J Comput Virol Hack Tech (2013) 9:205–214
DOI 10.1007/s11416-013-0188-1

CORRESPONDENCE

Building a practical and reliable classifier for malware detection

Cristina Vatamanu · Dragoş Gavriluţ · Răzvan-Mihai Benchea

Received: 30 April 2013 / Accepted: 28 May 2013 / Published online: 15 June 2013

Abstract Having a machine learning algorithm that can correctly classify malicious software has become a necessity as old methods of detection based on hashes and hand written heuristics tend to fail when dealing with the intensive flow of new malware. However, in order to be practical, the machine learning classifiers must also have a reasonable training time and a very small amount, preferably zero, of false positives. There were a few authors who addressed both these issues in their papers but creating such a model is more difficult when more than 3 million files are involved/needed in the training. We mapped a zero false positive perceptron in a new space, applied a feature selection algorithm and used the resulted model in an ensemble, voting or a rule based clustering system we've managed to achieve a detection rate around 99 % and 0.07 % false positives while keeping the training time suitable for large data sets.

Keywords Malware detection · One side class algorithm · False positives · Machine learning · Large data sets

C. Vatamanu
Gheorghe Asachi Univerisity, Iasi, Romania

C. Vatamanu (✉) · D. Gavriluţ · R.-M. Benchea
Bitdefender, 37 Sfantul Lazar Street, Solomons Building, Iasi, Romania
e-mail: cvatamanu@bitdefender.com

D. Gavriluţ · R.-M. Benchea
Alexandru Ioan Cuza Univerisity, Iasi, Romania
e-mail: dgavrilut@bitdefender.com

R.-M. Benchea
e-mail: rbenchea@bitdefender.com

1 Introduction

Due to the exponential increase in the number of malware pieces in the last years [3], a good solution is to detect malicious software using a machine learning algorithm. Standard detection techniques (ex: using hashes to detect parts of a file) are no longer feasible due to the rapid changes that appear in important malware families. Even though there are many machine learning algorithms that can be used to classify malware, most of them are not practical when used on very large data sets (millions of records and hundred of features) because of their training time complexity. Training time is important as different versions of the same malware can appear at intervals of a couple of hours rendering slow training algorithms inefficient. Another important criteria for choosing the best algorithm for malware detection is the ability to be modified in order to achieve a zero (or very close to zero) false positives.

We chose the perceptron algorithm because of its ability to parallelize (a necessary thing when dealing with large data sets) and its relatively low time complexity. Another useful feature of the perceptron algorithm is its ability to be fine-tuned in case of false alarms [10]. Despite all its advantages, the perceptron has one significant shortcoming. The perceptron, unlike other machine learning algorithms (i.e. neural networks) cannot create features, leaving this task exclusively to the programmer. Theoretically, the perceptron can provide the same classification as any other machine learning algorithm with the right features.

Our work is a follow-up to the research conducted in paper [9]. The authors present a modified perceptron algorithm that can achieve a low number of false positives with a medium detection rate. In order to increase the detection the authors applied the kernel trick method [1] which increased the number of features exponentially. Despite a higher detection rate, this method yields a higher time complexity rendering the algorithm unsuited for very large data sets. The algorithm presented in the previous paper (named one side class perceptron) was further optimized in order to increase its training speed ([10]). Another part that further needs optimization is the way features are combined in order to increase the detection rate while keeping the algorithm practical in terms of time complexity. This issue along with other methods to increase the one side class algorithm's detection is addressed in this paper.

In our previous work [10] we've created a new algorithm, name one side class 3 (OSC3) by modifying the perceptron algorithm to produce a medium detection rate in a practical amount of time while training for zero false positives. Despite our initial recommendation of using this algorithm in conjunction with other detection methods for a greater detection rate, we now believe that this algorithm has even better chances to provide better malware detection by further modifying it. This paper focuses on presenting the proper methods to achieve this goal. We concentrated on the features used by the algorithm. By using the kernel trick [1] that is basically a combination of all features between every record with all the rest others we created a new set of features that gave the algorithm a new perspective over the data. Several functions are used for the combination and their advantages are also analyzed. Given the fact that this method increases the complexity in an exponent manner, we concentrated part of our work on selecting the most relevant features. By using different features with the OSC3 algorithm we achieved different classifiers, each of them with zero false positives and correctly classifying a different part of the elements from the other class. We've analyzed different to combine the classifiers to achieve a better detection rate. By applying these optimizations we've managed to increase the detection rate from 53 to 99 % while keeping the false positive level very close to zero.

2 Related work

The perceptron algorithm introduced by Frank Rosenblatt in 1958 [17] marked the beginning of machine learning and gave a new direction to the classification problem. In its basic form, it mimics the way a neuron cell works by applying a signum function over a linear combination of the inputs and weights of the synapses. Even though its use has decreased over time, being replaced by other machine learning algorithms, the perceptron algorithm remains a very useful tool for classifying large data sets due to its simplicity and its ability to be parallelized.

In order to provide parallelization to the algorithm, the authors in [16] modified it so it would adjust the separating plan after having observed the whole training set. Because the order of the elements no longer affects the final result, the algorithm can be ran in a parallel manner by using the map-reduce paradigm [6]. The advantages of running the perceptron on a distributed system were studied by the authors in [4] and [20]. Similar work was carried out by the authors in [14] by using several processing cores with shared memory to perform a stochastic gradient descent and a combination of results. Even though in its simple form the map-reduce paradigm suggests to mix the results after each parallel training unit has finished processing its shard of the data, the authors in [16] showed that a straight-forward parameter mixing is not appropriate for the perceptron algorithm.

