

# Malware Detection Using Machine Learning

Dragoş Gavriluţ<sup>1,2</sup>, Mihai Cimpoesu<sup>1,2</sup>, Dan Anton<sup>1,2</sup>, Liviu Ciortuz<sup>1</sup>

1 - Faculty of Computer Science, “Al. I. Cuza” University of Iaşi, Romania

2 - BitDefender Research Lab, Iaşi, Romania

Email: {gdt, mcimpoesu, dan.anton, ciortuz}@info.uaic.ro

**Abstract**—We propose a versatile framework in which one can employ different machine learning algorithms to successfully distinguish between malware files and clean files, while aiming to minimise the number of false positives. In this paper we present the ideas behind our framework by working firstly with cascade one-sided perceptrons and secondly with cascade kernelized one-sided perceptrons. After having been successfully tested on medium-size datasets of malware and clean files, the ideas behind this framework were submitted to a scaling-up process that enable us to work with very large datasets of malware and clean files.

## I. INTRODUCTION

Malware is defined as software designed to infiltrate or damage a computer system without the owner’s informed consent. Malware is actually a generic definition for all kind of computer threats. A simple classification of malware consists of file infectors and stand-alone malware. Another way of classifying malware is based on their particular action: worms, backdoors, trojans, rootkits, spyware, adware etc.

Malware detection through standard, signature based methods [1] is getting more and more difficult since all current malware applications tend to have multiple polymorphic layers to avoid detection or to use side mechanisms to automatically update themselves to a newer version at short periods of time in order to avoid detection by any antivirus software. For an example of dynamical file analysis for malware detection, via emulation in a virtual environment, the interested reader can see [2]. Classical methods for the detection of metamorphic viruses are described in [3].

An overview on different machine learning methods that were proposed for malware detection is given in [4]. Here we give a few references to exemplify such methods.

- In [5], boosted decision trees working on  $n$ -grams are found to produce better results than both the Naive Bayes classifier and Support Vector Machines.
- [6] uses automatic extraction of association rules on Windows API execution sequences to distinguish between malware and clean program files. Also using association rules, but on honeytokens of known parameters, is [7].
- In [8] Hidden Markov Models are used to detect whether a given program file is (or is not) a variant of a previous program file. To reach a similar goal, [9] employs Profile Hidden Markov Models, which have been previously used with great success for sequence analysis in bioinformatics.
- The capacity of neural networks to detect polymorphic malware is explored in [10]. In [11], Self-Organizing Maps are

used to identify patterns of behaviour for viruses in Windows executable files.

In this paper, we present a *framework* for malware detection aiming to get as few false positives as possible, by using a simple and a simple multi-stage combination (cascade) of different versions of the perceptron algorithm [12]. Other automate classification algorithms [13] could also be used in this framework, but we do not explore here this alternative. The *main steps* performed through this framework are sketched as follows:

1. A set of *features* is computed for every binary file in the training or test *datasets* (presented in Section II), based on many possible ways of analyzing a malware.
2. A machine learning system based firstly on one-sided perceptrons, and then on feature mapped one-sided perceptrons and a kernelized one-sided perceptrons (Section III), combined with feature selection based on the F1 and F2 scores, is trained on a medium-size dataset consisting of clean and malware files. Cross-validation is then performed in order to choose the right values for parameters. Finally, tests are performed on another, non-related dataset. The obtained results (see Section IV) were very encouraging.
3. In the end (Section V) we will analyse different aspects involved in the scale-up of our framework to identifying malware files on very large training datasets.

## II. DATASETS

We used three datasets: a *training* dataset, a *test* dataset, and a “*scale-up*” dataset. The number of malware files and respectively clean files in these datasets is shown in the first two columns of Table I. As stated above, our main goal is to achieve malware detection with only a few (if possible 0) false positives, therefore the clean files in this dataset (and also in the scale-up dataset) is much larger than the number of malware files.

From the whole feature set that we created for malware detection, 308 binary features were selected for the experiments to be presented in this paper. Files that generate similar values for the chosen feature set were counted only once. The last two columns in Table I show the total number of unique combinations of the 308 selected binary features in the training, test and respectively scale-up datasets. Note that the number of clean combinations — i.e combinations of feature values for the clean files — in the three datasets is much smaller than the number of malware unique combinations

TABLE I  
NUMBER OF FILES AND UNIQUE COMBINATIONS OF FEATURE VALUES IN THE TRAINING, TEST, AND SCALE-UP DATASETS.

Database	Files		Unique combinations	
	malware	clean	malware	clean
Training	27475	273133	7822	415
Test	11605	6522	506	130
Scale-up	approx. 3M	approx. 180M	12817	16437

TABLE II  
MALWARE DISTRIBUTION IN THE TRAINING AND TEST DATASETS.

