Malware Detection in PDF Files Using Machine Learning

Bonan Cuan, Aliénor Damien, Claire Delaplace, Mathieu Valois

Malware Detection in PDF Files Using Machine Learning

Bonan Cuan [*1], Aliénor Damien [†2, 3], Claire Delaplace [‡4, 5], Mathieu Valois [§6],

Under supervision of

Boussad Addad[2], Olivier Bettan[2], and Marius Lombard-Platet[2]

[1]INSA Lyon, CNRS, LIRIS, Lyon, France
[2]Thales Group, France
[3]CNRS, LAAS, Toulouse, France
[4]Univ Rennes, CNRS, IRISA, 35000 Rennes, France
[5]Univ. Lille, CNRS, Centrale Lille, UMR 9189 - CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, 59000 Lille, France
[6]Normandie Univ, UNICAEN, ENSICAEN, CNRS, GREYC, 14000 Caen, France

[*]bonan.cuan@liris.cnrs.fr
[†]alienor.damien@laas.fr
[‡]claire.delaplace@irisa.fr
[§]mathieu.valois@unicaen.fr

Abstract

In this report we present how we used machine learning techniques to detect malicious behaviours in PDF files.
At this aim, we first set up a SVM (Support Machine Vector) classifier that was able to detect 99.7% of malware. However, this classifier was easy to lure with malicious PDF, we forged to make them look like clean ones. We first proposed a very naive attack, that was easily stopped by the establishment of a threshold. We also implemented a gradient-descent attack to evade this SVM. This attack was almost 100% successful. In order to fix this problem, we provided counter-measures to the latter attack. A more elaborated features selection, and the use of a threshold, allowed us to stop up to 99.99% of these attacks.
Finally, using adversarial learning techniques, we were able to prevent gradient descent attacks by iteratively feeding the SVM with malicious forged PDF. We found that after 3 iterations, every gradient-descent forged PDF were detected, completely preventing the attack.

Keywords. Malicious PDF detection, SVM, Evasion attacks, Gradient-Descent, Feature Selections, Adversarial Learning.

Acknowledgement

This work has been accomplished during the french working session REDOCS'17. REDOCS stands for *Rencontre Entreprises DOCtorants en Sécurité*. It is a one week event, where cybersecurity PhD students are working in groups to solve real life problems proposed by industrial companies.

The authors of this work would like to thank the pre-GDR Security for organising the event. Special thanks go to Pascal Lafourcade who guided them during the whole week. The authors also thank Thales Group for having proposed this project. They particularly thank Boussad Addad, Olivier Bettan, and Marius Lombard-Platet for having supervised this work.

1 Introduction

Billions of PDF files are available on the web. Not all of them are as harmless as one may think. In fact, PDF files may contain various objects, such as JavaScript code or binary code. Sometimes, these objects may be malicious. A PDF file may then try to exploit a flaw in the reader in order to infect the machine.

In 2017, sixty-height vulnerabilities have been discovered in Adobe Acrobat Reader [CVE17]. More than fifty of them may be exploited to run arbitrary code. Every PDF reader has its own vulnerabilities, and a malicious PDF may find a way to take advantage of them.

In this context, several works proposed to use *machine learning* to detect malicious PDF files, (e.g. [Kit11, MGC12, Bor13]). These works basically rely on the same main idea: select discriminating features (i.e. features that are more likely to appear in malicious PDF files), build what is called a classifier. For a given PDF file, this classifier will take as input the selected features and, considering the number of occurence of each of them, will try to determine if the PDF file is clean or contains malware. Many approaches may be considered, both for the choice of the features and for the design of the classifier. In fact there are many classification algorithms that can be utilised: Naive Bayes, Decision Tree, Random Forest, SVM... The authors of [MGC12] first described their own way to select features, and used a Random Forest algorithm in their classifier, while the author of [Bor13] relied on the choice of features that was proposed by Didier Steven [Ste06], and used a SVM (Support Vector Machine) classifier. Both approaches seemed to provide very accurate results.

However, it is still possible to bypass such detection algorithms. Several attacks have been proposed (e.g. [AFM$^+$13, BCM$^+$13]). In [BCM$^+$13], the authors proposed to evade SVM and Neural Network classifiers using gradient-descent algorithms. Meanwhile, the authors of [AFM$^+$13] explained how to learn informations about the training dataset used by a given target classifier. To do so, they built a meta-classifier, and trained it with a set of classifiers that were themselves trained with various datasets. These datasets have different properties. Once their meta-classifier is trained, they ran it on the classifier they aim to attack. Doing so, their goal is to detect some interesting properties in the training dataset utilised by this classifier. Hopefully they can take advantage of this knowledge to attack the said classifier.