One major setback of the perceptron is the fact that it needs the right features to achieve a good classification. The individual creation of these features means a considerable amount of work for a programmer and it also poses the risk of creating irrelevant features. This is the reason why many prefer to create these features automatically and keep only a part of them based on a feature selection algorithm. This method however can skip a lot of the features a programmer can easily spot. The algorithm we designed combines the control of the manual features with the increased detection rate of the automatic features. Other methods of creating and selecting features were also studied. The authors in [15] provide a way to optimize a compiler by continuously creating features with a genetic algorithm and keeping only those that provide good results based on a machine learning tool. Fuzzy pattern recognition was used by authors in [23] in order to create features from extracted windows API calls. The method was based on dynamic analysis meaning that an antivirus had to run the malware to detect it. The method is also not feasible when dealing with a high number of malware samples. This is probably why the tests were performed on a very small set of files. The authors from [11] carried on this research and modified the algorithm to use static features. The number of tested files is still very small(120 benign and 100 malicious) which can lead to inaccurate results. Other authors, including those in [5], addressed the issue of using groups of API calls as features. Both malware and clean files are run in a virtual machine and features are created based on the API calls. Next, a feature selection algorithm is run in order to select the best features. The classification tool used is a support vector machine. Based on the idea that the strings and the API calls provide important semantics and can show the executable intent, the authors in [21] have created features that incorporate just that. The features were further filtered by applying a Max-Relevance algorithm. By extracting dynamic features (mainly API calls) and applying the Information Gain to filter them before sending to a voted perceptron, the authors in [2] achieved a 99 % detection rate and 1 % false positives.

However the authors did not specify the number of malware and clean files tested, so we cannot know how much that 1 % really means. In order to train a classifier for a high detection rate in malware detection and a low number of false positives, the authors in [12] provide a new mechanism of feature selection. The idea is to combine more features and select only the ones that are representative for the subclass the author needs to classify and add them to a subset. For malware detection, there will be two feature subsets, one to detect clean files, and the other the malicious files. After removing redundant features and combining the two subsets a new set of features will be created. This set will go through the same process as the original one until a final set is reached.

Apart from the effort invested into the optimization of the training speed and the accuracy of the algorithm, special attention should also be given to the issue of false positives. In special cases, such as the anti malware industry, a false alarm can do greater harm than not detecting a malware. For instance, not detecting a sample is not as harmful as deleting a system file or an important document. As far as we know, little research has been done with respect to limiting the number of false positives for malware detection systems. The authors in [19] studied many machine learning algorithms to detect spam messages while trying to limit the number of false positives. They came to the conclusion that the perceptron cannot be altered to limit the number of false alarms. Other papers came however to another conclusion. A zero false positive perceptron was studied by authors in [9] and was further optimized by us in a previous paper [10]. In its basic form, the algorithm is made up of a standard perceptron algorithm and another part that aims to eliminate false positives. In each iteration, after the perceptron adjusts the separating plan, another algorithm will adapt it to correctly classify all the elements from one class. Thus, at the end of each iteration all the elements from one class will be correctly classified. The authors in [22] are looking into a new strategy to limit the number of false positives and false negatives. The main idea is to do the training in two stages. The first stage is done on the whole training set and its aim is to discard the easy detectable samples (good and spam). The second one is done on the remaining samples and its purpose is to make a classifier able to detect the spam samples that resemble much with the clean emails. The issue of achieving a low positive classifier was also addressed by the authors in [13] by using a cost sensitive support vector machine. The cost sensitive classifier approach was also studied by the authors in [7] but their method is not specific to support vector machines. In order to increase accuracy, 53 several spam classifiers are combined in a voting manner. The authors conclude that the combination provides better results than the best classifier and in order to achieve the same performance only half of the classifiers are needed.

3 Discussion—improving the detection of perceptron-based algorithms

Over the past years malicious software has evolved to the point where new versions of the same malware family can appear every couple of hours. This is the main reason machine learning is considered a detection mechanism. Unfortunately this is not enough.

It is important that used algorithms have a short training time when dealing big malware streams and have a feasible model for the new malicious files.

Being able to have a quick response is not the only important point. In the AV industry, few false positives (close to 0) is a critical demand. The OSC-3 version [10] can achieve this requirement. Basically, the training process needs to be divided into two steps: a training step (where all the records are used) and a class minimize step (where only the clean files are used). At the end of the class minimize step all the clean files are correctly classified. After achieving these demands the next step is to improve the detection of these algorithms by following some steps:

1. Improve data separability using:
   (a) Kernel function.
   (b) Create new features.
   (c) Obtain new features using old ones.
2. Combine algorithms:
   (a) Vote System.
   (b) Ensemble System.
   (c) Clustering hybrid method.

3.1 Improve data separability

3.1.1 Kernel functions

While in theory this method gives good results, in practice it is not feasible. The number of operations in case of kernel functions is $|R| \times |R| \times m$ ('R' is the set of records, 'm' is the number of features) while in case of a simple perceptron it is $|R| \times m$. If we consider a data-base of 22 million records,the time required to complete the training process in the second case would be 22,000,000 times bigger. Also, using a Gram Matrix in this case is not feasible. The size of this matrix, for an end-point product, would reflect on the the client computer and an AV product can't afford to allocate this kind of storage on the computer of a average user.

3.1.2 Create new features

Over the past few years, the number of new malicious files that appear every day has increased significantly. Most malware creators change their old variant of malware until the new version will pass undetected by most AV products. This issue stresses the constant demand for new features.

