An Evasion Resilient Approach to the Detection of Malicious PDF Files

Davide Maiorca(✉), Davide Ariu, Igino Corona, and Giorgio Giacinto

University of Cagliari, Piazza d'Armi, 09123 Cagliari, Italy
{davide.maiorca,davide.ariu,igino.corona,giacinto}@diee.unica.it

Abstract. Malicious PDF files still constitute a serious threat to the systems security. New reader vulnerabilities have been discovered, and research has shown that current state of the art approaches can be easily bypassed by exploiting weaknesses caused by erroneous parsing or incomplete information extraction. In this work, we present a novel machine learning system to the detection of malicious PDF files. We have developed a static approach that leverages on information extracted by both the structure and the content of PDF files, which allows to improve the system robustness against evasion attacks. Experimental results show that our system is able to outperform all publicly available state of the art tools. We also report a significant improvement of the performances at detecting reverse mimicry attacks, which are able to completely evade systems that only extract information from the PDF file structure. Finally, we claim that, to avoid targeted attacks, a more careful design of machine learning based detectors is needed.

Keywords: PDF · Evasion · Malware · Javascript · Machine learning

1 Introduction

Malicious PDF files still constitute a major threat to computer systems, as new attacks against their readers have recently been released. The integration of the PDF file format with third-party technologies (e.g., Javascript or Flash) is often exploited to execute them. Despite the efforts of software vendors such as Adobe, PDF readers are vulnerable to zero-day attacks, as the creation of ad-hoc patches is often a complex task. Antivirus products also exhibit problems at providing protection against novel or even known attacks, due to the various code obfuscation techniques employed by most of the attacks [1].

Javascript is often adopted by attackers to exploit PDF vulnerabilities, by resorting to popular techniques such as Return Oriented Programming and Heap Spraying [2,3]. Some vulnerabilities also employed different attack vectors, such as ActionScript. For example, CVE 2010-3654 exploits a vulnerability in Adobe Flash Player by means of a "Just in Time Spraying" approach [4]. Some attacks also use advanced encryption methods for hiding malicious code or malicious embedded files [5].

© Springer International Publishing Switzerland 2015
O. Camp et al. (Eds.): ICISSP 2015, CCIS 576, pp. 68–85, 2015.
DOI: 10.1007/978-3-319-27668-7_5

Most of commercial anti-malware tools resort to signature-based approaches that are based on heuristics or string matching. However, they are often not able of detecting novel attacks, as they are inherently weak against polymorphism [6]. For this reason, recent research works analyzed malicious PDF files from two different perspectives: first, they examined malicious Javascript code within PDF files, through both static and dynamic (behavioral) analysis [7–9]. Then, they focused on the external structure of the PDF files to detect malicious ones regardless of the exploit they carried [10–12]. The latter approach is considered to be more effective than the former, as it allows to detect a wider variety of attacks, including non-Javascript ones.

However, further research proved that such strategy is extremely vulnerable against targeted attacks [13,14]. Its vulnerabilities are related to two aspects: **(a)** *File parsing*, as the logical connection among objects is often ignored and embedded contents are overlooked; **(b)** *Weak information*, i.e., data that can be easily crafted by an attacker. For this reason, new efforts have been made to provide a better detection of malicious Javascript code [15,16] and to harden security through the adoption of *sandboxes* [17].

In this work, we present a novel machine learning-based system to the detection of malicious PDF files that extracts information both from the *structure* and the *content* of the PDF file. Information on the file *structure* is obtained by examining: **(a)** basic file structure properties and **(b)** objects structural properties, in terms of *keywords*. *Content-based* information is obtained from: **(a)** malformed objects, streams and codes, **(b)** known vulnerabilities in Javascript code and **(c)** embedded contents such as other PDF files. We leverage on two well-known tools for PDF analysis, namely, `PeePDF`[1] and `Origami`[2], to provide a reliable information extraction process and to avoid parsing-related vulnerabilities.

With this approach, it is possible to accurately detect PDF malware deployed in the wild (including non-Javascript attacks), with very low false positives. At the same time, we report a significant improvement on detecting targeted attacks in comparison to the other state of the art approaches. We also show that a careful choice of the learning algorithm is crucial to ensure a correct detection of evasion attacks. We therefore encourage further research on this aspect, as we believe it can provide remarkable improvements to the security of machine learning systems. This work is an extension of a previously presented paper presented by us [18]. In this version, we provide a detailed analysis of the evasion attacks that might be perpetrated against a malicious PDF file detector, as well as a deeper insight into the solutions we have adopted to detect them.

Contributions. We summarize the contributions provided by this work in four points:

[1] http://eternal-todo.com/tools/peepdf-pdf-analysis-tool.
[2] http://esec-lab.sogeti.com/pages/origami.

- We develop a novel, machine learning based system to the detection of malicious PDF files that extracts information from the *structure* and the *content* of a PDF file;
- We experimentally evaluate the performances of our system on a dataset containing various PDF-related vulnerabilities. We compare our results to the ones obtained using publicly available tools;
- We evaluate the robustness of our system against evasion attacks that are able to completely bypass most of the released PDF files detectors;
- We discuss the limits of our system and the importance that the *learning algorithm* has to ensure a good robustness. In relation to that, we provide research guidelines for future work.

Structure. This work is divided into six Sections beyond this one. Section 2 provides the basics to understand the structure of the PDF files. Section 3 presents related works on malicious PDF detection. Section 4 describes our general methodology to the detection of malicious PDFs, and our strategies to tackle evasion attacks. Section 5 provides the experimental results. Section 6 discusses the limits of our approach and provides guidelines for future research work. Section 7 provides the conclusions of our work.