Our work. During the REDOCS week, we worked on three aspects of malware detection in PDF files.

Our first mission was to implement our own PDF classifier, using SVM algorithm, as it is known to provide good results while remaining simple. We explored different possibilities for features selection: our initial choice was based on [Ste06] selection. We refined this choice, the following way: from the set of available features, we selected those who appear to be the most discriminating in our case. We trained and tested our SVM with a dataset of 10 000 clean PDF files and 10 000 malicious PDF files from the Contagio database [Con13], and we also tuned the SVM to study its behavior. We came up with a classifier that had more than 99% of success rate, at that stage.

The second part of our work was about studying evasion techniques against our classifier. We proposed two ways to edit malicious PDF files such that they are not detected by our classifier. The first one was a quite naive attack that simply consists in highly increasing the number of occurrences of one arbitrary feature. The second one implemented a gradient-descent attack which is very similar to the one proposed by [AFM$^+$13].

Finally, we studied how to prevent these attacks. We propose here three techniques. Our first intuition was to set up a threshold value for each feature. This is fully enough to prevent the first naive attack. We also proposed a smarter features selection to make our SVM more resistant against gradient-descent attacks. Finally we suggested to update our SVM using *adversarial learning*. We found that these various methods allowed us to stop almost all gradient-descent attacks.

```
PDFiD 0.2.1 CLEAN_PDF_9000_files/rr-07-58.pdf
 PDF Header: %PDF-1.4
 obj                   23
 endobj                23
 stream                 6
 endstream              6
 xref                   2
 trailer                2
 startxref              2
 /Page                  4
 /Encrypt               0
 /ObjStm                0
 /JS                    0
 /JavaScript            0
 /AA                    0
 /OpenAction            0
 /AcroForm              0
 /JBIG2Decode           0
 /RichMedia             0
 /Launch                0
 /EmbeddedFile          0
 /XFA                   0
 /Colors > 2^24         0
```

Figure 1: Output of PDFiD

2 Malware Classifier

2.1 Useful Tools

We first recall the structure of a PDF file, and explain how this knowledge can help us to detect malware in a PDF.

2.1.1 PDF Analysis

A PDF file is composed of objects which are identified by one or several tags. These tags stand for features that characterise the PDF file. There are several tools made to analyse PDF files. In this work, we used the PDFiD Python script designed by Didier Stevens [Ste06]. Stevens also selected a list of 21 features that are commonly used in malicious PDF files. For instance the feature `/JS` indicates that a PDF file contains JavaScript and `/OpenAction` indicates that an automatic action is to be performed. It is quite suspicious to find these features in a PDF file, and sometimes, it can be a sign of a malicious behaviour. PDFiD essentially scans through a PDF file, and count the number of occurrences of each of these 21 features. It can also be utilised to count the number of occurrences of every features (not only the 21) that characterise a PDF file.

Figure 1 shows an output example of PDFiD. For each feature, the corresponding tag is given on the first column, and the number of occurrences on the second one.

We can represent these features by what we call a *feature vector*. Each coordinate represents the number of occurrences of a given feature.

2.1.2 Supervised Learning and PDF Classification

Machine Learning is a common technique to determine whether a PDF may or not contain malware. We consider a classifier function *class* that maps a feature vector to a *label* whose value is 1 if the PDF is considered as clean and -1 otherwise. To infer the *class* function, we used what is called *Supervised Learning* techniques.

We considered a dataset of 10 000 clean PDFs and 10 000 PDFs containing malware from the Contagio database [Con13]. By knowing which PDF are clean, and which are not, we were able to label them. We then split our dataset in two parts. The first part has been used as a *training dataset*. In other words, for each PDF of this dataset of feature vector x, we set $class(x)$ to 1 if the PDF was clean and $x = -1$ if it contained malware. We used a classification algorithm to infer the *class* function using this knowledge. The second part of our dataset has then been used to test if the predictions of our classifier were correct.