Over the last four years we've manage to create over 20,000 features, of which some 60 % are Boolean type while the rest are values.

3.1.3 Obtain new features from the old ones

Since the most common method to obtain new versions of malware is to change some aspects of the initial file (add a new section, change the packer) a method to improve detection is to obtain new features from the old ones.

Two approaches were considered. The first one is to create new Boolean features from the old value-based ones. This can be achieved either by using one value-based at a time (e.g. from the feature 'file size' (a numeric value) it can be obtained the Boolean record if the file size is greater than a specific threshold) or by using multiple value-based features in a Boolean expression ($valueOf(feat_1) + valueOf(feat_2) > 50$). The downside of this method is that from a practical point of view it is not suitable. The time needed to gather information from a big data base is very large. For example for a data base with 32 million of records, the time needed to extract the information about the size of each file (using a 16-core computer) was 15 min. For 600 value-based features it will be required 600×15 min = 150 h = approximately 7 days.

Another way to obtain new features is use the old Boolean features and apply Boolean operators such as AND, OR, XOR over two or more features.

The next algorithm (Algorithm 1) illustrates a simple way of mapping every two features into a higher space.

To facilitate the writing of the following algorithms, we considered these abbreviations:

1. $Record \rightarrow R$
2. $Features \rightarrow F$
3. $Label \rightarrow L$

Algorithm 1 MapAllFeatures algorithm

```
1: function $MapAllFeatures(R)$
2:     $newFeatures \leftarrow \emptyset$;
3:     for $i = 1 \rightarrow |R.F|$ do
4:         for $j = i \rightarrow |R.F|$ do
5:             $newFeatures \leftarrow newFeatures \cup R.F_i\ |\overline{OP}|\ R.F_j$;
6:         end for
7:     end for
8:     $R.F \leftarrow newFeatures$;
9: end function
```

The logical operations (***OP***) considered were the above mentioned AND, OR and XOR. The main problem here is the large number of new features. Considering our data base of 11,863 features, after applying the above algorithm:

$$newFeatureCount \leftarrow \frac{n \times (n+1)}{2}$$

$$newFeatureCount \leftarrow \frac{11,863 \times (11,863+1)}{2} = 70,371,316\ features$$

One possible solution would be to sort out the new features based on a score (for example F2 score [18]) and select the most 'n' relevant ones. Using an OSC-based algorithm that guarantees a 0-false positive rate we can compare the detection rates for the three logic operations. From the 11,863 features, 200 were selected (based on F2 score) to create a new data-base: DB-F2-200. Mapping the chosen 200 features will result in 20,100 new features—a number that will allow the following test to be executed in an acceptable frame of time. Since the number of features was reduced to only 200, some records from different classes had the same feature combinations. Because this kind of records downgrade the accuracy of any linear classifier, those records were removed from the DB-F2-200 data-base (Table 1).

Table 1 Feature selection on large data base

Category	Initial database	DB-F2-200
Total records	22,100,23	13,932,320
Malware	3,116,937	2,486,210
Clean	18,983,294	11,446,110
Size	4,890,201,838	554,439,480
Features	11,863	200

Considering the following notations:

1. OSC-3-MAP-AND → OSC-3 Algorithm, using mapping technique with AND functions.
2. OSC-3-MAP-OR → OSC-3 Algorithm, using mapping technique with OR functions.
3. OSC-3-MAP-XOR → OSC-3 Algorithm, using mapping technique with XOR functions.
4. OSC-3 → OSC-3 Algorithm without any mapping technique.

The next figure (Fig. 1) illustrates the time needed for each of the four algorithms to create a model by using the DB-F2-200 data-base and the obtained detection rates. The first three algorithms were used to compare the detection results. The last one was added as reference. All the tests were made for 100 iterations, on the same computer [CPU Intel(R) Xeon(R) E5630 at 2.53 GHz (2 processors), 12GB RAM, Windows Server 2008 R2 Enterprise 64-bit], using 16 threads for parallelization.

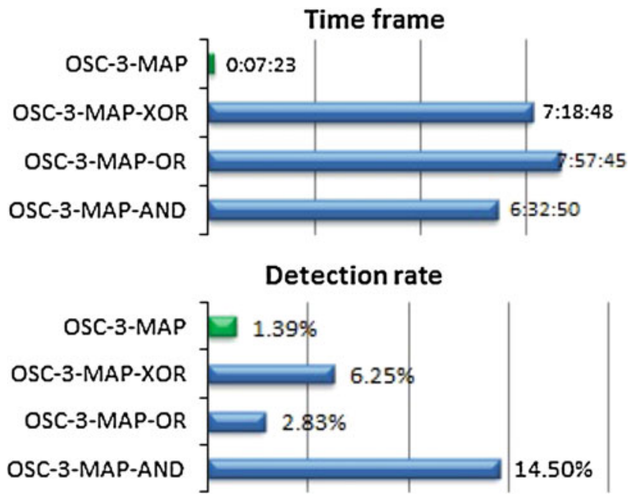

Fig. 1 Time needed by OSC-3-MAP-AND, OSC-3-MAP-OR, OSC-3-MAP-XOR and OSC-3 algorithm to complete 100 iterations and resulted detection rates

As shown, this mapping features method gave, for the AND operation, a 10 % improvement for the detection rate from the original algorithm. Even if the time needed to create a model, using this method, increases exponentially, the benefit added by the boost in the detection rate is good enough to consider it in practice. In addition, this paper also offers some steps that help improve the time required by the creation of a model, while preserving a similar detection rate for the mapping algorithm.