2 PDF File Format

A PDF file is a hierarchy of objects logically connected to each other. Its structure is composed by four parts [19]:

- **header:** a line that gives information on the PDF version used by the file.
- **body:** the main portion of the file, which contains all the PDF objects.
- **cross-reference table:** it indicates the position of every *indirect* object in memory.
- **trailer:** it gives relevant information about the *root object* and number of revisions made to the document. The root object is the first, in the logical hierarchy, to be parsed by the reader. New revisions (also called *versions*) are created every time the user causes changes to the PDF file. This leads to the generation of a new trailer and an updated cross-reference table, which will be appended at the end of the file.

The objects contained in the body can be of two types. *Indirect* ones are typically introduced by the expression `ObjectNumber 0 obj` and can be referenced. *Direct* objects, on the contrary, cannot be referenced and are typically less complex than the former ones. Most of indirect objects are *dictionaries* that contain a sequence of coupled *keywords* (also called *name objects*), which are introduced by a /. Keywords provide a description of the data inside the object itself or in one of its references (e.g., in case of an attack, the keyword `/Javascript` can be related to the presence of malicious code). An object might also include a *stream*, which usually contains compressed *data* that will be parsed by the reader and visualized by the user (e.g., in case of an attack, a malicious code can be

compressed into a stream that will be deployed along with the object containing the keyword `/Javascript`). For more information on the PDF structure, please check the PDF Reference [19].

3 Related Work

First approaches to malicious PDF detection resorted to static analysis on the raw (byte-level) document, by employing *n-gram* analysis [20,21] and *decision trees* [22]. However, these approaches were not focused on detecting PDF files, as they were developed to detect as many malware as possible, such as DOC and EXE based ones. Moreover, they are vulnerable to modern obfuscation techniques, such as AES encryption [5], and they can be also evaded by polymorphic malware that employ techniques like Return Oriented Programming, Heap Spraying or JIT Spraying [2–4].

Being Javascript the most popular attack vector contained in PDF files, subsequent works focused on its analysis. Many solutions have been proposed in the context of web security. For instance, `Jsand` [7], `Cujo` [23], `Zozzle` [24], `Prophiler` [25] are popular tools for the static and dynamic analysis of Javascript code. These tools are often adopted to detect threats embedded in different document formats.

`Wepawet`[3], a popular framework for the analysis of web-based threats, relies on `JSand` to analyze Javascript code within PDF files. `Jsand` [7] adopts `HtmlUnit`[4], a Java-based browser simulator, and Mozilla's `Rhino`[5] to extract dynamic behavioral features from the execution of Javascript code. The system is trained on samples containing benign code and resorts to *anomaly detection* to detect malicious files, by leveraging on the strong differences between legitimate and dangerous ones.

A similar approach is adopted by `MalOffice` [26]. `Mal Office` uses `pdftk`[6] to extract Javascript code, and `CWSandbox` [27] to analyze the code behavior: Classification is carried out by a set of rules (`CWSandbox` has also been used to classify general malware behavior [28]). `MDScan` [9] follows a different approach as malicious behavior is detected through `Nemu`, a tool able to intercept memory-injected shellcode. A different approach, with some similarities to the previous ones, has been developed in `ShellOS` [29].

Dynamic detection by executing Javascript code in a virtual environment is often time consuming and computationally expensive, and it is vulnerable to evasion when an attacker is able to exploit code parsing differences between the attacked system and the original reader [9]. To reduce computational costs, `PJScan` [8] proposed a fully static lexical analysis of Javascript code by training a statistical classifier on malicious files.

[3] http://wepawet.iseclab.org/index.php.
[4] http://htmlunit.sourceforge.net.
[5] http://www.mozilla.org/rhino.
[6] http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit.

In 2012 and 2013, malicious PDF detectors that extract information on the *structure* of the PDF file, without analyzing Javascript code, have been developed. We usually refer to them as *structural systems* [10–12]. `PDFRate`[7] is the most popular, *publicly available* approach. It is based on 202 features extracted from both document metadata and structure and it resorts to random forests to perform classification. Such approach allows to detect even non-Javascript vulnerabilities such as Actionscript based ones. Moreover, it provided significantly higher performances when compared to previous approaches. However, recent works [13,14] showed that such systems are easily attackable by exploiting, for example, parsing vulnerabilities.

As structural systems might be unreliable under targeted attacks, research focused on improving malicious Javascript code detection. New approaches resorted to discriminant API analysis [15], code instrumentation [16] and *sandboxing* [17]. Recently, a complete state of the art survey of malicious PDF files detectors has been proposed [30].

4 Proposed Detection Approach

As stated in Sect. 3, the vast majority of recent works on malicious PDF detection focused on the analysis of either the Javascript code (*content-based systems*) or the PDF file structure (*structural systems*). Such information is usually processed by a *machine learning* system, i.e., it is converted into a *vector* of numbers (*features*) and sent to a mathematical function (*classifier* or *learner*), whose parameters have been tuned through a process called *training*. Such training is performed by using samples whose classes (benign or malicious) were already known.