Malware type	Training Dataset		Test Dataset
	Files	Unique combinations of feature values	Files
Backdoor	35.52%	40.19%	9.16%
Hacktool	1.53%	1.73%	0.00%
Rootkit	0.09%	0.15%	0.04%
Trojan	48.06%	43.15%	37.17%
Worm	12.61%	12.11%	33.36%
Other malware	2.19%	2.66%	20.26%

because most of the features were created to emphasize an aspect (either a geometrical form or behaviour aspect) of malware files.

The clean files in the training database are mainly system files (from different versions of operating systems) and executable and library files from different popular applications. We also use clean files that are packed or have the same form or the same geometrical similarities with malware files (e.g use the same packer) in order to better train and test the system.

The malware files in the training dataset have been taken from the Virus Heaven collection. The test dataset contains malware files from the WildList collection and clean files from different operating systems (other files than the ones used in the first database). The malware collection in the training and test datasets consists of trojans, backdoors, hacktools, rootkits, worms and other types of malware. The first and third columns in Table II represent the percentage of those malware types from the total number of files of the training and respectively test datasets. The second column in Table II represents the corresponding percentage of malware unique combinations from the total number of unique combinations of feature values for the training dataset. As shown in the first and last column of Table II, the distribution of malware types in this test dataset is significantly different from the malware distribution in the training dataset.

The third, large dataset was used for testing the scaling-up capabilities of our learning algorithms. This dataset was divided into 10 parts denoted as  $S_{10}, S_{20}, \dots, S_{100}$ , where  $S_i$  represents  $i\%$  of the total dataset, and  $S_i \subset S_{i+10}$ . We used these parts in order to evaluate the training speed and the malware detection rate for larger and larger datasets.

### III. ALGORITHMS

The main goal of this section is to modify the perceptron algorithm [12], so as to correctly detect malware files, while

---

#### Algorithm 1 The Perceptron Training Subroutine

---

```

Sub Train ( $R, LR\_Malware, LR\_Clean$ ) :
 $\gamma_i = 0$ 
 $i = 1, \dots, n$ 
for all record in  $R$  do
  if Fitness(record)  $\neq$  record.label then
    for all  $\gamma_i$  do
      if record. $F_i \neq 0$  then
        if record.label = 1 then
           $\gamma_i = \gamma_i + LR\_Malware$ 
        else
           $\gamma_i = \gamma_i + LR\_Clean$ 
        end if
      end if
    end for
  end if
end for
for all  $w_i$  do
   $w_i = w_i + \gamma_i$ 
end for
End sub

```

---

forcing (if possible) a 100% detection rate for one category. In the sequel we will use the following data structures:

- $F = (f_{a1}, f_{a2}, \dots, f_{an})$  is an array representing the feature values associated to a file, where  $f_{ai}$  are file features.
- $R_i = (F_i, label_i)$  is a record, where  $F_i$  is an array of file feature as above, and  $label_i$  is a boolean tag. The value of  $label_i$  identifies the file characterised by the array of feature values  $F_i$  as being either a malware file or a clean file.
- $R = (R_1, R_2, \dots, R_m)$  is the set of records associated to the training files that we use.

We use non-stochastic versions of the perceptron algorithm so that we can parallelize the training process. This measure will enable us to speed up the training process on large datasets.

Algorithm 1 is the the standard *perceptron* algorithm. Instead of working with floats, it uses a large integer representation for the weights  $w_i, i = 1 \dots n$ , where  $n$  is the total number of attributes/features. This adaptation is non-restrictive, assuming multiplication with a certain factor related to the representation. Other notations used by this algorithm are:

- $\gamma_i, i = 1 \dots n$ , are the additive values that will modify the weights  $w_i$  after one iteration;
- $LR\_Malware$  and  $LR\_Clean$  are the learning rate constants for the malware and respectively the clean sets of files.

The perceptron's fitness function is defined as:

**Fitness** ( $R_i$ ) = **Sign**( $\sum_{j=0}^n w_j R_i.F_j - Threshold$ ), where  $R_i.F_j$  denotes the value of the feature  $f_{aj}$  in the file record  $R_i$ . We define **Sign**( $x$ ) = 1 if  $x \geq 0$ , and -1 otherwise.

Algorithm 2, henceforth called the *one-sided perceptron* is a modified version of Algorithm 1. It performs the training

---

**Algorithm 2** One-Sided Perceptron

---

```
NumberOfIterations  $\leftarrow$  0
MaxIterations  $\leftarrow$  100
repeat
  Train ( $R$ , 1, -1)
  while FP( $R$ ) > 0 do
    Train ( $R$ , 0, -1)
  end while
  NumberOfIterations  $\leftarrow$  NumberOfIterations + 1
until (TP( $R$ ) = NumberOfMalwareFiles) or
      (NumberOfIterations = MaxIterations)
```

---

for one chosen label (in our case either malware or clean), so that in the end the files situated on one side of the learned linear separator have exactly that label (assuming that the two classes are separable). The files on the other side of the linear separator can have mixed labels.