Another approach would be to limit the feature set size to the one of the original feature set (only a small subset of features that can be used for mapping) while maintaining the improvement of the detection rate. The first step is to sort the initial set of features by a specific score. After this, the first k features are selected and used to generate new features in a similar way to the prior algorithm. These new features are added to the original set (Algorithm 2).

Algorithm 2 MapAllFeatures-version2 algorithm

```
1: function $MapAllFeatures2(R, Count)$
2:     $newFeatures \leftarrow \emptyset$;
3:     $Sort\ R.F\ set\ after\ a\ specific\ sore$;
4:     for $i = 1 \rightarrow Count$ do
5:         for $j = i + 1 \rightarrow Count$ do
6:             $newFeatures \leftarrow newFeatures \cup R.F_i\ |\overline{OP}|\ R.F_j$;
7:         end for
8:     end for
9:     $R.F \leftarrow R.F \cup newFeatures$;
10: end function
```

Although the detection rate increases with the $Count$ variable (the number of iterations), the training time will increase with the number of features and this is a problem from a practical point of view.

The next approach would be to map all the features, sort them based in different scores and choose the most relevant combinations. This approach poses a risk: some features from the original set may not appear in the new set (if $Count$ is too low). This will reflect in the detection rate of the OSC-based algorithm.

Possible solutions to be considered:

1. First *CreateSortedIndexList* function is defined (Algorithm 3) that creates an AND production indexes. This list of indexes is sorted by applying a specific score to the combination of features.

Algorithm 3 CreateSortedIndexList algorithm

```
1: function $CreateSortedindexList(R)$
2:     $idx \leftarrow \bigcup_{i=1}^{|R.F|} \{\bigcup_{j=i}^{|R.F|} \{i, j\}\}$;
3:     $sort\ idx\ after\ a\ specific\ function$;
4:     $CreateSortedindexList \leftarrow idx$;
5: end function
```

2. Add to the new set of features the features from the original set that were missed in their original form (Algorithm 4)

Algorithm 4 MAP-2 algorithm

```
1: function $Map2(R, Count)$
2:     $idx \leftarrow CreateSortedIndexList(R)$;
3:     $featIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i$;
4:     $featIdx \leftarrow featIdx \bigcup_{i=1}^{|R.F|} \{i, i\}$;
5:     $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$;
6: end function
```

3. Add to the new set of features the features from the original set that were missed both in their original form or combined with some other feature. The main advantage in this case is that one can reduce the number of features while preserving every feature (either as it was or combined)—Algorithm 5.

Algorithm 5 MAP-3 algorithm

```
1: function $Map3(R, Count)$
2:     $idx \leftarrow CreateSortedIndexList(R)$;
3:     $featIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i$;
4:     $usedFeatIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} \{\bigcup_{j=1}^{idx_i} idx_{i_j}\}$;
5:     $featIdx \leftarrow featIdx \bigcup_{i=1}^{|R.F|} \{\{i, i\}\}, i \notin usedFeatIdx$;
6:     $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$;
7: end function
```

4. Instead of adding the original features that were not used at all (like in the previous algorithm), we can add the best combination where the missing feature is first used—Algorithm 6.

Algorithm 6 MAP-4 algorithm

```
1: function $Map4(R,\ Count)$
2:     $idx \leftarrow CreateSortedIndexList(R)$;
3:     $featIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} idx_i$;
4:     $usedFeatIdx \leftarrow \bigcup_{i=1}^{\min(|idx|, Count)} \{\bigcup_{j=1}^{idx_i} idx_{i_j}\}$;
5:     for $i = 1 \rightarrow |idx|$ do
6:         $allFeatUsed \leftarrow True$
7:         $allFeatUsed \leftarrow False,\ if\ idx_{i_j} \notin usedFeatIdx,\ j = \overline{1, |idx_i|}$
8:         $usedFeatIdx \leftarrow usedFeatIdx \bigcup_{j=1}^{idx_i} idx_{i_j}, idx_{i_j} \notin usedFeatIdx$;
9:         $featIdx \leftarrow featIdx \cup \{idx_i\},\ if\ not\ allFeatUsed$;
10:    end for
11:    $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$;
12: end function
```

From the previous three algorithms we can extract a new one to compute the exact number of combinations needed so that all original features appear at least in one combination (Algorithm 7).

Algorithm 7 MAP-5 algorithm

```
1: function $Map5(R,\ K)$
2:     $idx \leftarrow CreateSortedIndexList(R)$;
3:     $featIdx \leftarrow \emptyset$;
4:     $cFIdx_i \leftarrow 0, i = \overline{1, n}, n = |R.F|$;
5:     for $i = 1 \rightarrow |idx|$ do
6:         $allFeatUsed \leftarrow True$;
7:         $allFeatUsed \leftarrow False,\ if\ cFIdx_{idx_{i_j}} < K, j = \overline{1, |idx_i|}$;
8:         $cFIdx_{idx_{i_j}} \leftarrow cFIdx_{idx_{i_j}} + 1,\ if\ cFIdx_{idx_{i_j}} < K, j = \overline{1, |idx_i|}$
9:         $featIdx \leftarrow featIdx \cup \{idx_i\},\ if\ not\ allFeatUsed$;
10:    end for
11:    $R.F \leftarrow \bigcup_{i=1}^{|featIdx|} \{\bigwedge_{k=1}^{featIdx_i} (R.F_{featIdx_{i_k}})\}$;
12: end function
```

This algorithm guarantees that every feature is used at least "K" times ("K" should be at least 1 so that each original feature is used at least one time). The bigger the "K", the larger the number of resulted features and, as shown in the results section, the bigger the detection rate.