However, systems developed until now suffer from several weaknesses, which can be summed up in three categories:

- **Design Weaknesses:** some systems might be designed to only detect a specific type of attack (e.g., Javascript-based ones). However, such choice might make the system easy to evade when, for example, ActionScript is used [10].
- **Parsing Weaknesses:** some systems resort to what we define as *naive parsing*, i.e., analyzing the whole file content without considering its *logical* structure. This might lead to examining, for example, objects that will never be parsed by the reader. This might expose such systems to *evasion attacks*, as it is very easy to introduce changes that will deceive the systems without having any impact on the reader. Moreover, ignoring the logical structure also leads to overlooking *embedded* content, such as other PDF files [11,13].
- **Features Weaknesses:** some features might be easily crafted by an attacker. For example, a system might rely on the number of lowercase or uppercase letters of the file. Modifying such elements is a straight-forward task and might simplify the system evasion.

[7] http://pdfrate.com/.

To overcome these weaknesses, we propose a new machine learning-based approach that extracts information from the *structure* and the *content* of a PDF file. This method is purely *static* and, as the file is not executed by a PDF rendering engine.

Figure 1 shows the high-level architecture of our system. To extract information, we created a *parser* that adopts `PeePDF` and `Origami`. These tools perform an in-depth analysis of PDF files to detect known exploits, suspicious objects, or potentially malicious functions (for example, see vulnerability `CVE-2008-2992`). Moreover, they will extract and parse, as a separate sample, any *embedded* PDF file. When combined, these tools provide a reliable parsing process in comparison to other ones, such as `PdfID`, which naively analyzes PDF files ignoring their logical properties, thus allowing attackers to easily manipulate them [13].

Each PDF file will be represented by a vector composed by: **(a)** `8` features that describe the *general structure* of the file in terms of number of objects, streams, etc.; **(b)** A *variable* number of features (usually not more than `120`, depending on the training data) related to the *structure of the PDF objects*. Such features are represented by the occurrence of the most *frequent keywords* in the training dataset; **(c)** `7` features related to the *content* of the PDF objects. In particular, the PDF objects are parsed to detect *known vulnerabilities*, *malformed objects*, etc.

The remaining of this Section is organized as follows. Section 4.1 provides a detailed description of all the features that we extract to discriminate between benign and malicious PDF files. Section 4.2 describes and motivates the chosen classification algorithm. Section 4.3 describes the evasion problem and the strategies that have been adopted to counteract it.

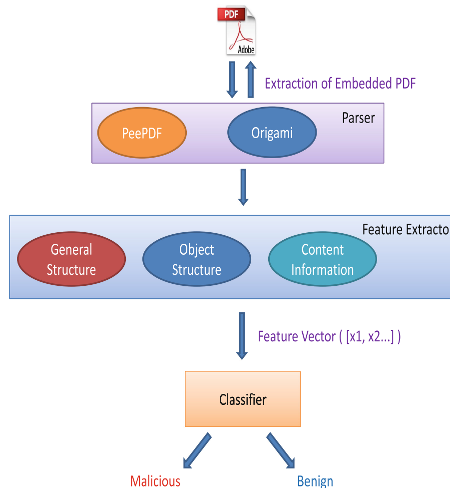

Fig. 1. High-level architecture of our system.

4.1 Features

General Structure. We extract 8 features that contain information about: **(i)** The *size* of the file; **(ii)** The number of *versions* of the file; **(iii)** The number of *indirect objects*; **(iv)** The number of *streams*; **(v)** The number of *compressed objects*; **(vi)** The number of *object streams*[8]; **(vii)** The number of *X-ref streams*[9]; **(viii)** The number of *objects containing* Javascript.

Whereas these features may not be discriminant when singularly used, they provide a good overview of the whole PDF structure when used together. For instance, malicious PDFs (and their number of objects/streams) are often smaller, in terms of size, than legitimate ones. This is reasonable, as malicious PDFs do not usually contain text. The smaller is the file size, the smaller is the time needed to infect new victims. The number of *versions* is usually higher than 1 in benign files, as a new version is typically generated when a user directly modifies or extends a PDF file. Malicious files usually exhibit a higher number of Javascript objects compared to benign files. This is because many exploits are executed by *combining* multiple Javascript pieces of code in order to generate the complete attack code. Finally, *object and X-ref streams* are usually employed to hide malicious objects inside the file, and *compressed objects* can include embedded contents, such as scripting code or other EXE/PDF files.

Object Structure. We extract the *occurrence* of the most *characteristic* keywords defined in the PDF language. *Characteristic* keywords are the ones that appeared in our training dataset D with a frequency that is higher of a threshold t. Other works, such as [12], obtained a similar threshold by arbitrarily choosing a reasonable value for it. We obtain t in a more systematic way, so that it is better related to the data in D. In order to do so, we:

1. Split D into D_m and D_l. D_m only contains *malicious* files and D_l only *legitimate files*. Obviously, $D = D_m \cup D_l$;
2. For each dataset, and for each keyword k_n of the PDF language, we define: $f_n = F(k_n)$, where f_n represents *the number of samples* of each dataset in which k_n appears at least once;
3. For each dataset, we extract the frequency threshold value t by resorting to a *k-means clustering* algorithm [31] with $k = 2$ clusters, computed through an euclidean distance. To precisely determine the sizes of the two clusters, the algorithm has been tested five times with different starting points[10]. In this way, basing on their f_n value, we split keywords into two groups. Thus, for each dataset, we extract the set of keywords K defined as: $K = \{(k_n) | f_n > t\}$. Therefore, for D_m we will obtain a set K_m and for D_l a set K_l;
4. Finally, we get the final set of characteristic keywords K_t by: $K_t = K_m \cup K_l$.