In the specification of the one-sided perceptron, the **FP**( $R$ ) function is assumed to return the number of false positives for the  $R$  set, while the **TP**( $R$ ) function returns the number of true positives for the  $R$  set.

There are two steps inside the repeat loop of Algorithm 2. The first step, **Train**( $R$ , 1, -1), performs usual training on the labeled data, obtaining a linear separator (see the perceptron algorithm). The second step, **while FP**( $R$ ) > 0 **do Train**( $R$ , 0, -1), tries to further move the linear separator, until no clean file is eventually misclassified.

For what we call the *mapped one-sided perceptron*, we will use the previous perceptron algorithm, except we first map all our features in a different space using a simple feature generation algorithm, namely Algorithm 3.

Remember that we noted  $F = (f_{a_1}, f_{a_2}, \dots, f_{a_n})$ . We map  $F$  to  $F'$  so that  $F' = (f'_{a_1}, f'_{a_2}, \dots, f'_{a_m})$ , where  $m = n(n+1)/2$  and  $f'_{a_k} = f_{a_i} \& f_{a_j}$ ,  $i = \lfloor k/n \rfloor + 1$ ,  $j = k \% n + 1$ ,  $k = 1 \dots m$ , where  $\&$  denotes the logical *and* operator.

---

**Algorithm 3** Simple Feature Generation

---

```
pos  $\leftarrow$  0
for  $i = 1$  to  $n$  do
  for  $j = 1$  to  $n$  do
     $f'_{a_{pos}} \leftarrow f_{a_i} \& f_{a_j}$ 
     $pos \leftarrow pos + 1$ 
  end for
end for
```

---

The number of resulted features in  $F'$  will be  $n(n+1)/2$ , where  $n$  is the number of features in  $F$ . The computational time increases heavily (e.g. for 308 features in  $F$ , we will have 47586 features in  $F'$ ). However, the detection rate (i.e. sensitivity) at cross-validation increases with about 10%, as it will be shown later in the results section.

Finally, we used the same one-sided perceptron (Algorithm 2), but in the dual form [14] and with the training entry mapped into a larger feature space via a kernel function

$K$  [15]. The resulting *kernelized one-sided perceptron* is the Algorithm 4 given below.

---

**Algorithm 4** Kernelized One-Sided Perceptron

---

```
for  $i = 1$  to  $n$  do
   $\Delta_i \leftarrow 0$ 
   $\alpha_i \leftarrow 0$ 
end for
for  $i = 1$  to  $n$  do
  if ( $label_i \times \sum_{j=1}^n (\alpha_j \times K(i, j))$ )  $\leq 0$  then
     $\Delta_i \leftarrow \Delta_i + label_i$ 
  end if
end for
for  $i = 1$  to  $n$  do
   $\alpha_i \leftarrow \alpha_i + \Delta_i$ 
   $\Delta_i \leftarrow 0$ 
end for
```

---

The algorithms 1, 2 and 4 presented above will be used in the sequel as bricks in *cascade* (or: multi-stage) classification algorithms. Given a set of binary classification algorithms  $\{A_1, A_2, \dots, A_k\}$ , a *cascade* over them is an aggregated classification algorithm that classifies a given test instance  $x$  as follows.<sup>1</sup>

---

**Algorithm 5** Cascade Classification

---

```
if  $A_{i_1}(x)$  or  $A_{i_2}(x)$  or  $\dots$   $A_{i_k}(x)$  then
  return 1
else
  return -1
end if
```

---

## IV. RESULTS

We performed cross-validation tests by running the three versions of the cascade one-sided perceptron presented in Section III on the training dataset described in Section II (7822 malware unique combinations, and 415 clean unique combinations).

For the kernelized one-sided perceptron, the following kernel functions were used:

- Polynomial Kernel Function:  
 $K(u, v) = (1 + \langle u, v \rangle)^d$ , where  $\langle u, v \rangle$  denotes the dot product of the  $u$  and  $v$  vectors;
- Radial-Base Kernel Function:  
 $K(u, v) = \exp\left(\frac{-\|u-v\|^2}{2 \times \sigma^2}\right)$ .

The results shown by the tables in this section have been obtained for the previously presented algorithms.