3.2 Detection systems based on the use of multiple perceptron algorithms

3.2.1 Ensemble systems

As previously discussed another way of improving detection rate is to use an ensemble-like algorithm. The used algorithms must be able to separate a subset of records belonging to the same class.

By using an OSC-based algorithm, a hyper-plane is created to correctly classify, for instance, all the clean files. This means that all the clean files along with some incorrectly classified malicious ones will be on one side of the

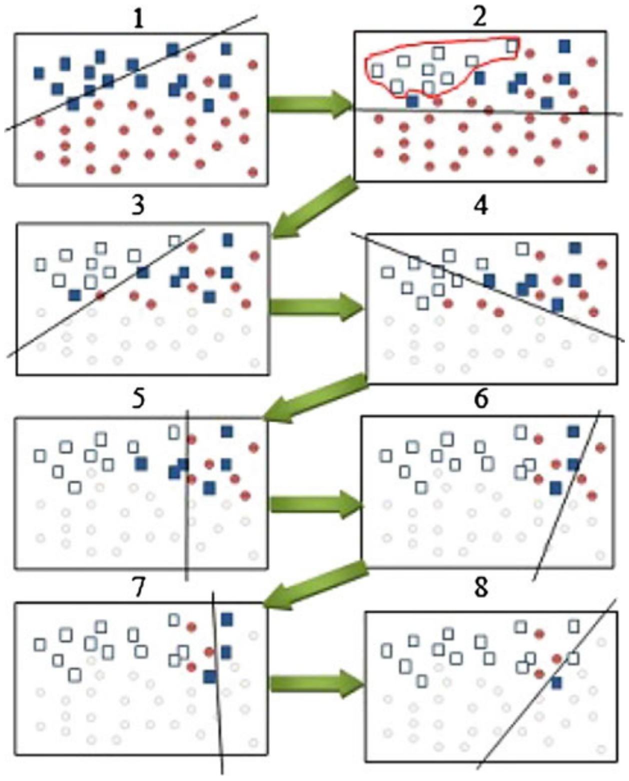

Fig. 2 Ensemble algorithm (8 ensemble iteration)

hyperplane while the other side will contain only malicious samples. The subset of malware files correctly classified will be removed from the data-base at the next ensemble iteration as illustrated in Fig. 2 (e.g. the red selected records from the second ensembled iteration) and Algorithm 8. For the OSC-based algorithms it is better to find different labeled subsets to remove from each ensemble iteration.

Algorithm 8 Ensemble algorithm

```
1: function $Ensemble(R,\ Count)$
2:     $Models \leftarrow \emptyset; index \leftarrow 0$;
3:     while $(index < Count)\ AND\ (|R| > 0)$ do
4:         $M \leftarrow OSCAlgorithm(R,\ Positive)$;
5:         $R \leftarrow R \setminus \{TP(R, M)\}$;
6:         $Models \leftarrow Models \cup \{M\}$;
7:         $M \leftarrow OSCAlgorithm(R,\ Negative)$;
8:         $R \leftarrow R \setminus \{TN(R, M)\}$;
9:         $Models \leftarrow Models \cup \{M\}$;
10:        $index \leftarrow index + 1$;
11:    end while
12: end function
13: $where:$
14: $TP(R, M) - true\ positive\ elements\ obtained\ from\ R\ with\ M$
15: $TN(R, M) - true\ negative\ elements\ obtained\ from\ R\ with\ M$
```

Theoretically, if the number of ensemble iterations is infinite, all the records are classified correctly. In practice the variable has a finite value, which limits the detection rate. After the first ensemble iteration (when a big number of records are removed from the set) the records removed on subsequent iterations are fewer. For a large data-base it is hard to find a balance between a reasonable value for the variable *Count* and a good detection rate.

3.2.2 Vote systems

We used a modified version of the voted perceptron. In the original form, the voted perceptron keeps a list of all prediction vectors generated after each misclassified element and counts how many iterations each vector lasts. By using this number of iterations as votes, the models surviving the most (fewer mistakes are made using this model) will have a greater weight in the majority vote [8]. Keeping many models at a time makes this method impractical. We will use the fact that the final result will be a weighted average of the votes given by every participating algorithm, but we will reduce the number of algorithms to very few. We also propose to use different votes for each class of elements. This way, if an algorithm is better at detecting one class than the other (i.e. OSC 3 which was trained for zero false positives) it will give a higher vote for elements belonging to that class and a very low vote for the others.

The next approach is to use a vote system based on a set of templates *Models*. Notations to be considered:

1. M is an element of the set *Models*, one of the used machine learning algorithms.
2. M.PositiveVote (Record) is the vote weight that model M uses in the vote system in determining if a Record should have a positive label (in this case if the Record corresponds to a malware file).
3. M.NegativeVote (Record) is vote weight that model M uses in the vote system in determining if a Record should have a negative label (in this case if the Record corresponds to a clean file).