[8] Streams containing other objects.

[9] A new typology of cross-reference table introduced by recent PDF specification.

[10] The seed value has been set to the default value indicated here: http://weka.sourceforge.net/doc.dev/weka/clusterers/SimpleKMeans.html.

The number of keywords in K_t depends on the training data and on the clustering result. The reason why we considered characteristic keywords occurrences is that their presence is often related to specific actions performed by the file. For example, */Font* is a characteristic keyword in benign files. This is because it represents the presence of a specific font in the file. If this keyword occurs a lot inside one sample, it means that the PDF renderer displays different fonts, which is an expected behavior in legitimate samples. Selecting the most characteristic keywords also helps to ignore the ones that do not respect the PDF language standard. Including the occurrence of non-characteristic or extraneous keywords in the feature set might make the system vulnerable to evasion attacks, as an attacker could easily manipulate the PDF features without altering the file rendering process.

Content-Based Properties. We verify if a PDF file is accepted or rejected by either `PeePDF` or `Origami`. There are two features associated to this information, one for `PeePDF` and one for `Origami` and they are extracted by means of a *non-forced scan*[11]. Such scan evaluates the overall *integrity* of the file. For example, if the PDF file exhibits a bad or malformed header, it will be immediately rejected by the two tools. In more complex cases, rejecting a file usually means that it contains suspicious elements such as the execution of code, malformed or incorrect x-ref tables, corrupted headers, etc. However, such elements might as well be present in legitimate samples. Therefore, `PeePDF` and `Origami` cannot be used alone as malicious PDF files detectors, as they would report a lot of *false positives*.

There are also 5 features that provide information about *malformed* **(a)** objects (e.g., when scripting codes are directly put in a PDF dictionary), **(b)** streams, **(c)** actions (using keywords that do not belong to the PDF language), **(d)** code (e.g., using functions that are employed in vulnerabilities) and **(e)** compression filters (e.g., when compression is not correctly performed). This is done as malicious PDF files often contain objects with some of the aforementioned malformations, as the reader would parse them without raising any warnings about them.

4.2 Classification

We resort to a *supervised* learning approach, i.e., both benign and malicious samples are used for training, and we adopted decision trees classifiers [32]. Decision trees are capable of natively handling different types of features features, and they have successfully been used in previous works related to Malicious PDF files [10,11,15].

As classifier, we choose the *Adaptive Boosting* (`AdaBoost`) algorithm, which linearly combines a set of *weak* learners, each of them with a specific weight, to produce a more accurate classifier [33]. A *weak learner* is a low-complexity classification algorithm that is usually better than random guessing. The weights

[11] A scan that is stopped if it finds anomalies in the files. This definition is valid for `PeePDF`; in `Origami`, such scan is defined as *standard mode*.

of each weak learner are dependent on the ones of the training instances with which each learner is trained. Example of weak learners are decision stumps (i.e., decision trees with only one leaf) or simple decision trees (J48). Choosing an ensemble of trees usually guarantees more robustness against *evasion attacks* compared to a single tree, as an attacker should know which features are most discriminant for *each* tree of the ensemble to perform an optimal attack.

4.3 Evasion Detection

Introduction on Mimicry. Differently from current state of the art approaches, the features of our system, as well as its parsing mechanism, have been designed to consider the possibilities of *deliberate* attacks against structural systems. Typically, an attacker crafts a malicious file (for instance, by adding objects that will never be parsed by the reader) so that the feature values extracted by the analysis system are closer to the ones of a file that is treated as benign by the system itself. This approach is called *mimicry*.

As already observed by Biggio et al. [34], the effectiveness of the attack depends on the amount of *knowledge* possessed by the attacker. We usually distinguish between *perfect* and *imperfect* knowledge. In case of *perfect knowledge*, the attacker should be aware of the *features and the classification algorithm* employed by the system that is attacked. He should also be knowledgeable about how the features are computed. In case of *imperfect knowledge*, the attacker has incomplete information about the system features and classification algorithm. This attack is performed by means of algorithm such as *gradient descent* [34], but some simplified versions have been in other works, for example to test PDFRate [11].

Mimicry is an attack that is performed on the *feature* level. This means that first the attacker has to determine which features to modify and how many changes should be made on them. Then, he should *rebuild* the sample from the feature values he has determined. Finally, he has to ensure that the targeted system exactly extracts, from the rebuilt sample, the feature values he has obtained in the first step. Rebuilding the sample from specific feature values might be a very difficult task as some changes, although valid on the feature level, might *break* the functionalities of the file itself. For instance, keeping certain keywords is critical to assure the correct functionality of the file, and they cannot be removed. A possible solution to this problem has been proposed by Snrdic et al. [14], by limiting the changes to only *adding* features on a certain area of the file. Although effective, the authors also state that this changes might easily be detected.

Reverse Mimicry. To address the problems introduced by employing *mimicry*, an attacker can perform a variant of this attack called *reverse mimicry*, i.e., crafting a *benign* sample by injecting malicious contents in a way that its features receive as few changes as possible. To achieve this, the malicious content is injected so that the structure of the file (from which structural systems extract the file features) is only slightly changed. This has been shown to be

Fig. 2. A simplified example of the mimicry and reverse mimicry attacks. On the left (mimicry) it can be seen that the features of a malicious sample are changed to go into the benign region. On the right (reverse mimicry) a benign sample is injected with malicious content with few structural changes, so that the sample could keep staying in the benign region.

extremely effective against structural systems [13]. To better explain the differences between *mimicry* and *reverse mimicry*, Fig. 2 shows a graphical, simplified representation in a 2-D feature space of the two attacks.