The following notations will be used in the sequel:

- COS-P – Cascade One-Sided Perceptron;
- COS-P-Map-F1 and COS-P-Map-F2 – Cascade One-Sided Perceptron with explicitly mapped features, after

<sup>1</sup>It is understood that the execution of the  $A_{i_1}, A_{i_2}, \dots, A_{i_k}$  algorithms that occur in the definition of the cascade algorithm is done in this order.

application of feature selection based on the F1 and F2 statistical scores;

- COS-P-Poly2/Poly3/Poly4 – Cascade One-Sided Perceptron with the Polynomial Kernel Function shown above, with the degree 2/3/4;
- COS-P-Radial – Cascade One-Sided Perceptron using the RBF Kernel Function with  $\sigma = 0.4175$ .
- TP – the number of true positives;
- FP – the number of false positives;
- SE – the sensitivity measure value;
- SP – the specificity measure value;
- ACC – the accuracy measure value [16].

For *feature selection* in conjunction with one-sided perceptrons we used the F-scores statistical measures. The discriminative power of the  $i^{th}$  feature is described commonly by the statistical F1 and F2 scores defined below. The larger the values for the  $i^{th}$  feature, the more likely this feature possesses discriminative importance.

$$F1 = \frac{|\mu_i^+ - \mu_i^-|}{|\sigma_i^+ - \sigma_i^-|}, \quad F2 = \frac{(\mu_i^+ - \bar{\mu}_i)^2 + (\mu_i^- - \bar{\mu}_i)^2}{(\sigma_i^+)^2 + (\sigma_i^-)^2} \quad (1)$$

Here,  $\mu_i^+/\mu_i^-$  and  $\sigma_i^+/\sigma_i^-$  denote the means and standard deviations of the positive (+) and respectively negative (-) subsets of the training dataset. The numerator describes the discrimination between the two classes, while the denominator measures the discrimination within each of the two classes [17].

Our measurements have shown that there is a quite good (although non-linear) *correlation* between the F1 and F2 scores. That means that most probably we will get similar training results when taking the same percentage of (F1 and respectively F2) best-scored features.

Then we performed feature selection based on the F1 and F2 scores to see whether we can find a subset of features that will produce similar classification results with those obtained when using all features. Towards this aim, we have split the training dataset in 2 partitions, i.e a training partition with about 66% records and another one for testing with the rest of 33%. We tested the cascade one-sided perceptron (COS-P) algorithm using the first 10%, 20%, 30% ... 100% features selected with F1 and respectively F2 scores. The results (not shown here) indicate that both F1 and F2 with the first 30% features have similar results compared to the results obtained by the same (COS-P) algorithm when using all features.

The COS-P-Map algorithm, i.e the one-sided perceptron with explicitly mapping, uses a lot of features. This will slow down the training algorithm. This is why for the COS-P-Map-F1 algorithm we used 30% of the original features given by the best F1 score values, after which they were transformed into 1830 new features using Algorithm 3. It should be taken into account that after selecting the first 30% features, duplicates appeared in the training datasets and after the elimination, 6580 entries remained. COS-P-MAP-F2 works similarly to COS-P-Map-F1, with the only difference that we sorted the original features using the F2 score; 6644 records remained after duplicate elimination.

TABLE III  
3-FOLD CROSS-VALIDATION RESULTS ON THE TRAINING DATASET.

Algorithm	TP	FP	SE	SP	ACC
COS-P	2269	<b>9</b>	87.04%	<b>93.01%</b>	87.34%
COS-P-Map-F1	2023	36	97.62%	69.70%	96.08%
COS-P-Map-F2	2029	40	97.38%	69.05%	95.71%
COS-P-Poly2	2535	41	97.24%	70.12%	95.87%
COS-P-Poly3	2551	48	97.84%	64.82%	<b>96.18%</b>
COS-P-Poly4	2554	54	<b>97.95%</b>	60.97%	96.09%
COS-P-Radial	2534	48	70.49%	58.79%	69.90%

TABLE IV  
5-FOLD CROSS-VALIDATION RESULTS ON THE TRAINING DATASET.

Algorithm	TP	FP	SE	SP	ACC
COS-P	1342	<b>5</b>	85.83%	<b>93.98%</b>	86.24%
COS-P-Map-F1	1209	18	97.25%	74.09%	95.97%
COS-P-Map-F2	1212	17	96.98%	77.50%	95.83%
COS-P-Poly2	1518	23	97.05%	71.57%	95.76%
COS-P-Poly3	1532	29	97.95%	64.10%	<b>96.25%</b>
COS-P-Poly4	1533	31	<b>98.01%</b>	61.69%	96.18%
COS-P-Radial	1524	30	73.70%	60.00%	73.01%

*Cross-validation* tests for 3, 5, 7, and 10 folds were performed for each algorithm (COS-P, COS-P-Map, COS-P-Poly and COS-P-Radial) on the *training* dataset. For each algorithm, we used the best result from maximum 100 iterations.