The vote system is defined in Algorithm 9:

Algorithm 9 Vote system algorithm

```
1:  function $VoteSystem(R,\ Models)$
2:      for $i = 1 \to |R|$ do
3:          $posV \leftarrow initPositiveValue$;
4:          $negV \leftarrow initNegativeValue$;
5:          $posV \leftarrow posV\ |\overline{OP}|\ Models_j.PosVotes(R_i),\ j = 1, |Modes|$;
6:          $negV \leftarrow negV\ |\overline{OP}|\ Models_j.NegVotes(R_i),\ j = 1, |Modes|$;
7:          if $IsPositive(posV, negVs)$ then
8:              $declare\ R_i\ as\ positive$;
9:          else
10:             $declare\ R_i\ as\ negative$;
11:         end if
12:     end for
13: end function
```

Where *IsPositive* function decides if the result should be considered positive or negative Algorithm 10:

Algorithm 10 IsPositive function

```
1: function $IsPositive(positiveVotes,\ negativeVotes)$
2:     if $positiveVotes \geq negativeVotes$ then
3:         $IsPositive \leftarrow True$;
4:     else
5:         $IsPositive \leftarrow False$;
6:     end if
7: end function
```

The initial values of the variables (*initPositiveValue* and *initNegativeValue*) depend on the used operation. If ***OP*** is an addition operation these values should be 0 and if it is multiplication these variables should be 1.

There are many ways to build these models and here are some approaches:

1. Homogenous system: all the models are obtained from the same algorithm, with different parameters.
2. Non-homogenous system: each model will be obtain from a different algorithm

Some interesting results can be obtained if the models are trained on different data sets:

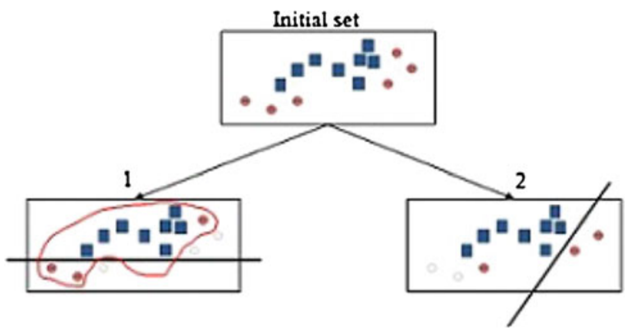

Fig. 3 Data-base splitting using random subsets. From the initial set two models were obtained

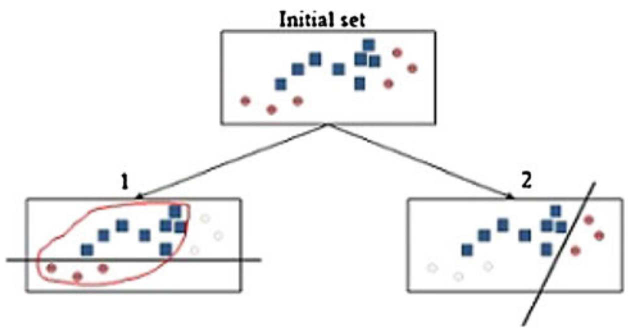

Fig. 4 Data-base splitting using a clustering algorithm. From the initial set two models were obtained

1. Split the initial data-base in multiple subsets, each used to train one model.
2. If an OSC-based algorithm is used, the data can be split in the following manner: each subset has all the records labeled as clean and some records labeled as malware(random split)—Fig. 3.
3. In the case of the OSC algorithm it would be more efficient to cluster records labeled as malware and each subset contain one cluster (Fig. 4).

3.2.3 Hybrid method

From our experience in malware analyses it is easier to create several classification models (one for each malware family), than it is to create just a single model to correctly classify all the malware files. The hybrid solution that was considered can be seen as a decision tree which acts as a filter for a set of models. The result would be a set of rules that can generate subsets of records (Algorithm 11).

Algorithm 11 Decision tree algorithm

```
1: function $DecisionTree(R,\ Path)$
2:     $Store\ the\ path,\ if\ |Path| \geq a\ specific\ depth$
3:     $sortedFeat \leftarrow sort\ all\ features\ from\ R\ after\ a\ score;$
4:     $selectedFeat \leftarrow \emptyset\ ; i \leftarrow 1;$
5:     while $(i \leq |sortedFeatures|) AND (selectedFeature = \emptyset)$ do
6:         if $(sortedFeat_i \in Path) OR (NOT sortedFeat_i \in Path)$ then
7:             $selectedFeat \leftarrow sortedFeat_i;$
8:         end if
9:         $i \leftarrow i + 1;$
10:    end while
11:    $R^+ \leftarrow \emptyset;\ R^- \leftarrow \emptyset;$
12:    $R^+ \leftarrow R^+ \cup \{R_i\},\ if\ selectedFeat\ \neq \emptyset, i = \overline{1, |R|};$
13:    $R^- \leftarrow R^- \cup \{R_i\},\ if\ selectedFeat\ \neq \emptyset, i = \overline{1, |R|};$
14:    $DecisionTree(R^+,\ Path \cup \{selectedFeat\});$
15:    $DecisionTree(R^-,\ Path \cup \{NOT\ selectedFeat\});$
16: end function
```

For every extracted subset obtained with this method, a model can be build using an OSC-3 algorithm. This particular solution has the great advantage of obtaining much smaller and easy to process subsets. Important for the improvement of the detection rate is the score function used in the decision tree algorithm, that chooses the most relevant features that decide how the main and resulted subsets will split. There are several function that can be used: F2 score, or MedianClose score.

The MedianClose score defined as follows:

$$MedianClose(Feat) = \frac{2 - abs\left(\frac{Feat.pozCount}{PozCount} - 0.5\right) - abs\left(\frac{Feat.negCount}{NegCount} - 0.5\right)}{2}$$

Where:

1. Feat → a specific feature.
2. Feat.pozCount → number of positively labeled records (1) from the records set that have this feature set.
3. Feat.negCount → number of negatively labeled records (−1) from the records set that have this feature set.
4. PozCount → number of positively labeled records (1) from the records set.
5. NegCount → number of negatively labeled records (−1) from the records set.

