, we considered five state-of-art algorithms (three of them ensemble-based), suitably empowered with the same drift-detection models as in our framework, for the sake of fairness:

- *HoeffTree*: The popular algorithm *Hoeffding Tree* in Hulten et al. (2001), devised to incrementally induce a decision tree in an efficient (and yet sound) way by only using a minimal number (chosen according to the Hoeffding bound) of training examples to decide each split (in the growing phase).
- *RandHTree*: A randomized version of the Hoeffding Tree, defined in Bifet et al. (2012), where only a small subset of the attributes are considered for splitting the training data associated with a leaf the decision tree, during the growing phase. More precisely, from the m original data attributes, only $\lfloor\sqrt{m}\rfloor$ attributes are selected at random, and used to decide the next attribute to split on.
- *HTBoost*: The boosting scheme *OzaBoost* (Oza 2001), instantiated with an Hoeffding-Tree base learner. Basically, the meta-learning scheme *OzaBoost* can be regarded as an adaptation of the well-known AdaBoost algorithm (Schapire 1995) to online classification settings that employs a Poisson distribution to decide whether an example will be used or not for training the chain of classifiers.
- *RHTBoost*: The same boosting scheme *OzaBoost* (Oza 2001) as in the previous case, using the above-mentioned

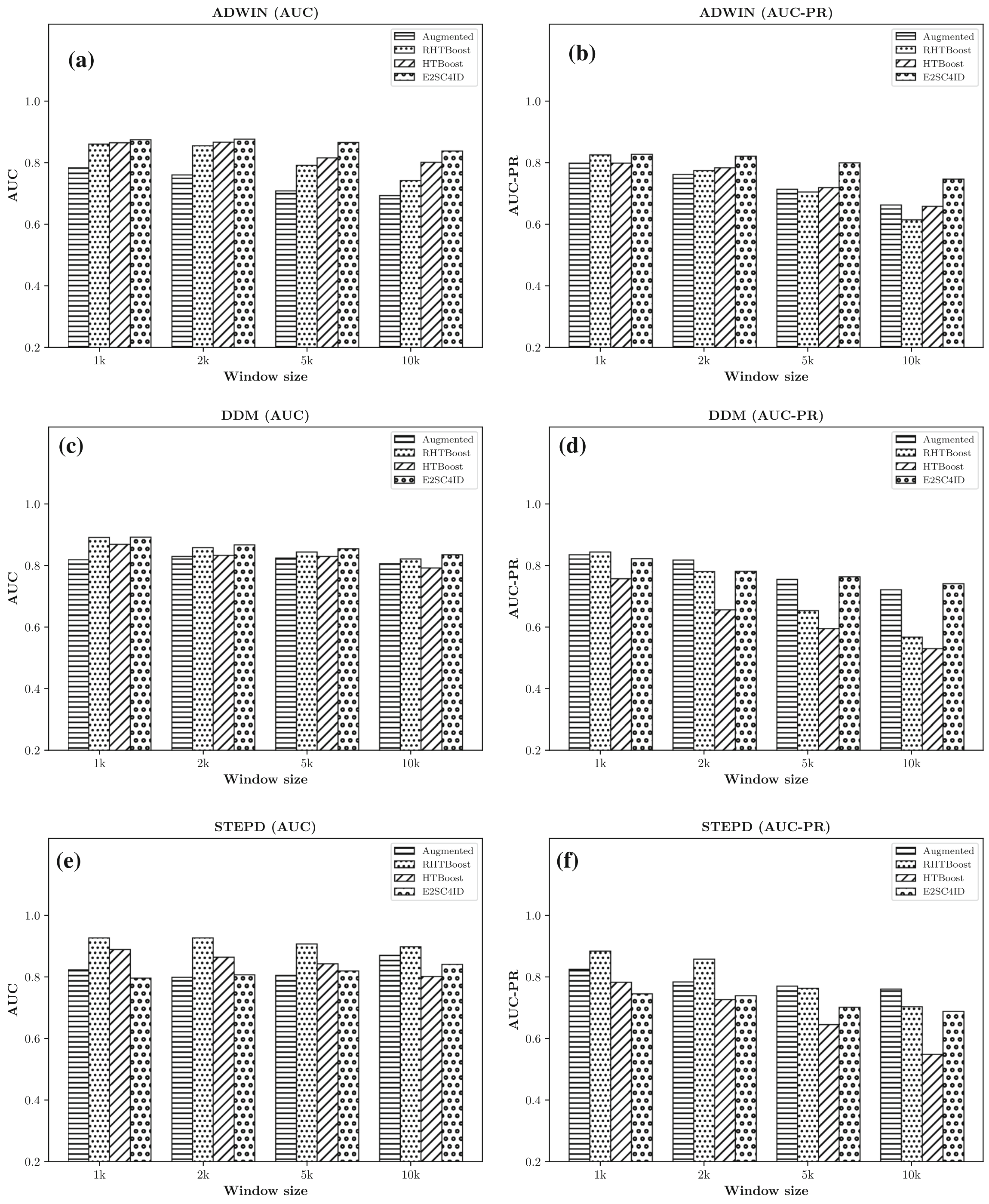

Fig. 6 Comparison of the meta-learning-based competitors with E2SC4ID on dataset ISCX for different window sizes (namely, 1k, 2k, 5k, 10k): AUC (left) and AUC-PR (right) results obtained when using different drift-detection strategies, namely ADWIN (top), DDM (middle), and STEPD (bottom)

Randomized Hoeffding-Tree algorithm of (Bifet et al. 2012) as base learner.

- *Meta-TAC*: The meta-learning (more precisely, wrapper) scheme *Temporally Augmented classifier* in Žliobaitė et al. (2015), combined with an Hoeffding-Tree base learner. Basically, before applying the base learner, every example tuple d_i in the input stream is augmented with k additional attributes, representing the class labels of the k most recent examples appeared in the stream before d_i. The (meta) training instances obtained this way are then passed to the base learner (which will compute an Hoeffding-Tree classifier, in our case). This allows for capturing dependencies that link the input features to the past class labels, or the labels at different times to one another—via a sort of k-order Markov chain. The default setting $k = 1$ was used in the tests presented next.

All the experiments described in the following were carried out by only using a window size of 1k, but combining each algorithm with each of the three drift-detection strategies (i.e., ADWIN, DDM and STEPD) considered. For testing the competitor algorithms, we leveraged the respective implementations available in the MOA library.