Figure 1 shows a comparative view between the *F-measure* values produced by our algorithms at cross-validation on the training set. F-measure is defined as  $Fm = \frac{2(SP \times PPV)}{SP + PPV}$ , where  $PPV$  (Positive Predictive Value) =  $\frac{TP}{TP + FP}$  and  $TP/FP$  designate the number of true/false positives.

The cross-validation results found in Tables III–VI show that although the COS-P-Poly4 algorithm has the best malware detection rate (i.e sensitivity) on training dataset, the number of false alarms produced by this algorithm is much higher

TABLE V  
7-FOLD CROSS-VALIDATION RESULTS ON THE TRAINING DATASET.

Algorithm	TP	FP	SE	SP	ACC
COS-P	957	<b>3</b>	85.67%	<b>94.94%</b>	86.14%
COS-P-Map-F1	863	13	97.25%	74.37%	95.99%
COS-P-Map-F2	866	13	96.95%	76.47%	95.74%
COS-P-Poly2	1084	16	97.01%	72.05%	95.75%
COS-P-Poly3	1092	20	97.74%	64.81%	96.08%
COS-P-Poly4	1094	21	<b>97.92%</b>	63.12%	<b>96.16%</b>
COS-P-Radial	1085	30	74.09%	61.92%	73.47%

TABLE VI  
10-FOLD CROSS-VALIDATION RESULTS ON THE TRAINING DATASET.

Algorithm	TP	FP	SE	SP	ACC
COS-P	667	<b>2</b>	85.27%	<b>94.47%</b>	85.74%
COS-P-Map-F1	603	8	97.10%	75.43%	95.91%
COS-P-Map-F2	605	7	96.77%	80.06%	95.79%
COS-P-Poly2	758	11	97.00%	72.29%	95.75%
COS-P-Poly3	764	15	97.79%	63.83%	96.08%
COS-P-Poly4	766	15	<b>97.97%</b>	61.65%	<b>96.14%</b>
COS-P-Radial	761	14	85.07%	94.95%	85.57%

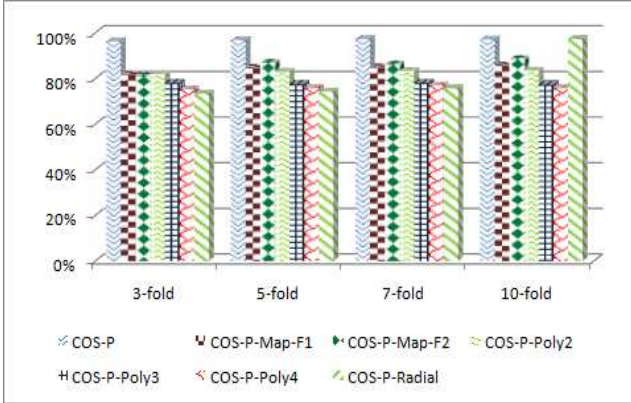


Fig. 1. Comparison of F-measure values for 3, 5, 7, 10-fold cross-validation with the cascade one-sided perceptron (COS-P) algorithm on the training dataset.

TABLE VII  
RESULTS ON THE TEST DATASET.

Algorithm	TP	FP	SE	SP	ACC
COS-P	356	3	68.73%	<b>97.46%</b>	74.06%
COS-P-Map-F1	356	<b>2</b>	83.76%	96.97%	85.54%
COS-P-Map-F2	357	<b>2</b>	83.22%	97.14%	85.17%
COS-P-Poly2	455	9	87.84%	92.37%	88.68%
COS-P-Poly3	466	19	89.96%	83.90%	<b>88.84%</b>
COS-P-Poly4	465	20	<b>89.77%</b>	83.05%	88.52%
COS-P-Radial	451	17	50.97%	83.90%	57.08%

than the one obtained for the COS-P algorithm. (Note that the number of files that are actually detected is much higher since the algorithm works with unique combinations of features and not with actual files.)

The results for the *test* dataset (Table VII) show that both COS-P-Map-F1 and COS-P-Map-F2 algorithms produce good results, with a good specificity (83%) and very few (2) false positives, even if the malware distribution in this dataset is different from the one in the training dataset.

From the technical point of view, the most convenient algorithms are the cascade one-sided perceptron (COS-P) and its explicitly mapped version (COS-P-Map). Both have a small memory footprint, a short training time (Table VIII), a good detection rate and few false alarms.

## V. WORKING WITH VERY LARGE DATASETS

All the results presented in this section are obtained on the large (“*scale-up*”) dataset that was described in Section II. We will present two main optimisations that we incorporated in the implementation of the one-sided perceptron (OS-P) algorithm introduced in Section III, and then we will address the problem of overfitting caused by human adnotation errors in the training datasets.