Our previous work described three variants of reverse mimicry [13]: **(a)** *Java script (JS) Injection* (injecting a Javascript object that exploits a vulnerability), **(b)** *EXE Embedding* (injecting an executable that is automatically executed at runtime) and **(c)** *PDF Embedding* (injecting a malicious PDF file that is opened after the main file).

Detecting Reverse Mimicry. To tackle reverse mimicry attacks, we resort to different strategies. To counteract *PDF Embedding* we look for objects that, in their dictionary, contain the keyword `/EmbeddedFiles`. If such object is found, the relative object stream is decompressed, saved as a separate PDF and then analyzed. If this file is found to be malicious, then the original starting file will be considered malicious as well. To detect the other two attacks, it is important to correctly tune the *learning algorithm parameters* that we chose to train our system. In particular, we show that the robustness of the learning algorithm is strongly dependent on two aspects:

- The *weight threshold* (W) parameter of the `AdaBoost` algorithm (expressed, in our case, as a *percentage*) [33]. Thanks to this value, it is possible to select the samples that will be used, for each iteration of the `AdaBoost` algorithm, to tune the weights of the weak classifiers. In particular, for each iteration, the samples are chosen as follows:
  1. We order the training set samples by their *normalized* weights (the lowest weight first). Samples that have been incorrectly classified at the previous iteration get higher weights. The normalized weights sum S_w is set to zero.
  2. Starting from the first sample, we compute $S_w = S_w + w_s$, where w_s is the normalized weight of the sample. If $S_w < W$, then the sample will be employed for the training[12]. Otherwise, the algorithm stops.

[12] If W is in its percentage form, it must be divided by 100 first.

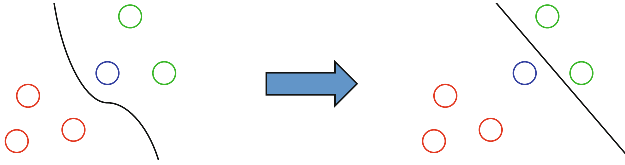

Fig. 3. A simplified example of the optimization effects on the decision function (original on the left, optimized on the right). In blue we represent a benign sample for which the classifier had to adapt its shape in order to correctly classify it (Color figure online).

The usage of a reduced weight threshold means that the weak classifiers will not be trained on samples that have been misclassified during previous iterations. This avoids that the global decision function changes its shape trying to correctly classify a particularly hard sample. This might also lead to more false positives.

– The *training data quality*. The *reverse mimicry* attacks directly address the *shape* of the classifier decision function [13], which depends on the weights of each weak classifier. Some functions might be particularly vulnerable after being trained, i.e., might have a combination of weights that could be particularly sensitive to *reverse mimicry* attacks. An empirical way to fix this problem is tuning the function weights by using *resampling*, i.e., generating *artificial training data* from the samples set obtained, given a specific weight threshold W. However, tuning the weights of an already robust function might create a *vulnerable* shape. Therefore, this empirical correction should only be used *after* having checked the weights of the function and *after* having verified its vulnerability. We call this correction *function optimization*.

Figure 3 shows a simplified example of possible performance optimizations effects. From this Figure we can observe that, when performances are optimized, the shape of the decision function will not try to adapt to the blue benign training sample. This results in a simplified decision function shape. As a further consequence, the blue sample will be misclassified. However, benign samples (in green) are now much closer to the boundary, and this will make a *reverse mimicry* attack applied on these samples most likely fail, as even with slight changes they would end up in the malicious region.

5 Experimental Evaluation

We start this Section by discussing the dataset adopted in our experiments, as well as the training and test methodology for evaluating performances. Then, we describe *two* experiments. In the first one, we compared the general performances of our approach, in terms of detection rate and false positives, to the ones of the other state of the art tools. In particular, we focused on `PJScan`, `Wepawet`, and `PDFRate`, as they can be considered the most important and *publicly available*

research tools for detecting malicious PDF files. The second experiment tested our system against the *reverse mimicry attacks* that have been described in Sect. 4.3, and compared its results to the ones provided by the tools described in the previous experiment. We do so by producing a high number of *real, working* attack samples.

Dataset. We executed our experiments using real and up-to-date samples of both benign and malicious PDFs in-the-wild. Overall, we collected `11,138` unique malicious samples from `Contagio`[13], a well-known repository that provides information about latest PDF attacks and vulnerabilities. Moreover, we *randomly* collected `9,890` benign PDF samples, by resorting to the public `Yahoo search engine API` (http://search.yahoo.com). We kept a balance between malicious and benign files to ensure a good supervised training.

For the second experiment, we created 500 attack samples variants for each of the three attacks described in Sect. 4.3: *Javascript Injection*, *EXE Embedding* and *PDF Embedding*. Hence, we generated a total of 1500 real attack samples.

Training and Test Methodology. For the *first experiment*, to carefully evaluate the performances of our system, we randomly split our data into two different datasets:

- A training set composed by `11,944` files, split into `5,993` malicious and `5,951` benign files. This set was used to train the classifier.
- A test set composed by `9,084` files, split into `5,145` malicious and `3,939` benign files. This set was used to evaluate the the classifier performances.