Usually, between 60% and 80% of the dataset is used for training. Bellow 60%, the training set may be to small, and the classifier will have poor performances when trying to infer *class*. On the other hand, if more than 80% of the dataset is used for training, the risk of overfitting the SVM increases. Overfitting is a phenomena which happens when a classifier learns noise and details specific to the training set. Furthermore, if one uses more than 80% of the dataset for training, one will have to test the classifier with less than 20% of the data, which may not be enough to provide representative results. For these reasons, we first chose to use 60% of our dataset for training, and saw how the success rate of our classifier evolved when we increased the size of the training set up to 80%.

We used a *Support Vector Machine* algorithm (SVM) as classification algorithm. Basically this algorithm considers one scatterplot per label, and finds an hyperplan (or a set of hyperplans when more than two labels are considered) to delimit them. Usually, it is unlikely that the considered set is linearly separable. For this reason, we consider the problem in a higher-dimensional space, that will hopefully make the separation easier. Furthermore, we want the dot-product in this space to be easily computed, with respect to the coordinates of the vectors of the original space. We define this new dot-product in term of *kernel function* $k(x, y)$, where x and y are two vectors of the original space. This well known trick is due to [ABR64] and was first applied to SVM in [BGV92]. It allows to work in a higher dimensional space, without having to compute the coordinates of our vectors in this space, but only dot-products, which is computationally less costly.

If we denote by $n_{features}$ the number of features we consider (i.e. the size of our vector), and we choose to use a *Gaussian Radial Basis Function* (RBF) kernel:

$$k(x, y) = \exp(-\gamma \cdot ||x - y||^2),$$

with parameter $\gamma = 1/n_{features}$.

2.2 Experimentations

We implemented our classifier in Python, using the Scikit-learn package. We initially used PDFiD with the 21 default features to create our vectors,. Then, we trained our SVM on 60% of our shuffled dataset. We used the remaining 40% data to test our SVM and calculate its accuracy: the ratio (*number of well classified PDF files*)/(*total number of PDF files*).

After having split our dataset as described above (60%/40%), we obtained an accuracy of 99.60%. Over 2622 PDF containing malware, only 29 have been detected as clean (1.11%), and over 5465 clean PDF, only 3 have been detected as containing malware (0.05%).

Using different settings. To go further in our experimentations, we slightly modify our SVM. For instance, we tested other values of parameter γ, in the RBF kernel. We also changed tested the other kernels proposed by Scikit-learn package. We figured out that using the RBF kernel with default parameter $\gamma = 1/n_{features}$ yields to the best results, considering all the settings we tried.

Change splitting ratio. We changed how we split the initial dataset into training and testing sets. We saw that, if we chose 80% of our dataset for training, and 20% for testing, then the success rate was slightly higher. We also used cross-validation: we restart our training/testing process several times with different training sets and testing sets, and combined the results we obtained. We did not notice any overfitting issue (i.e. our SVM does not seem to be affected by the noise of the training set).

Table 1: Results of features selection using: PDFiD default features (D), frequency selection on all features (F), the merge of these two lists (M), and better sublist selection (BS) applied to each of these feature set.

	Features selection	Accuracy (cross-validation)	Nb of features	Time to compute SVM
(D)	Default 21 features	99,43%	21	21,17s
(F)	Frequency (90%)	99,22%	31	54,49s
(M)	Frequency (95%) + default features	99,40%	39	47,66s
(D+BS)	Sublist from 21 default features	99,68%	11	7,03s
(F+BS)	Frequency (80%) + Sublist	99,63%	13	11,30s
(M+BS)	Frequency (80%) + default features + Sublist	99,64%	10	15,89s

Change the default features. Instead of choosing the default 21 features proposed by Didier Stevens [Ste06], we tried some other features selections. In the whole set, we found more than 100000 different types of tags, so it was too much to compute a SVM considering each tag as a feature (about 12Gb of memory required to compute the vectors for each PDF file).

A first strategy implemented was to select features by their frequency in PDF files (e.g. "90% frequency" means that 90% of the PDF in the dataset possess this feature). We chose the most common features in clean PDF in one hand, and the most frequently used features in malicious PDF in the other hand, and combined them to obtain a sublist of features. Once this selection is made, the resulting list can be merged with the 21 default features.

A second strategy, that we call "better sublist selection" was to remove non-significant features from the first sublist, by throwing features one by one and compute the SVM to check if the accuracy (computed with cross-validation) was the same or was better.

