PDF-malware detection: a survey and taxonomy of current techniques*

Michele Elingiusti[†1], Leonardo Aniello[‡2], Leonardo Querzoni[§1], and Roberto Baldoni[¶1]

[1]CIS - Sapienza University of Rome, Italy
[2]Cyber Security Research Group - University of Southampton, UK

Abstract

Portable Document Format, more commonly known as PDF, has become, in the last twenty years, a standard for document exchange and dissemination due its portable nature and widespread adoption. The flexibility and power of this format are not only leveraged by benign users, but from hackers as well who have been working to exploit various types of vulnerabilities, overcome security restrictions, and then transform the PDF format in one among the leading malicious code spread vectors. Analyzing the content of malicious PDF files to extract the main features that characterize the malware identity and behavior, is a fundamental task for modern threat intelligence platforms that need to learn how to automatically identify new attacks. This paper surveys existing state of the art about systems for the detection of malicious PDF files and organizes them in a taxonomy that separately considers the used approaches and the data analyzed to detect the presence of malicious code.

1 Introduction

Portable Document Format, commonly known as PDF, has become, since its introduction in 1993, a *de-facto* standard for document exchange and dissemination. The widespread adoption of this document format is due to both its portable nature and its inherent flexibility. PDF files, in fact, can contain a variety of media (text, pictures), but also embedded files

*This is an author generated postprint of the article: *Elingiusti M., Aniello L., Querzoni L., Baldoni R. (2018) PDF-Malware Detection: A Survey and Taxonomy of Current Techniques. In: Dehghantanha A., Conti M., Dargahi T. (eds) Cyber Threat Intelligence. Advances in Information Security, vol 70. Springer, Cham*

[†]elingiusti.1483347@studenti.uniroma1.it
[‡]l.aniello@soton.ac.uk
[§]querzoni@dis.uniroma1.it
[¶]baldoni@dis.uniroma1.it

or code that will be interpreted and executed by the reading software. This latter ability makes PDF adaptable to a large amount of extremely different usage requirements.

Despite the complexity and the number of possibilities this file format offers, end users still treat PDF files as plain, static and immutable documents, without understanding that what the reader software shows them is the result of the execution of a potentially complex program. While end users have become increasingly aware, year after year, about the traps that other document formats may hide (mainly Microsoft Office documents including macros), such awareness struggles to extend toward PDF.

In the last ten years malicious actors have exploited this lack of awareness, together with the presence of vulnerabilities in mainstream PDF readers, to make PDF become an extremely successful vector for malware diffusion. In 2010 Symantec [20] already reported a large rise in PDF-driven attacks, mainly justifying it with a corresponding rise in vulnerabilities identified in the Adobe Reader software. More recently, Ke Liu reported [8] about its discovery since December 2015 of more than 150 vulnerabilities in the most common PDF reader software products. This latter news shows how, even today, PDF is an important infection vector that provides a large unsecured attack surface.

Malware developers typically use the possibility to supply Javascript to the PDF reader interpretation engine to execute their code. Such code is usually sandboxed[1] for execution, but it may still exploit unpatched vulnerabilities to escape the environment boundaries and execute shellcode at the user level. Complex payloads can be included in the PDF as obfuscated text to evade inspection techniques, or can be downloaded from the internet as soon as the attacker takes control of the user shell. Malicious PDF files are then delivered through different methods [20]: from drive-by downloads, to targeted attacks or mass mailing approaches.

To counteract such growing phenomenon, the research community produced in the recent past several solutions for detecting malicious PDFs. The most recent and promising solutions use a mix of techniques borrowed by standard malware analysis best practices (like static and dynamic code analysis) and adapted to the specificities of the file format to extract features that are then analyzed to identify malware. The analysis is performed using several different approaches that range from simple string matching through regular expressions to complex classifiers based on machine learning techniques. In this corpus of solutions it is somewhat difficult for the interested reader to identify which approaches are adopted by a given solution and how they are related to competing solutions. Nevertheless, such solutions represent an important building block for threat intelligence platforms that need to automatically analyze incoming data looking for suspicious infection vectors or indicators of compromise. Recently, a survey from Nissim et al. [12] provided an overview of academic contributions on this area, but limited its scope to systems leveraging machine learning approaches.

[1] This feature is actually reader-dependent. As an example the Google Chrome PDF reader executed embedded Javascript code within a Google Native Client sandbox.

This work proposes a survey of the existing techniques at the state of the art for the detection of malicious PDF files. The main novelty introduced in this survey lies in its analysis of the most relevant works that tackles two orthogonal, but strictly related aspects: i) which features are considered and how they derive from the analyzed PDF file, and ii) which techniques are used to analyze such features and detect malicious files. The taxonomy is completed by a global view that considers the mix of these two aspects to correctly contextualize analyzed works and propose possible gaps that could possibly pave the way for new research initiatives. Comparing to [12], we believe that discussing features and analysis techniques as orthogonal topics, helps in shedding further light on available solutions and identifying new directions for additional research.