This process was repeated *three* times: we computed the mean and the standard deviation of the True Positives (TP) and False Positives (FP) over these three replicas. As a unique measure of the classification quality, we also employed the so-called *Matthews Correlation Coefficient* (MCC) [35], defined as:

$$MCC = \frac{TP \cdot TN - FP \cdot FN}{\sqrt{(TP + FP)(TP + FN)(TN + FP)(TN + FN)}}$$

where TN and FN refer to the number of true and false negatives, respectively.

In our experiments, we trained an `AdaBoost` [33] ensemble of J48 trees, whose parameters were optimized with a 10-fold cross validation. We selected this classifier as it showed the best accuracy compared to single classifiers (we also experimented with random forest and SVM) or other ensemble techniques on our dataset.

For the *second experiment*, we employed the *same* training sets of the first experiment to train the system but, as a test set, the `1500` attack samples described before have been adopted.

5.1 Experiment 1: General Performances

In this experiment we compared the performances of our system to three public research tools for the detection of malicious PDFs: `Wepawet`, `PJScan` and `PDFRate`

[13] http://contagiodump.blogspot.it.

Table 1. Experimental comparison between our approach and other academic tools.

System	TP(%)	FP(%)	MCC
Our System	**99.805** (±.089)	**.068** (±.054)	**.997**
PDFRate	99.380 (±.085)	.071 (±.056)	.992
Wepawet	88.921 (±.331)	.032 (±.012)	.881
PJScan	80.165 (±1.979)	.013 (±.012)	.798

(see Sect. 3). As `PJScan` employs a *One Class SVM*, we did not use any benign files to train the system. `PJScan` was trained with the same malicious samples used for our system. `PDFRate` was trained with a balanced dataset of 5000 benign and 5000 malicious samples, the latter collected from `Contagio`. We point out that there are three different instances of `PDFRate`: Each of them employs the same classifier, but is trained with different data. To provide a fair comparison with our system, we considered only the one trained on the `Contagio` dataset, as `Contagio` is the same source from which we collected our malware samples. We also observe that the training size of `Wepawet` is unfortunately unknown[14]. Even though a perfect comparison would require the same exact training set for all the systems, we believe that, in this situation, our a set up was a very good compromise with which we could provide useful information about their performances.

In Table 1 we show the results of the comparison between our system and the other tools. For each system, we show the average percentage of true positives (TP), false positives (FP), the related standard deviation within parentheses, and the MCC coefficient computed on mean values for TP and FP. We point out that `Wepawet` was not able to analyze all the samples. In particular, it examined `5,091` malicious files and `3,883` benign files. We believe there were some parsing problems that affected the system, as it did not fully implement all the Adobe specifications and only simulated the execution of embedded Javascript code and executables. We also observe that `PJScan` considered as *benign* all the samples for which it could not find evidence of *Javascript* code usable for the analysis.

From this Table, it is evident that our system completely outperformed `Wepawet` and `PJScan`. `PJScan` showed the smallest false positive rate, but exhibited a much lower detection rate compared to the other systems. `Wepawet` performed slightly better than our solution in terms of FP rate, but it provided a lower TP detection rate. We also observe than our system performed better than `PDFRate`. In fact, results are superior both in terms of TP and FP rate, with a higher MCC coefficient. We point out that our approach was better to `PDFRate` while adopting a significantly lower number of features. In fact, `PDFRate` resorts to `202` features to perform its analysis [11], whereas our system has never gone beyond `135` (considering the variable number of object-related features).

[14] Being `Wepawet` and `PDFRate` online services, we could not train such systems with our own samples.

5.2 Experiment 2: Evasion Attacks

In this experiment we produced, for each attack described in Sect. 4.3, 500 attack variants for a total of 1500 samples, as the number of samples created in our previous work was not enough for deeply assessing their efficiency against the various systems [13]. The vulnerabilities exploited in these attacks are similar to the ones presented in our previous work, with some differences[15].

Table 2 shows the performances, in terms of *true positives* (TP), of the systems tested during the previous experiment (trained with the same data and with the same *splits* as before). It can be observed that Wepawet exhibited excellent performances on *EXE Embedding* and *JS Injection*. That was expected because *reverse mimicry* addresses *static structural systems*. However, Wepawet was not able to scan *PDF Embedding* attacks due to parsing problems. As we pointed out in the previous experiment, we believe that Wepawet did not fully implement the Adobe PDF specifications, and was therefore not able of analyzing some elements of the file. PJScan also exhibited several parsing problems in this experiment and was not able of analyzing *any* of the samples we provided. This is because PJScan could not analyze embedded files, i.e., PDFs or other files such as executables, and only focused on *Javascript* analysis (which also failed, in this case). Finally, PDFRate poorly performed, thus confirming the results of our previous work [13].

With respect to our system, we notice that it was able to detect all *PDF Embedding* attacks, thanks to its advanced parsing mechanism. As shown in Table 2, using the default weight threshold, namely, $W = 100$ (the one adopted in Experiment 1) with no function optimization, we obtained performances that were already better than PDFRate, yet not fully satisfactory. With $W = 1$ and an optimized decision function, performances were almost two times better, completely outperforming all the other static approaches. Using $W = 1$ on the test data of Experiment 1, we also noticed that false positives increased up to 0.2 %. This was predictable, as explained before, as a simplified decision function shape might lead to more mistakes in the detection of benign files. It is a small trade off we had to pay for a higher robustness. The standard deviation values deserve a deeper discussion in the next section.

Table 2. Comparison, in terms of true positives (TP), between our approach and research tools with respect to *evasion* attacks (%).