4 Results

The above mentioned algorithms were implemented in C++ programming language and tested on the same computer. The data-base used for the tests has 3,073,228 records (with 937,831 of them being labeled as malware and 2,135,397 of them being labeled as clean). Each record has 1,000 Boolean features extracted. Some files that were not suitable for a linear classifier algorithm(such as file infectors, damage files, gray-ware files, patched files…) were removed from the data-base.

The first test is a comparison between the mapping methods. All the algorithms using the mapped features are OSC-3. The simple OSC-3 algorithm (without any map features) was added to illustrate the increased detection rate.

The algorithms were trained for more than 1,000 iterations. Notations:

1. OSC-3 algorithm.
2. OSC-3-MAP3-F2-400. OSC-3 algorithm using map feature algorithm MAP-3, F2 measure as the score function for the sort algorithm and $Count = 400$.
3. OSC-3-MAP5-F2-2. OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 2$.
4. OSC-3-MAP5-F2-3. OSC-3 OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 3$.
5. OSC-3-MAP5-F2-1-ORIG. OSC-3 algorithm using map feature algorithm MAP-5, F2 measure as the score function for the sort algorithm and $K = 1$. In addition, all of the original features that were not used were added as well.

Figure 5 illustrates the results for these algorithms:

As shown in Fig. 5, most algorithms have a better detection rate than the pure OSC-3 algorithm.

The second test consists in checking the detection improvement for different ensemble systems. Notations:

1. ENS-1 (ensemble with only one ensemble iteration Count = 1).
2. ENS-2 (ensemble with two ensemble iterations Count = 2).

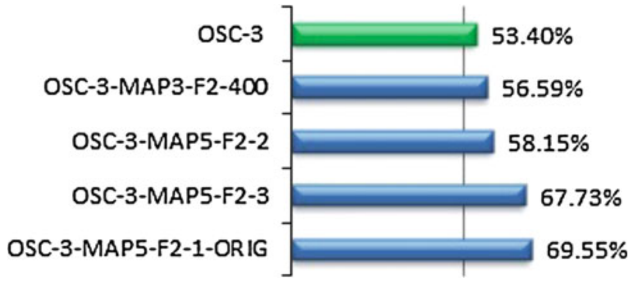

Fig. 5 Detection rate for different MAP based algorithms. OSC-3 algorithm is added as a reference

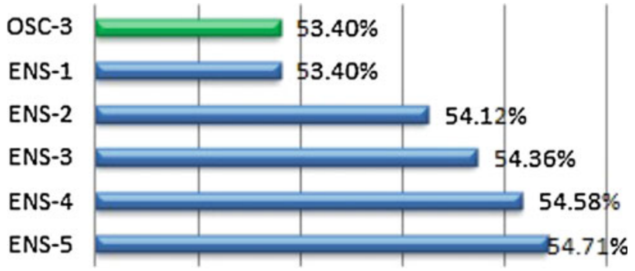

Fig. 6 Detection rate for different ensemble algorithms variations. OSC-3 algorithm is added as a reference

3. ENS-3 (ensemble with three ensemble iterations Count = 3).
4. ENS-4 (ensemble with four ensemble iterations Count = 4).
5. ENS-5 (ensemble with five ensemble iterations Count = 5).

Figure 6 illustrates the results for these algorithms:

The third experiment had to determine the detection rate for two different vote-systems:

1. VOT-S → Simple vote system. The models have all the clean files and subsets of malware files generated in a random manner. The number of used models were 1,000. The voting weight for clean files was set to be very high: if at least one model decides that a file is clean, than that file is clean.
2. VOT-C → Cluster vote system. The created models were based on clusters of all the malware files. The number of resulted models were 45. The clusters were created based on the Manhattan distance between two files.

Figure 7 illustrates the results for these algorithms:

The fourth and the last test was to examine a hybrid approach: decision tree combined with an OSC-3 algorithm. The algorithms used in this test were:

1. OSC-3 → OSC-3 simple algorithm was added as base for this comparation.

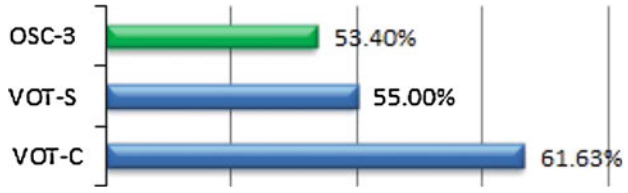

Fig. 7 Detection rate for different VOTE algorithm variations. OSC-3 algorithm is added as a reference

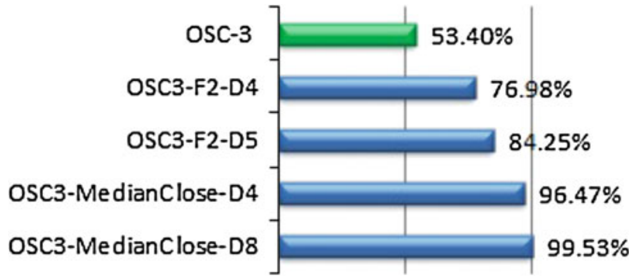

Fig. 8 Detection rate for different decision-tree algorithms variations

Table 2 False Positives on a threefold cross-validation

Algorithm	False positives (%)	Detection (%)
OSC3-F2-D4	0.0628	94.70
OSC3-F2-D5	0.0884	94.27
OSC3-MedianClose-D4	0.0739	97.09
OSC3-MedianClose-D8	0.8926	98.75

2. OSC3-F2-D4 → OSC-3 algorithm, over a decision tree using F2 score and depth = 4 (16 models).
3. OSC3-F2-D5 → OSC-3 algorithm, over a decision tree using F2 score and depth = 5 (32 models).
4. OSC3-MedianClose-D4 → OSC-3 algorithm, over a decision tree using MedianClose score and depth = 4 (16 models).
5. OSC3-MedianClose-D8 → OSC-3 algorithm, over a decision tree using MedianClose score and depth = 8 (256 models).