The results that our algorithm (with the Wheel replacement strategy) and the competitors obtained on the Hyperplane and ISCX datasets (when using a window of 1k) are shown in Tables 5 and 6, respectively. For each algorithm, three different configurations are reported in this comparison, which correspond to the three different drift-detection strategies considered (i.e., ADWIN, DDM and STEPD).
For each column, i.e., for each combination of quality metrics and drift detection strategy, when the Friedman test detects differences, we have adopted the following strategy: the algorithms were ranked on the basis of the results of the Nemenyi post hoc test, and the algorithm (or group of algorithms), which is significantly better than all the other competitors, is highlighted in bold. In the remaining cases (with no significant differences found by the Friedman test), no algorithm has been highlighted.
As far as concern the artificial dataset Hyperplane, our algorithm is significantly better than all the competitors when used together with the ADWIN or the DDM drift-detection strategy, whereas its results are comparable to *HTBoost* when adopting the STEPD strategy. The tree-based algorithms and *Meta-TAC* appear to perform considerably worse than the other ensemble-based algorithms.

A similar trend emerges from analyzing the results on the real dataset ISCX, even if the degradation in performance of the tree-based algorithms is less evident. When used together with the ADWIN strategy, E2SC4ID obtains the best results in terms of both AUC and AUC-PR (even if for the latter metrics, it is comparable to *RHTBoost*). When detecting the drifts with DDM, the best AUC is obtained by our approach (together with *RHTBoost*), while *RHTBoost* and *Meta-TAC* outperform all the others in terms of AUC-PR, but our approach is only slightly worse. Finally, the *RHTBoost* algorithm performs significantly better than all the other ones when used together with the STEPD strategy.

6.4 The effect of different sizes of the window

This subsection aims to analyze the behavior of the ensemble/meta-learning-based algorithms (E2SC4ID and the competitors *Meta-TAC*, *HTBoost*, *RHTBoost*) when varying the size of the windows/chunks (1k, 2k, 5k and 10k).

Figure 6 reports the results obtained on the ISCX dataset, for the different window sizes, when combining each of the compared algorithms with the ADWIN (subfigures a and b), DDM (subfigures c and d) and STEPD (subfigures e and f) drift-detection strategies.

By looking at this figure, a common trend can be observed for all the analyzed algorithms: when the size of the window is increased, both the AUC and AUC-PR score degrade, as it was expected—indeed, when using a larger window, less drifts will be detected or, in any case, the classification models are adapted to the drift in a more delayed way. However, the *RandHTree* algorithm appears less affected by this degradation, especially in terms of AUC, probably because it works with the most recent examples in the input stream and, therefore, it is less prone to the problem of not detecting the drift in a timely manner.