System	PDF E.	EXE E.	JS INJ.
Our System (W = 1, Optimized)	**100** (±0)	**62.4** (±12.6)	**69.1** (±16.9)
Our System (W = 100)	**100** (±0)	**32.26** (±9.18)	**37.9** (±10.65)
PDFRate	0.8	0.6	5.2
Wepawet	0	99.6	100
PJScan	0	0	0

[15] For *EXE Embedding* we exploited the CVE-2010-1240 vulnerability and for *PDF Embedding* and *Javascript Injection* we exploited the CVE-2009-0927.

6 Discussion

Results attained in the second experiment showed that the features we had chosen allowed for a significantly higher robustness when compared to the state of the art. However, the high standard deviation attained in Experiment 2 also showed some limits in our approach: In this work we mainly focused on defining improving robustness by defining a more powerful set of features, but we did not *design* a *robust* decision function so that its shape would guarantee more robustness against targeted attacks. Therefore, the performances optimizations we have introduced in the previous section are only *empirical*, i.e., they are strongly dependent on the training data that are used. As future work, it would be interesting to design of a more robust *decision function* that, regardless of the quality of the training data, was able to reliably detect targeted attacks. This aspect has been often overlooked, especially in computer security applications and has been pointed out, for example, by Biggio et al. [34,36,37]. It would be also interesting to analyze the effects of *poisoning attacks* on the classifier detection, as our approach only focused on *test-time* evasion attacks [38,39]. Moreover, recent works have shown that clustering algorithms can also be vulnerable against evasion and poisoning attacks [40,41]. Since our method resorts on a clustering phase, possible future works might also address its resilience against such attacks.

7 Conclusions

Malicious PDF files have become a well-known threat in the past years. PDF documents still constitute a very effective attack vector for cyber-criminals, being their readers often vulnerable to zero-day attacks. Despite all the detection approaches that have been developed during the years, research has shown how it is possible to craft PDF samples so that it is easy for an attacker to evade even the most sophisticated detection system. In this work, we presented a new approach that leveraged on both structural and content-based information to provide a very accurate detection of PDF malware. Our approach has been designed to cope with evasion attacks, thus significantly improving the detection of reverse mimicry attacks. Finally, our work pointed out the need of secure learning techniques for malware detection, as vulnerabilities of machine learning systems seriously affect their performances at detecting targeted attacks.

Acknowledgement. This work is supported by the Regional Administration of Sardinia, Italy, within the project "Advanced and secure sharing of multimedia data over social networks in the future Internet" (CUP F71J11000690002). Davide Maiorca gratefully acknowledges Sardinia Regional Government for the financial support of his PhD scholarship (P.O.R. Sardegna F.S.E. Operational Programme of the Autonomous Region of Sardinia, European Social Fund 2007–2013 - Axis IV Human Resources, Objective l.3, Line of Activity l.3.1.).

References

1. Symantec: Internet Security Threat Reports. 2013 Trends. Symantec (2014)
2. Buchanan, E., Roemer, R., Sevage, S., Shacham, H.: Return-oriented programming: exploitation without code injection. In: Black Hat 2008 (2008)
3. Ratanaworabhan, P., Livshits, B., Zorn, B.: Nozzle: a defense against heap-spraying code injection attacks. In: Proceedings of the 18th Conference on USENIX Security Symposium (2009)
4. Bania, P.: Jit spraying and mitigations. CoRR abs/1009.1038 (2010)
5. Adobe: Adobe Supplement to ISO 32000. Adobe (2008)
6. Esparza, J.M.: Obfuscation and (non-)detection of malicious pdf files. In: S21Sec e-crime (2011)
7. Cova, M., Kruegel, C., Vigna, G.: Detection and analysis of drive-by-download attacks and malicious javascript code. In: Proceedings of the 19th International Conference on World Wide Web (2010)
8. Laskov, P., Šrndić, N.: Static detection of malicious javascript-bearing pdf documents. In: Proceedings of the 27th Annual Computer Security Applications Conference (2011)
9. Tzermias, Z., Sykiotakis, G., Polychronakis, M., Markatos, E.P.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the 4th European Workshop on System Security (2011)
10. Maiorca, D., Giacinto, G., Corona, I.: A pattern recognition system for malicious pdf files detection. In: Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition (2012)
11. Smutz, C., Stavrou, A.: Malicious pdf detection using metadata and structural features. In: Proceedings of the 28th Annual Computer Security Applications Conference (2012)
12. Šrndić, N., Laskov, P.: Detection of malicious pdf files based on hierarchical document structure. In: Proceedings of the 20th Annual Network and Distributed System Security Symposium (2013)
13. Maiorca, D., Corona, I., Giacinto, G.: Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (2013)
14. Šrndic, N., Laskov, P.: Practical evasion of a learning-based classifier: a case study. In: Proceedings of the 2014 IEEE Symposium on Security and Privacy, SP 2014, pp. 197–211. IEEE Computer Society, Washington, D.C. (2014)
15. Corona, I., Maiorca, D., Ariu, D., Giacinto, G.: Lux0r: detection of malicious pdf-embedded javascript code through discriminant analysis of API references. In: Proceedings of the 7th ACM Workshop on Artificial Intelligence and Security (AiSEC). Scottdale, Arizona, USA (2014)
16. Liu, D., Wang, H., Stavrou, A.: Detecting malicious javascript in pdf through document instrumentation. In: Proceedings of the 44th Annual International Conference on Dependable Systems and Networks (2014)
17. Maass, M., Scherlis, W.L., Aldrich, J.: In-nimbo sandboxing. In: Proceedings of the 2014 Symposium and Bootcamp on the Science of Security, HotSoS 2014. ACM, New York, pp. 1:1–1:12 (2014)
18. Maiorca, D., Ariu, D., Corona, I., Giacinto, G.: A structural and content-based approach for a precise and robust detection of malicious pdf files. In: Proceedings of the 1st International Conference on Information Systems Security and Privacy (ICISSP 2015), pp. 27–36. INSTICC (2015)