Obviously, on a very big dataset (e.g. millions of both malware and clean files), the time required for training may be a problem. Since we used the non-stochastic version of the perceptron algorithm, this problem could be easily overcome by using distributed computing on a grid system. Another solution is to modify the one-sided perceptron (OS-P) algorithm

TABLE VIII  
TIME AND MEMORY CONSUMPTION AT TRAINING.

Algorithm	Time (min)	Size (MB)
COS-P	6.25	0.1216
COS-P-Map-F1	13.5	0.732
COS-P-Map-F2	14.5	0.732
COS-P-Poly2	22	532
COS-P-Poly3	22.25	532
COS-P-Poly4	22.75	532
COS-P-Radial	2	532

### Algorithm 6 Optimised One-Sided Perceptron

---

```

NumberOfIterations ← 0
MaxIterations ← 100
repeat
  Train (R, 1, -1)
  R' = R - {all malware samples}
  while FP(R') > 0 do
    Train (R', 0, -1)
    R' = R' - {all samples correctly classified}
  end while
  NumberOfIterations ← NumberOfIterations + 1
until (TP(R) = NumberOfMalwareFiles) or
(NumberOfIterations = MaxIterations)

```

---

in such a way that it will not use all the data when it tries to minimize the number of false alarms. This can be easily achieved as follows.

One could immediately see that inside the OS-P algorithm’s loop, the second step, i.e the one where the weight vectors are modified so as to reduce the number of false alarms, only affects the weights corresponding to the clean files.

Let us assume that  $R_x = f_{a_1}, \dots, f_{a_n}$  are the feature values for a clean file, and  $w_i, i = 1, \dots, n$  is the weight vector of the OS-P algorithm (Algorithm 2 in Section III). Also, let us assume that at the iteration  $k$  of the execution of that (second) step inside the OS-P algorithm — where the number of false alarms gets reduced — we obtain such a value for  $w_i$  that  $\sum w_i f_{a_i} < 0$ . As already said, further work at this step in the OS-P algorithm affects only the clean files, which means that  $w_i$  will decrease until the number of false alarms eventually becomes 0. More exactly, at iteration  $k+1$ , the elements of the weight vector will become smaller or equal to the values they have had at the previous step. Therefore, if  $\sum w_i f_{a_i} < 0$  at iteration  $k$ , then  $\sum w_i f_{a_i}$  will be even smaller (or 0) at every subsequent iteration. That is why we no longer need to do training on that (clean) sample  $R_x$  after iteration  $k$ .

Algorithm 6 incorporates this *first main optimisation* into the OS-P algorithm. It sequentially reduces the size of the training lot and thus increases the training speed. Using this optimised version of the OS-P algorithm we obtained a speed-up factor of ten or even better (see Figure 3 and Figure 3).

Now we present the *second main optimisation*. A quite simple idea that in fact was already incorporated in the code of the one-sided perceptron algorithm (Algorithm 2 in

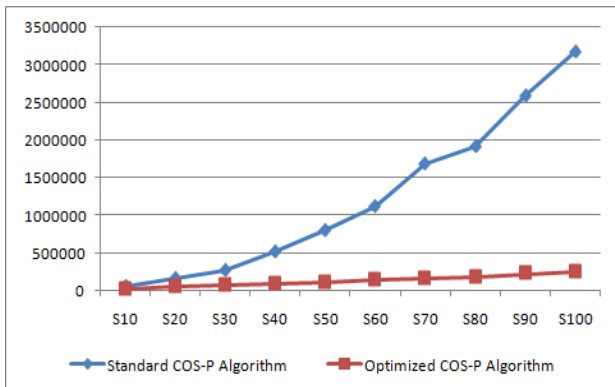


Fig. 2. Training time (measured in milliseconds) for the cascade one-sided perceptron (COS-P) algorithm, on the *scale-up* dataset.

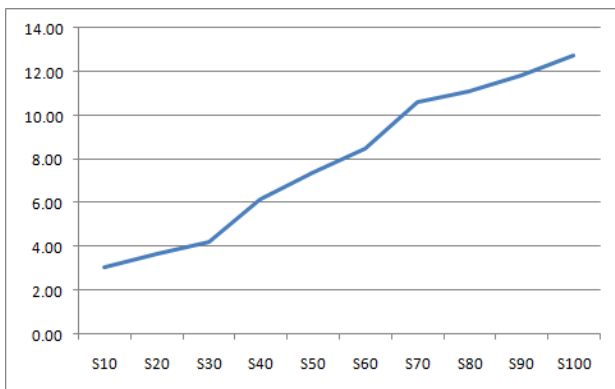


Fig. 3. Comparing the optimised COS-P algorithm with the standard COS-P algorithm, on the *scale-up* dataset. On the vertical axis: the training speed-up factor.