Note that these two strategies can be combined together. In practice, we did the following:

- Select initial sublist : from 21 default features and/or frequency selection,
- Apply the "better sublist selection".

The Table 1 shows some results found by applying these two strategies, regarding the original result.

Results The application of the "frequency" selection method did not improve the accuracy of our SVM, and increased significantly the number of features, making the training and testing of the SVM much slower. Applying the "better sublist" selection method improved significantly the accuracy of the SVM, and kept a reasonable amount of features. We also saw that applying the "better sublist" method to the 21 default features, improved the accuracy (+0.25%). The resulting set contains only 11 features, reducing significantly the time to train and test the SVM.

3 Evasion Attacks

In this section, we propose some evasion attacks to trick the trained SVM. We expose the goal of these attacks in subsection 3.1. Then, we define our adversary model in subsection 3.2. Finally, we present our attacks in subsection 3.3 alongside their results.

3.1 Goal

The main goal of these attacks is to modify the objects in infected PDF files to make them considered as clean by the SVM. At this end, the modifications performed on the files should not be noticeable by human eyes. Removing objects is a risky practice that may in some cases change the display of the file. On the other end, adding empty objects seems to be the easiest way to modify a PDF file, without changing its physical appearance.

3.2 Adversary Model

We consider a white box adversary. In this model, the adversary has access to everything the defender has, namely:

- The training dataset used to train the classifier
- The classifier algorithm (SVM, DecisionTree, K-Neighbors, ...)
- The classifier parameters (kernel, used features for vector, threshold, ...)
- Infected PDF files that are detected by the classifier

This attacker is the most powerful one, since she knows everything about the scheme she attacks.

3.3 Attacks

In this section we present the naive attack we implemented with its counter-measure in section 3.3.1, and the gradient descent attack in the section 3.3.2.

3.3.1 Naive Attack

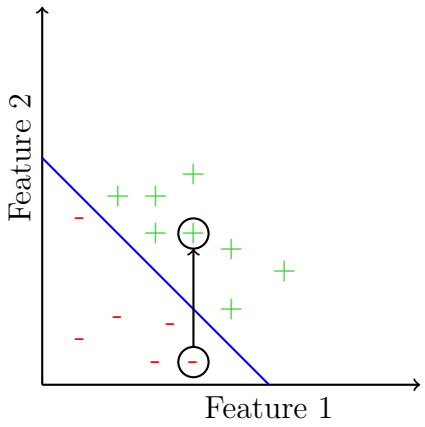

Figure 2: Example of component brutal raise attack, where feature 2 is brutally increased to cross the hyperplan

The first attack that has been implemented to lure the classifier is the component brutal raise. Given the feature vector of a PDF, the attacker picks one feature and increments it until the vector is considered as clean by the classifier. The choice of the feature is either done arbitrarily or with the same process as feature selection in section 2.2. An example of component brutal raise is illustrated by Figure 2.

Concretely, we conducted an experiment where each feature was individually incremented until all malware vectors were considered as clean. At the end of each computation, the vectors were reset to their original values, before increasing the value of the next feature. The number of additional

occurrences that are required for each feature, in order to change the label of the vector are detailed in Table 2. This gave us a more precise idea of the features to modify in order to have an effective attack.

Table 2: Number of required additional feature occurrences for all the malicious vectors to cross the hyperplan

Object	Required occurrences
obj	711
endobj	710
stream	249
endstream	246
xref	7
trailer	6
startxref	12
/Page	25
/Encrypt	2
/ObjStm	31
/JS	33
/JavaScript	37
/AA	26
/OpenAction	4
/AcroForm	5
/JBIG2Decode	3
/RichMedia	5
/Launch	2
/EmbeddedFile	19
/XFA	3
/Colors > 2 ^24	2

On this table, some features need few occurrences to achieve the attack like '/Encrypt', '/Launch', and '/Colors > 2 ^21', while some other need a huge amount of new occurrences to attack the classifier like 'obj', 'endobj', 'stream' and 'endstream'. Hence, it is more interesting for an attacker to append '/Encrypt' tags to the PDF because it requires less modification to the original file, making the changes less visible.

Counter-measure A quick counter-measure that can be applied is to ignore the surplus number of features when this number is too high, and consider it as the maximum permitted value. To implement this idea, we used a threshold value. A threshold is a value that is considered as the maximum value a feature can take. For example, if the threshold is 5, the original feature vector $x = (15, 10, 2, 3, 9, 1)$ would be cropped to become the vector $x' = (5, 5, 2, 3, 5, 1)$.