19. Adobe: PDF Reference. Adobe Portable Document Format Version 1.7. Adobe (2006)
20. Li, W.J., Stolfo, S., Stavrou, A., Androulaki, E., Keromytis, A.D.: A study of malcode-bearing documents. In: Proceedings of the 4th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (2007)
21. Shafiq, M.Z., Khayam, S.A., Farooq, M.: Embedded malware detection using markov n-grams. In: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (2008)
22. Tabish, S.M., Shafiq, M.Z., Farooq, M.: Malware detection using statistical analysis of byte-level file content. In: Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics (2009)
23. Rieck, K., Krueger, T., Dewald, A.: Cujo: efficient detection and prevention of drive-by-download attacks. In: Proceedings of the 26th Annual Computer Security Applications Conference (2010)
24. Curtsinger, C., Livshits, B., Zorn, B., Seifert, C.: Zozzle: fast and precise in-browser javascript malware detection. In: Proceedings of the 20th USENIX Conference on Security (2011)
25. Canali, D., Cova, M., Vigna, G., Kruegel, C.: Prophiler: a fast filter for the large-scale detection of malicious web pages. In: Proceedings of the 20th International Conference on World Wide Web (2011)
26. Engleberth, M., Willems, C., Holz, T.: Detecting malicious documents with combined static and dynamic analysis. In: Virus Bulletin (2009)
27. Willems, C., Holz, T., Freiling, F.: Toward automated dynamic malware analysis using cwsandbox. IEEE Secur. Priv. **5**, 32–39 (2007)
28. Rieck, K., Holz, T., Willems, C., Düssel, P., Laskov, P.: Learning and classification of malware behavior. In: Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (2008)
29. Snow, K.Z., Krishnan, S., Monrose, F., Provos, N.: Shellos: enabling fast detection and forensic analysis of code injection attacks. In: Proceedings of the 20th USENIX Conference on Security (2011)
30. Nissim, N., Cohen, A., Glezer, C., Elovici, Y.: Detection of malicious PDF files and directions for enhancements: a state-of-the art survey. Comput. Secur. **48**, 246–266 (2015)
31. MacQueen, J.B.: Some methods for classification and analysis of multivariate observations. In: Cam, L.M.L., Neyman, J., (eds.) Proceedings of the Fifth Berkeley Symposium on Mathematical Statistics and Probability, vol. 1, pp. 281–297. University of California Press (1967)
32. Quinlan, J.R.: Learning decision tree classifiers. ACM Comput. Surv. **28**, 71–72 (1996)
33. Freund, Y., Schapire, R.E.: A decision-theoretic generalization of on-line learning and an application to boosting. J. Comput. Syst. Sci. **55**(1), 119–139 (1997). doi:10.1006/jcss.1997.1504
34. Biggio, B., Corona, I., Maiorca, D., Nelson, B., Šrndić, N., Laskov, P., Giacinto, G., Roli, F.: Evasion attacks against machine learning at test time. In: Blockeel, H., Kersting, K., Nijssen, S., Železný, F. (eds.) ECML PKDD 2013, Part III. LNCS, vol. 8190, pp. 387–402. Springer, Heidelberg (2013)
35. Baldi, P., Brunak, S., Chauvin, Y., Andersen, C.A.F., Nielsen, H.: Assessing the accuracy of prediction algorithms for classification: an overview. Bioinformatics **16**, 412–424 (2000)
36. Biggio, B., Fumera, G., Roli, F.: Security evaluation of pattern classifiers under attack. IEEE Trans. Knowl. Data Eng. **26**, 984–996 (2014)

37. Biggio, B., Corona, I., Nelson, B., Rubinstein, B., Maiorca, D., Fumera, G., Giacinto, G., Roli, F.: Security evaluation of support vector machines in adversarial environments. In: Ma, Y., Guo, G. (eds.) Support Vector Machines Applications, pp. 105–153. Springer, Heidelberg (2014)
38. Biggio, B., Nelson, B., Laskov, P.: Poisoning attacks against support vector machines. In: Langford, J., Pineau, J. (eds.) 29th International Conference on Machine Learning (ICML). Omnipress (2012)
39. Biggio, B., Fumera, G., Roli, F.: Multiple classifier systems for robust classifier design in adversarial environments. Int. J. Mach. Learn. Cybernet. **1**, 27–41 (2010)
40. Biggio, B., Rieck, K., Ariu, D., Wressnegger, C., Corona, I., Giacinto, G., Roli, F.: Poisoning behavioral malware clustering. In: Proceedings of 2014 Workshop on Artificial Intelligent and Security Workshop, AISec 2014. ACM, New York, pp. 27–36 (2014)
41. Biggio, B., Pillai, I., Bulò, S.R., Ariu, D., Pelillo, M., Roli, F.: Is data clustering in adversarial settings secure? In: Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security, AISec 2013, ACM, New York, pp. 87–98 (2013)