Interestingly, E2SC4ID, for all the drift-detection techniques, does not degrade too much also when the size of the windows arrives to 10k. However, while when using the AWIN and DDM strategies E2SC4ID outperforms the other competitors, when it is combined with the STEPD strategy it is surpassed by both *RHTBoost* and *HTBoost*, even if the differences are less relevant in terms of the AUC-PR metric.

7 Conclusion and future work

An ensemble-based framework for the online classification of intrusion detection data has been proposed here, which relies on using an ensemble classification model where the combiner function is defined in terms of non-trainable aggregation functions, and discovered in a data-driven way through a Genetic Programming (GP) method. This approach is supported by a system architecture, which integrating different kinds of functionalities, ranging from drift-detection mechanisms, to the induction/replacement of base models, to the efficient GP-based computation of the combiner function.

A suite of experiments, conducted on artificial and real datasets, permitted us to compare our approach with several competitors, and to study the effect of different window sizes and different strategies for both detecting the drifts and replacing the base classifiers. The result of these experiments confirmed that the proposed framework is capable of dealing with non-stationary data streams in an effective manner, and hence constitutes a valuable solution for real-life intrusion detection scenarios.

Future work In order to empower the framework with the ability to equip the ensemble models with more expressive (and hopefully more effective) combiners, we plan to investigate on complementing the simple range of non-trainable aggregation functions currently used in it with classic fuzzy-logics functions (such as t-norms) and/or *generalized mixture functions* (Costa et al. 2018). Moreover, we will try to extend the framework in a way that "context-aware" combination schemes can be derived, which can adapt to the features of the tuple that is being classified, in the spirit of Dynamic Ensemble Selection/Weighting (Cruz et al. 2018) approaches. These two lines of extension clearly entail a careful study and design, seeing as they might strongly impact on computational aspects.

A further direction of future work concerns adapting the pre-filtering of the base models (preliminary to the discovery of a new combiner) in a way that the degree of diversity among the base models is taken into account. Indeed, if having a high level of diversity is a desideratum for an ensemble classifier in general, it may become a key feature for making it robust enough towards the dynamically changing nature of intrusion detection scenarios.

Compliance with ethical standards

Conflict of interest The authors declare that they have no conflict of interest.

Ethical approval This article does not contain any studies with human participants or animals performed by any of the authors.

References

Aburomman AA, Reaz MBI (2017) A novel weighted support vector machines multiclass classifier based on differential evolution for intrusion detection systems. Inf Sci 414:225–246

Acosta-Mendoza N, Morales-Reyes A, Escalante HJ, Gago-Alonso A (2014) Learning to assemble classifiers via genetic programming. IJPRAI 28(7)

Bifet A, Gavalda R (2007) Learning from time-changing data with adaptive windowing. In: SDM, vol 7. SIAM

Bifet A, Frank E, Holmes G, Pfahringer B (2012) Ensembles of restricted hoeffding trees. ACM Trans Intell Syst Technol (TIST) 3(2):1–20

Borji A (2007) Combining heterogeneous classifiers for network intrusion detection. In: Cervesato I (ed) Advances in computer science—ASIAN 2007. Computer and network security, vol 4846. Lecture notes in computer science. Springer, Berlin, pp 254–260

Buczak AL, Guven E (2016) A survey of data mining and machine learning methods for cyber security intrusion detection. IEEE Commun Surv Tutor 18(2):1153–1176

CERT Australia (2012) Cyber crime and security survey report. Technical report, 2012

Costa VS, Farias ADS, Bedregal B, Santiago RHN, de P Canuto AM, Magaly de A (2018) Combining multiple algorithms in classifier ensembles using generalized mixture functions. Neurocomputing 313:402–414

Cruz RMO, Sabourin R, Cavalcanti GDC (2018) Dynamic classifier selection: recent advances and perspectives. Inf Fusion 41:195–216

de Oliveira DF, Canuto AMP, de Souto MCP (2009) Use of multi-objective genetic algorithms to investigate the diversity/accuracy dilemma in heterogeneous ensembles. In: International joint conference on neural networks. IEEE, pp 2339–2346

De Stefano C, Folino G, Fontanella F, Scotto di Freca A (2014) Using bayesian networks for selecting classifiers in GP ensembles. Inf Sci 258:200–216