Using results from Table 2, we see that the minimum additional features required to lure the classifier is 2. Thus, we don't want an attacker to be able to add more than 1 object to a malicious file since it would be considered as clean by our classifier. We deduced then the threshold to use: it's the minimum value for the given feature in the original dataset + the number of required occurrences to cross the hyperplan - 1.

If we consider the feature '/Launch' for example, the minimum value of this feature is 0 in the malicious training set, and the required number of additionnal occurrences to lure the SVM is 2. Thus, we applied a threshold of 1 on this feature.

Because some features resulted in the same threshold, and to perform a simple counter-measure, we decided to apply the same threshold of 1 to every feature. Thus, no component could be increased

enough to bypass the classifier.

We experimented this idea with the naive attack shown in Figure 2 with the same features (the default 21 from PDFiD), and the result is that no feature can be increased enough to lure the classifier. Thus, with a threshold of 1, the naive attack can be completely blocked.

3.3.2 Gradient-Descent

The gradient-descent is a widely used algorithm in machine learning to analytically find the minimum of a given function. Among its various applications, we were particularly interested in how it can be utilised to attack SVM classifiers. Given a PDF file of feature vector x, that contains malware and has been correctly classified, the goal is to find a vector x' on the other side of the hyperplan, so that the difference between x and x' is the smallest possible. Usually, to quantify this difference, the L_1 distance is utilised. In other words, given a feature vector x such that $class(x) = -1$, we aim to find a vector x' such that $class(x') = 1$ and

$$||x - x'||_1 = \sum_i |x_i - x'_i|,$$

is minimized. The gradient-descent algorithm tries to converge to this minimum using a step by step approach: first, initialise x_0 to x, then at each step $t > 0$, x_t is computed to be equal to:

$$x^t = x^{t-1} - \epsilon_t \cdot \nabla class(x^{t-1}),$$

with ϵ_t a well chosen step size and $\nabla class(x^{t-1})$ is the gradient of $class$ at the point x^{t-1}. The iteration terminates when $class(x^t)$ is equal to 1.

An illustration of the result of this attack is shown on Figure 3 where features 1 and 2 are slightly increased to cross the hyperplan.

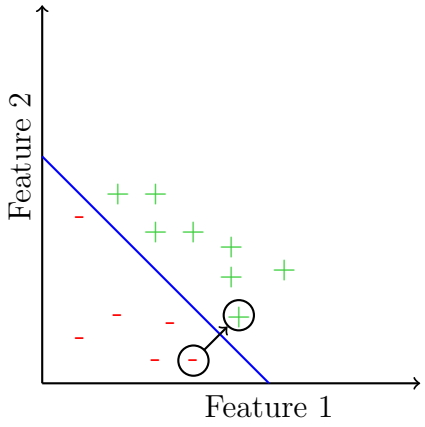

Figure 3: Example of attack using gradient-descent

This attack does not consider components individually but as a whole, allowing the algorithm to find a shorter difference vector than with the naive attack. Hence the L_1 distance between the crafted and the original vector is significantly lower than with the naive attack, it results in a crafted PDF that has been way less modified.

Using this attack we conducted two experiments: the first was to compute its theoretical success rate and the second its success rate in practice.

Theoretical success rate: to compute the theoretical success rate, we took the feature vector of every infected PDF and ran the gradient-descent to get the feature vector that lures the classifier. By this experiment, we found that 100% of the forged vectors are detected as clean by the classifier (namely every gradient-descent succeeded).

Practical success rate: to compute the practical success rate of the attack, we ran the gradient-descent on vector of every PDF, and then reconstructed new PDF according to the crafted feature vector.

Remark: If we denote by m the number of selected features considered by the SVM, the gradient-descent computes a vector $x' \in \mathbb{R}^m$, however only an integer number of objects can be added to the PDF, thus a rounding operation is needed in practice. For this attack we rounded component values to the nearest integer (the even one when tied).

Due to the rounding operation, this practical attack had a 97.5% success rate instead of the 100% of the theoretical success rate.

4 Counter-measures

The gradient-descent attack has an impressive high success rate. This is due to the huge degree of freedom the algorithm has. Every component of the vector can be increased as much as required.