Figure 8 illustrates the results for these algorithms:

To get a better understanding of how these algorithms behave with respect to false positives in a real life situation we've performed threefold cross validation test. The results are illustrated in Table 2. As shown, the OSC3-MedianClose-D4 algorithm is suited best for a practical use due to its very low percentage of false positives and its high detection rate.

5 Conclusion

When we first created OSC3 algorithm and noticed its medium detection rate and its small number of false positives, we decided it made a suitable complementary detection method. This paper takes the algorithm forward. Based on its very high detection rate (over 99 %) and its small number of false positives, this algorithm alone qualifies as an extremely efficient method to detect malware in real situations. We encountered many difficulties while designing this detection method. The most important one was the time needed to train different algorithms on very large data sets. This alone made testing of new ideas very difficult. The current result wouldn't have been possible if it weren't for the results we achieved in out previous paper when we significantly decreased the training time of the one side class perceptron.

Using hybrid algorithms (OSC-3 algorithm and decision tree) proved very efficient and we are thinking of further incorporating in algorithms these types of combinations. As future assignment, we plan to include this method in the Bitdefender cloud service. Parallelism shortens the training time and makes this method appropriate for a protection mechanism that needs to adapt every few hours to detect new malware found in the wild.

References

1. Aizerman, M., Braverman, E., Rozonoer, L.: Theoretical foundations of the potential function method in pattern recognition learning. Autom. Remote Control, 821–837 (1964)
2. Altaher, A., Ramadass, S., Ali, A.: Computer virus detection using features ranking and machine learning. Aust. J. Basic Appl. Sci., 1482–1486 (2011)
3. Avtest. http://www.av-test.org/en/statistics/malware/ (2012)
4. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: Map-reduce for machine learning on multicore. In: NIPS , pp. 281–288 (2006)
5. Dai, J., Guha, R., Lee, J.: Effcient virus detection using dynamic instruction sequences. J. Comput., 405–414 (2009)
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: OSDI'04, sixth symposium on operating system design and implementation, San Francisco (2004)
7. Domingos, P.: Metacost: a general method for making classifiers costsensitive. In: Proceedings of the fifth international conference on knowledge discovery and data mining, pp. 155–164 (2009)
8. Freund, Y., Schapire, R.E.: Large margin classification using the perceptron algorithm. Mach. Learn., 277–296 (1999)
9. Gavrilut, D., Cimpoesu, M., Anton, D., Ciortuz, L.: Malware detection using machine learning. In: Proceedings of the international multiconference on computer science and information technology, IMCSIT 2009, Mragowo, 12–14 October 2009, pp. 735–741 (2009)
10. Gavrilut, D., Vatamanu, C., Benchea, R.: Optimized zero false positives perceptron training for malware detection. In: Proceedings of SYNASC conference, Timisoara (2012)
11. Hung, T.C., Lam, D.X.: A feature extraction method and recognition algorithm for detection unknown worm and variations based on static features. Cyber J. Multidiscip. J. Sci. Technol. J. Select. Areas Softw. Eng. (JSSE) (2011)
12. Jiang, Q., Zhao, X., Huang, K.: A feature selection method for malware detection. Inform. Autom. In: IEEE international conference, pp. 890–895 (2011)
13. Kolcz, A., Alspector, J.: Svm-based filtering of e-mail spam with content-specific misclassification costs. In: IEEE international conference on data mining (2001)
14. Langford, J., Smola, A., Zinkevich, M.: Slow learners are fast. J. Mach. Learn. Res., 1–9 (2009)
15. Leather, H., Bonilla, E., O'Boyle, M.: Automatic feature generation for machine learning based optimizing compilation. In: Code generation and optimization international, symposium, pp. 81–91 (2009)
16. McDonald, R., Hall, K., Mann, G.: Distributed training strategies for the structured perceptron. In: HLT '10 human language technologies: the 2010 annual conference of the North American chapter of the association for, computational linguistics, pp. 456–464 (2002)
17. Rosenblatt, F.: The perceptron: a probabilistic model for information storage and organization in the brain. Psychol. Rev., 386–407 (1958)
18. Stanley, Kwang Loong, Mishra, Santosh K.: De novo svm classification of precursor micronas from genomic pseudo hairpins using global and intrinsic folding measures. J. Bioinform. **23**, 1321–1330 (2007)
19. Tretyakov, K.: Machine learning techniques in spam filtering. In: Data mining problem-oriented, Seminar, pp. 60–79 (2004)
20. Whitney, M., Clifton.A., Sarkar A., Fedorova A.: Making the most of a distributed perceptron for NLP. In: proceedings of Nortwest NLP (2012)
21. Ye, Y., Chen, L., Wang, D., Li, T., Jiang, Q., Zhao, M.: Sbmds: an interpretable string based malware detection system using svm ensemble with bagging. J. Comput. Virol., 283–293 (2009)
22. Yih, W., Goodman, J., Hulten, G.: Learning at low false positiverates. In: Proceedings of the 3rd conference on email and anti-spam (2006)
23. Zhang, B., Yin, J., Hao, J.: Using fuzzy pattern recognition to detect unknown malicious executables code. Fuzzy Syst. Knowl. Discov., 629–634 (2005)