Section III) consists in writing the whole algorithm using 32 bit integer values rather than float values. This can be easily achieved by multiplying the threshold and the weights with a big number (usually a power of 2; we used  $2^{16}$ ). Now the learning rate will become 1 – the smallest non-zero value that can be represented using a 32 bit integer.

This simple representation issue can lead to an important optimisation in the implementation of the one-sided perceptron (OS-P) algorithm. Remember that the testing function for the perceptron is:  $\sum w_i f a_i$ , where  $w_i$ . Let us assume that  $f a_i$  has two values (0 and 0xFFFFFFFF) instead of the usual values 0 and 1. In this case we can write the testing function for the OS-P algorithm as  $\sum w_i \& f a_i$ , since it has the same as the classical testing function  $\sum w_i f a_i$ .

As shown in Table IX, at assembler level, the number of CPU ticks necessary for computing the  $\sum w_i f a_i$  sum is  $2 + 17 \times nr\_of\_features$ , while the sum  $\sum w_i \& f a_i$  requires  $2 + 7 \times nr\_of\_features$  CPU ticks. Since using the OS-P-Map algorithm implies that the number of features will be very large, it follows that the second (optimised) code will be about 2.5 times faster than the first (standard) one.

Finally, a problem that occurs when working with large datasets is *overfitting* caused by the noise appearing in the

TABLE IX  
TIME CONSUMPTION IN CPU TICKS FOR THE SUMS

$$\sum w_i f a_i \quad \text{AND} \quad \sum w_i \& f a_i.$$

Standard sum	Ticks	Optimised sum	Ticks
<code>mov ecx, 0</code>	1	<code>mov ecx, 0</code>	1
<code>mov esi, 0</code>	1	<code>mov esi, 0</code>	1
<code>sum_loop:</code>		<code>sum_loop:</code>	
<code>mov eax, features[ecx]</code>	1	<code>mov eax, features[ecx]</code>	1
<code>mov ebx, weights[ecx]</code>	1	<code>and ebx, weights[ecx]</code>	1
<code>imul ebx</code>	10	<code>add esi, eax</code>	1
<code>add esi, eax</code>	1	<code>inc ecx</code>	1
<code>inc ecx</code>	1	<code>cmp ecx, nr_of_features</code>	2
<code>cmp ecx, nr_of_features</code>	2	<code>ja sum_loop</code>	1
<code>ja sum_loop</code>	1		

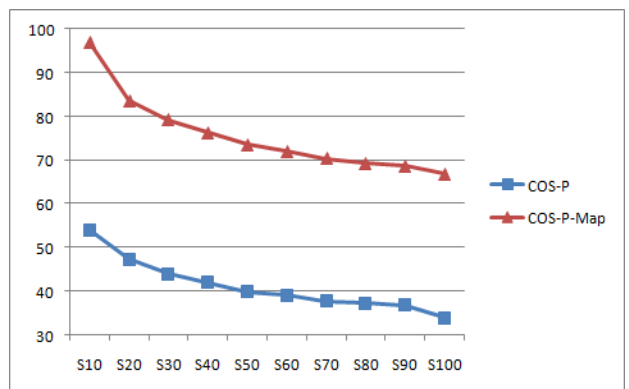


Fig. 4. Comparison of the detection rate (SE) reduction for the COS-P and COS-P-Map algorithms on the *scale-up* (large) dataset.

form of human adnotation errors. Not all of the malware designated samples are actually malware, and not all of the clean samples are clean indeed. That is why, the bigger the database, the more likely is to get misclassified samples in the training set. Because our algorithms aim to reduce the number of false alarms to 0, the detection rate (sensitivity) obtained on a large dataset will be much smaller (due to the misclassification issue). In Figure 4 we can see how the detection rate decreases when the data base gets larger. Table X shows that the accuracy, specificity and the number of false positives roughly decrease while the size of the database increases.

## VI. CONCLUSION AND FUTURE WORK

Our main target was to come up with a machine learning framework that generically detects as much malware samples as it can, with the tough constraint of having a zero false positive rate. We were very close to our goal, although we still have a non-zero false positive rate. In order that this framework to become part of a highly competitive commercial product, a number of deterministic exception mechanisms have to be added. In our opinion, malware detection via machine learning will not replace the standard detection methods used by anti-virus vendors, but will come as an addition to them. Any commercial anti-virus product is subject to certain speed and memory limitations, therefore the most reliable algorithms

TABLE X  
DETECTION RATE (SE) COMPARISON ON THE SCALE-UP (LARGE)  
DATASET WHEN TRAINING THE COS-P ALGORITHM.