Hence, to counter the gradient-descent attack, one would reduce the degrees of freedom of the algorithm. That can be achieved by three different ways: applying a threshold, following the idea of section 3.3.1, smartly select features that are the hardest to exploit, and finally a mix of both solutions by applying a threshold with a restricted set of features.

Another approach to counter this attack is to restart the training of the SVM with some malicious forged PDF: it is called *adversarial learning*.

In this section, we detail each solution by explaining the method and the choice of parameters and we give the ratio of attacks that are thus blocked.

4.1 Vector Component Threshold

Once again, the threshold is defined by $t \in \mathbb{N}^*$ due to the discreteness of PDF objects number. To choose t, we have used algorithm 1. We considered a SVM with the 21 default features of PDFiD.

Algorithm 1 Calculate the best threshold to block as many attacks as possible

```
t ← 20
s(20) ← success rate of gradient-descent with t = 20
while t > 0 do
    apply t on each forged feature vector x
    compute success rate s(t) of gradient-descent
    if s(t) > s(t + 1) then
        return t + 1
    end if
    t ← t − 1
end while
return t
```

Algorithm 1 decreases the threshold until a local minimum success rate of the gradient is found (namely when the most attacks are blocked).

Remark: Algorithm 1 assumes that the function $s(t)$ is continuous. Hence, by the intermediate value theorem, the algorithm 1 can converge. Moreover, we only retrieve a local minimum, not a global one. Despite these imprecisions, our counter-measure seemed to be rather efficient in practice (cf. Table 3).

Table 3: Percentage of Gradient-Descent attacks stopped depending on the threshold value

Threshold	Attack prevention (theory)	Accuracy of SVM
5	0%	99,55%
4	0%	99,57%
3	29%	99,63%
2	38%	99,74%
1	99,60%	99.48%

All in all, a threshold of 1 allows the SVM to block almost every attack while keeping a reasonably good accuracy (the difference between no threshold and threshold of 1 is about 0,10%).

We also applied a threshold of 1 on SVM constructed from different lists of features (see section 2.2). The corresponding results are presented in Table 4.

Table 4: Features selection global method application results

Initial features set	Attack prevention (theory)	Accuracy	Nb of features
21 default features	99,60%	99,37%	7
Frequency (80%)	91,00%	99,45%	9
Frequency (90%)	100,00%	98,00%	30
Frequency (80%) + default features	21,00%	99,61%	17
Frequency (90%) + default features	96,40%	99,46%	29

Results The use of a threshold allows to keep a very good accuracy while reducing the success rate of gradient-descent attack, but this reduction is not optimal and depends a lot of the features list utilised in the SVM.

4.2 Features Selection - Prevent Gradient-Descent

Because the gradient-descent attack uses some vulnerable features, another idea of counter-measure is to select only features that are less vulnerable to this attack. As shown by Figure 4, we selected the features that an adversary has no interest in modifying, in order to make a malicious PDF pose as a clean one. In other words, increasing the number of occurence of these features will never allows to get closer to the hyperplan we aim to cross.

To detect the vulnerable features from a list, we used the algorithm 2. This algorithm computes a SVM with the current features list and performs the gradient-descent attack on it. At each iteration, the features used by the gradient-descent attack are removed from the current features list. The algorithm stops when the features list is stable (or empty).

Global features selection method To select the final list of features, the following steps are applied :

- Select initial sublist : from 21 default features and or frequency selection (see 2.2)
- Apply Better Sublist selection (see 2.2)
- Apply gradient-descent Resistant selection (Algorithm 2)
- Apply Better Sublist selection (see 2.2)