Demsar J (2006) Statistical comparisons of classifiers over multiple data sets. J Mach Learn Res 7:1–30

Folino G, Sabatino P (2016) Ensemble based collaborative and distributed intrusion detection systems: a survey. J Netw Comput Appl 66(C):1–16

Folino G, Pizzuti C, Spezzano G (2003) A scalable cellular implementation of parallel genetic programming. IEEE Trans Evol Comput 7(1):37–53

Folino G, Pizzuti C, Spezzano G (2008) Training distributed GP ensemble with a selective algorithm based on clustering and pruning for pattern classification. IEEE Trans Evol Comput 12(4):458–468

Folino G, Pisani FS, Sabatino P (2016a) A distributed intrusion detection framework based on evolved specialized ensembles of classifiers. In: Applications of evolutionary computation—19th European conference, EvoApplications 2016, Porto, Portugal, 30 March–1 April 2016, Proceedings, Part I, pp 315–331

Folino G, Pisani FS, Sabatino P (2016b) An incremental ensemble evolved by using genetic programming to efficiently detect drifts in cyber security datasets. In: Genetic and evolutionary computation conference, GECCO 2016, Denver, CO, USA, 20–24 July 2016, Companion material proceedings, pp 1103–1110

Folino G, Pisani FS, Pontieri L (2019) A cybersecurity framework for classifying non stationary data streams exploiting genetic programming and ensemble learning. In: Numerical computations: theory and algorithms—3rd international conference, NUMTA 2019, Crotone, Italy, 15–21 June 2019, Revised Selected Papers, Part I, volume 11973 of Lecture notes in computer science, pp 269–277

Gama J, Medas P, Castillo G, Rodrigues P (2004) Learning with drift detection. In: SBIA Brazilian symposium on artificial intelligence. Springer, pp 286–295

Gao X, Shan C, Hu C, Niu Z, Liu Z (2019) An adaptive ensemble machine learning model for intrusion detection. IEEE Access 7:82512–82521

García S, Herrera F (2009) An extension on "statistical comparisons of classifiers over multiple data sets" for all pairwise comparisons. J Mach Learn Res 9:2677–2694

Gonçalves PM Jr, de Carvalho Santos SGT, de Barros RSM, De Lima Vieira DC (2014) A comparative study on concept drift detectors. Expert Syst Appl 41(18):8144–8156

Hulten G, Spencer L, Domingos P (2001) Mining time-changing data streams. In: Proceedings of the seventh ACM SIGKDD international conference on knowledge discovery and data mining. ACM, pp 97–106

Koza JR (1992) Genetic programming: on the programming of computers by means of natural selection. MIT Press, Cambridge

Kumar G (2020) An improved ensemble approach for effective intrusion detection. J Supercomput 76(1):275–291

Kuncheva L (2004) Combining pattern classifiers: methods and algorithms. Wiley-Interscience, New York

Nishida K, Yamauchi K (2007) Detecting concept drift using statistical testing. In: Discovery science. Springer, pp 264–269

Oza N (2001) Online bagging and boosting. Proc Artif Intell Stat 2001:105–112

Perdisci R, Ariu D, Fogla P, Giacinto G, Lee W (2009) Mcpad: A multiple classifier system for accurate payload-based anomaly detection. Comput Netw 53(6):864–881 (Traffic classification and its applications to modern networks)

Schapire RE (1990) The strength of weak learnability. Mach Learn 5(2):197–227

Schapire RE (1995) Boosting a weak learning by majority. Inf Comput 121(2):256–285

Shiravi A, Shiravi H, Tavallaee M, Ghorbani AA (2012) Toward developing a systematic approach to generate benchmark datasets for intrusion detection. Comput Secur 31(3):357–374

Sindhu SSS, Geetha S, Kannan A (2012) Decision tree based light weight intrusion detection using a wrapper approach. Expert Syst Appl 39(1):129–141

Sylvester J, Chawla NV (2005) Evolutionary ensembles: combining learning agents using genetic algorithms. In: AAAI workshop on multiagent learning, pp 46–51

Tavallaee M, Stakhanova N, Ghorbani AA (2010) Toward credible evaluation of anomaly-based intrusion-detection methods. IEEE Trans Syst Man Cybern Part C Appl Rev 40(5):516–524

Žliobaitė I, Bifet A, Read J, Pfahringer B, Holmes G (2015) Evaluation methods and decision theory for classification of streaming data with temporal dependence. Mach Learn 98(3):455–482