Dataset	TP	FP	SE	SP	ACC
S10	170	5	51.76%	97.75%	71.94%
S20	309	5	46.94%	98.91%	69.73%
S30	438	6	44.24%	99.22%	68.32%
S40	555	6	42.13%	99.36%	67.18%
S50	648	5	39.32%	99.61%	65.72%
S60	764	5	38.66%	99.68%	65.39%
S70	842	2	36.55%	99.89%	64.29%
S80	969	2	36.82%	99.90%	64.45%
S90	1092	3	36.89%	99.87%	64.48%
S100	1100	3	33.45%	99.88%	62.56%

among those presented here are the cascade one-sided perceptron (COS-P) and its explicitly mapped variant (COS-P-Map).

Since most AntiVirus products manage to have a detection rate of over 90%, it follows that an increase of the total detection rate of 3% – 4% as the one produced by our algorithms, is very significant. (Note that the training is performed on the malware samples that are not detected by standard detection methods.)

As of this moment, our framework was proven to be a valuable research tool for the computer security experts at BitDefender AntiMalware Research Labs. For the near future we plan to integrate more classification algorithms to it, for instance large margin perceptrons [18] and Support Vector Machines [14], [19], [20].

#### ACKNOWLEDGMENTS

The authors would like to thank the management staff of BitDefender for their kind support they offered on these issues.

#### REFERENCES

- [1] I. Santos, Y. K. Peña, J. Devesa, and P. G. Garcia, “N-grams-based file signatures for malware detection,” 2009.
- [2] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, “Learning and classification of malware behavior,” in *DIMVA '08: Proceedings of the 5th international conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 108–125.
- [3] E. Konstantinou, “Metamorphic virus: Analysis and detection,” 2008, Technical Report RHUL-MA-2008-2, Search Security Award M.Sc. thesis, 93 pages.
- [4] P. K. Chan and R. Lippmann, “Machine learning for computer security,” *Journal of Machine Learning Research*, vol. 6, pp. 2669–2672, 2006.
- [5] J. Z. Kolter and M. A. Maloof, “Learning to detect and classify malicious executables in the wild,” *Journal of Machine Learning Research*, vol. 7, pp. 2721–2744, December 2006, special Issue on Machine Learning in Computer Security.
- [6] Y. Ye, D. Wang, T. Li, and D. Ye, “Imds: intelligent malware detection system,” in *KDD*, P. Berkhin, R. Caruana, and X. Wu, Eds. ACM, 2007, pp. 1043–1047.
- [7] M. Chandrasekaran, V. Vidyaraman, and S. J. Upadhyaya, “Spycon: Emulating user activities to detect evasive spyware,” in *IPCCC*. IEEE Computer Society, 2007, pp. 502–509.
- [8] M. R. Chouchane, A. Walenstein, and A. Lakhota, “Using Markov Chains to filter machine-morphed variants of malicious programs,” in *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, 2008, pp. 77–84.
- [9] M. Stamp, S. Attaluri, and S. McGhee, “Profile hidden markov models and metamorphic virus detection,” *Journal in Computer Virology*, 2008.
- [10] R. Santamarta, “Generic detection and classification of polymorphic malware using neural pattern recognition,” 2006.
- [11] I. Yoo, “Visualizing Windows executable viruses using self-organizing maps,” in *VizSEC/DMSEC '04: Proceedings of the 2004 ACM workshop on Visualization and data mining for computer security*. New York, NY, USA: ACM, 2004, pp. 82–89.
- [12] F. Rosenblatt, “The perceptron: a probabilistic model for information storage and organization in the brain,” pp. 89–114, 1988.
- [13] T. Mitchell, *Machine Learning*. McGraw-Hill Education (ISE Editions), October 1997.
- [14] N. Cristianini and J. Shawe-Taylor, *An introduction to Support Vector Machines and other kernel-based learning methods*. Cambridge University Press, March 2000.
- [15] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization and Beyond*. MIT Press, 2002.
- [16] P. Baldi, S. Brunak, Y. Chauvin, C. A. Andersen, and H. Nielsen, “Assessing the accuracy of prediction algorithms for classification,” *Bioinformatics*, no. 5, pp. 412–424, May 2000.
- [17] S. N. N. Kwang Loong and S. K. K. Mishra, “De novo SVM classification of precursor microRNAs from genomic pseudo hairpins using global and intrinsic folding measures,” *Bioinformatics*, January 2007.
- [18] Y. Freund and R. E. Schapire, “Large margin classification using the perceptron algorithm,” in *Machine Learning*, vol. 37, 1999, pp. 277–296.
- [19] C. Cortes and V. Vapnik, “Support-vector networks,” in *Machine Learning*, 1995, pp. 273–297.
- [20] V. N. Vapnik, *The Nature of Statistical Learning Theory (Information Science and Statistics)*. Springer, November 1999.