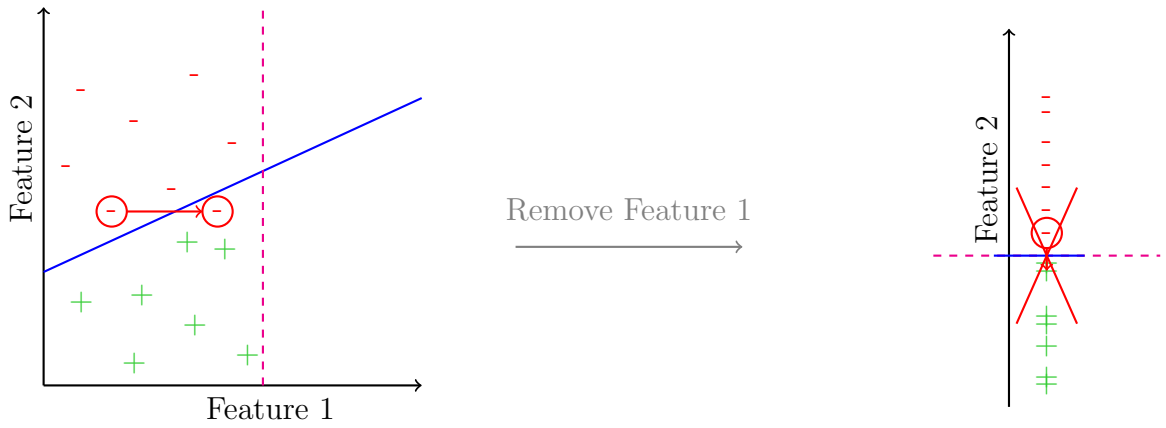

Figure 4: Suppression of features vulnerable to gradient-descent attack (Feature 1 is vulnerable here, and Feature 2 is not)

Algorithm 2 Find the features vulnerable to Gradient-Descent attack and remove them

$features\_list \leftarrow initial\_list$
$GD\_features\_used \leftarrow initial\_list$
while $GD\_features\_used \neq \perp$ **do**
  compute SVM for $features\_list$
  apply gradient-descent attack to current SVM
  $GD\_features\_used \leftarrow$ features used by gradient-descent Attack
  $features\_list.remove(GD\_features\_used)$
end while
return $features\_list$

We used different parameters here (frequency, force 21 default features or not), and the corresponding results are presented in Table 5.

Remark: accuracy is not taking into account the forged PDF, only the initial dataset of 10.000 clean PDF and 10.000 malicious PDF.

Table 5: Features selection global method application results

Initial features set	Attack prevention (theory)	Accuracy	Nb of features
21 default features (GD resistant selection only)	100%	98,03%	6
21 default features	100%	55,68%	2
Frequency (80%)	100%	67,64%	1
Frequency (90%)	100%	95,12%	3
Frequency (80%) + default features	100%	55,66%	1
Frequency (90%) + default features	100%	55,79%	3
Frequency (95%) + default features	100%	98,22%	3

Results. This features selection method drastically reduces the number of features (between 1 and 3, except the case without application of "better selection" method first), resulting in very bad accuracy results. The application of the global method on the 95% frequently used features + 21 default features only gives quite good results (98,22% of accuracy), but it is far less from the initial SVM.

4.3 Combination of Threshold & Features selection counter-measures

The threshold counter-measure previously proposed reduced significantly the attacks keeping a very good accuracy, but was not sufficient to block every gradient-descent attacks. The features selection counter-measure reduced totally the gradient-descent attack, but reduced significantly the accuracy, and had very different results depending on the initial features list. Hence, we tried to combined both counter-measures to obtain both good accuracy and total blocking of gradient-descent attack. Table 6 presents the results obtained by applying both of these techniques.

Results. We obtained in each case a theoretical block of gradient-descent attack of 100%, but only the initial features sets including the default 21 features (and the features selected by frequency or not) had a good accuracy (more than 99%).

Table 6: Features selection global method with threshold application results

Initial features set	Attack prevention (theory)	Accuracy	Nb of features
21 default features	100%	99,11%	6
Frequency (80%)	100%	94,36%	5
Frequency (90%)	100%	98,00%	30
Frequency (80%) + default features	100%	98,10%	8
Frequency (90%) + default features	100%	99,05%	6

Practical results. A summary of the best results obtained by applying each (and both) counter-measure is given by Table 7. The conclusion we can make is that the best compromise between accuracy and attack prevention is when both the threshold and the gradient-descent resistant features selection are applied. In this case, we are able to prevent 99,99% of gradient-descent attacks while conserving a reasonably good accuracy of 99,22%.

4.4 Adversarial Learning

One of the major drawbacks concerning the supervised learning that we used so far is that the SVM is only trained once. The decision is then always the same and the classifier does not learn from its mistakes.

The Adversarial Learning solves this issue by allowing the expert to feed the SVM again with vectors it wrongly classified. Hence, the classifier will learn step by step where are the ignorance regions that are used by malware to bypass the SVM.

The algorithm 3 describes how we iteratively fed our SVM to implement the adversarial learning. Note that we used the 21 default features from PDFiD.

The results of this counter-measure are presented in Table 8. With $n = 10$ we found that 3 rounds are enough for the SVM to completely stop the gradient-descent attack. Furthermore, at each round, we see that the gradient-descent algorithm requires more and more steps to finally converge,

Table 7: Results of Counter-measures "Threshold" and "Features Selection" application

	Attack prevention (in practice)	Accuracy	Nb of features
Treshold only	94,00%	99,81%	20
Features selection only	99,97%	98,05%	2 (/JS and /XFA)
Threshold + Features selection	99,99%	99,22%	9

Algorithm 3 Adversarial learning

$n \leftarrow$ number of PDFs to give to c at each iteration
$c \leftarrow$ trained SVM
$s_0 \leftarrow$ success rate of gradient-descent
$s_1 \leftarrow -1$
while $s_0 \neq s_1$ **do**
  $s_0 \leftarrow s_1$
  feed c with n gradient-descent-forged PDFs
  relaunch the learning step of c
  $s_1 \leftarrow$ success rate of gradient-descent
end while
return c

Table 8: Adversarial learning results

Round	# Support Vectors	Accuracy (%)	Steps number of GD	Success rate of GD (%)
0	293	99,70	800	100
1	308	99,68	1800	90
2	312	99,67	3000	0

and thus the attack becomes more and more costly. The Support Vectors column represents the number of support vectors the SVM has constructed. The accuracy represents the number of correct classifications of the SVM.

One interesting fact using this counter-measure is that the accuracy of the SVM barely changes. Furthermore, we did not need a more elaborated choice of features.

5 Conclusion and Perspectives

We implemented a naive SVM, that we easily tricked with a gradient-descent attack. We also implemented counter-measures against this attack: first, we set up a threshold over each considered features. This alone enabled us to stop almost every gradient-descent attack. Then, we reduced the number of selected features, in order to remove features that were used during the gradient-descent attack. This makes the attack even less practical, at the cost of reducing a bit the accuracy of the SVM. We also proposed another approach to reduce the chances of success of the gradient descent attack, using adversarial learning, by training the SVM with gradient-descent forged PDF files, and re-iterating the process over again. Our SVM was resistant to gradient-descent attack after only three iterations of the process.

Finally we would like to stress that this work as been accomplished in only four days, and obviously more can be said about the subject. For instance, it can be interesting to see what can happen if the adversary does not know the algorithms and features that have been used in the classifier (grey or black box adversary models). We could also perform a gradient-descent attack using other algorithms (e.g. Naive Bayes, Decision Tree, Random Forest) and see how many PDF files thus forged can bypass our SVM. The adversary could also use other types of attacks, like Monte-Carlo Markov Chains (MCMC) techniques. Other attacks, we did not consider, may exploit some properties, that are inherent in the training set. To avoid them, it may be interesting to have a look at unsupervised learning techniques, and try to identify a malicious behaviour with clustering.

We could also use deep learning algorithms like Generative Adversarial Network (GAN), in order to generate a classifier and test its resistance against various attacks.

References

[ABR64] M. A. Aizerman, E. A. Braverman, and L. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. Number 25 in Automation and Remote Control,, pages 821–837, 1964.

[AFM+13] Giuseppe Ateniese, Giovanni Felici, Luigi V. Mancini, Angelo Spognardi, Antonio Villani, and Domenico Vitali. Hacking smart machines with smarter ones: How to extract meaningful data from machine learning classifiers. *CoRR*, abs/1306.4447, 2013.

[BCM+13] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. *Evasion Attacks against Machine Learning at Test Time*, pages 387–402. Springer Berlin Heidelberg, 2013.

[BGV92] B. E. Boser, I. M. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, pages 144–152. ACM, 1992.

[Bor13] Knut Borg. Real time detection and analysis of pdf-files. Master's thesis, 2013.

[Con13] Contagio: Malware dump. `http://contagiodump.blogspot.fr/2013/03/16800-clean-and-11960-malicious-files.html`, 2013.

[CVE17] CVEDetails. Adobe vulnerabilities statistics. `https://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html`, 2017.

[Kit11] Jarle Kittilsen. Detecting malicious pdf documents. Master's thesis, 2011.

[MGC12] Davide Maiorca, Giorgio Giacinto, and Igino Corona. *A Pattern Recognition System for Malicious PDF Files Detection,*, pages 510–524. Springer Berlin Heidelberg, 2012.

[Ste06] D. Stevens. Didier stevens blog. `https://blog.didierstevens.com/`, 2006.