The text is organized as follows: after this introduction, the next Section 2 provides a basic background on the Portable Document Format and some information about obfuscation techniques that can be used to conceal malicious code in PDF files, with the aim of making the detection process less effective. Section 3 gives an overview of the study we conducted on the state of the art, describes the rationale that drives our taxonomy and details the taxonomy itself. Section 4 puts the pieces together and provides the reader with a unified view with a clear reference to existing works. Finally, Section 5 concludes the paper.

2 Background on malicious PDF files

This section introduces some basic concepts that are fundamental to understand how PDF malware detection solutions treat the internals of a PDF and how they extract features for further analysis. In addition, we briefly discuss obfuscation techniques that can be adopted by malware developers to hide malicious code with the aim of evading detection.

2.1 The Portable Document Format

The Portable Document Format is the world's leading language for describing the printed page, and the first one equally suitable for paper and online use. It is basically a file format defined in 1993 by Adobe Systems and used until today to exchange and represent documents reliably, independently from the available hardware, software, and operating system. This means that this is a format intended to display content identically in all platforms and media. In 2008 it became an open standard released as ISO 32000-1.

A PDF file may contain a mix of textual and binary data and is composed by different abstraction layers. The layers define the flow by which a PDF viewer application reads the contents in a sequence and draws them on the screen. According to the PDF Reference [3], the internal structure of a PDF file is made up of the elements depicted in Figure 1.

Metadata — All the data that can be extracted by exploring the "raw" PDF file, i.e. from its internal structure, as it is detailed below. *Metadata* includes elements such as embedded keywords, "EOF" characters located after the trailer, author field, creation data, etc.

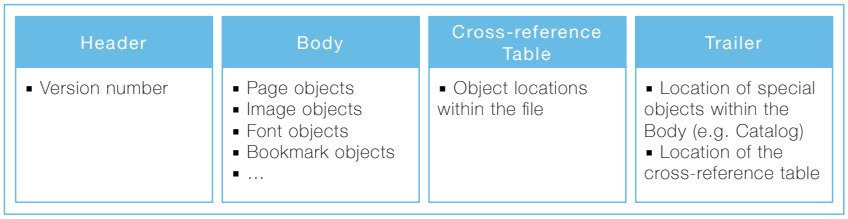

Figure 1: Internal structure of a PDF file.

Objects — The basic content of a PDF document is represented by a collection of *Objects*. Each object can contain a different element that will be used to render the file content, e.g., a page, a picture, a form, a portion of JavaScript code. Objects are the basic building blocks that collectively form the data structure of a PDF document.

An explicit definition is prefixed with a text label "`1 0 obj`". This kind of object is defined *indirect*, or also *labeled*, as it can be referenced by another object using the first number of its definition, 1 in this example, also known as its *object reference*. Conversely, *direct* objects are those that can not be referenced and do not contain any reference prefix, implying that they will always be embedded in other objects. The syntax used by a container object to refer to an indirect object follows the pattern "`1 0 R`". PDF only supports eight basic types of objects:

- **Boolean values**
- **Integer and real numbers**
- **Strings**: sequences of bytes. PDF strings have bounded length and can be represented in two distinct formats, namely as a sequence of literal characters enclosed in rounded parentheses, or as hexadecimal dump embedded in angle brackets.
- **Names**: *atomic* symbols *uniquely* defined by a character sequence.
- **Null value**: there exist only one object of type null represented by the corresponding keyword "`null`". If the null object is specified to be the value of a dictionary entry, it is like the entry did not exist. When an object references an indirect object that does not exist within the structure of the PDF, then this indirect reference is interpreted as a null object.
- **Arrays**: one-dimensional ordered collections of PDF objects enclosed in square brackets.
- **Dictionaries**: unordered sets of key-value pairs enclosed between the symbols "⟨⟨" and "⟩⟩", where each pair constitutes a dictionary's entry. Keys must be *name objects* and must be unique within a dictionary. The values may be any kind of PDF object, including nested dictionaries.

- **Streams**: sequences of bytes. Note that, while string objects must be read by a PDF viewer completely in their length, streams can undergo an incremental reading process. Furthermore, stream length is not bounded. This is the reason way large amount of data like images or JavaScript code are represented as streams.

File Structure — This layer refers to how objects are organized in a PDF file, and later accessed or updated. A PDF file structure consists of the following four parts:

- **Header**: represents the single first line of the PDF file. It has the format "`%PDF-a.b`", where `a.b` denotes the version of the PDF standard specification to which the file conforms.
- **Body**: this is the section which defines the content of the PDF document containing the objects.
- **Cross-reference table**: specifies the byte offset of every object contained in the Body starting from the top of the file.
- **Trailer**: a dictionary consisting of the "`trailer`" keyword followed by a set of key-value pairs enclosed in double angle brackets. It provides the location of the cross-reference table and of certain special objects within the body of the file, like the root object called *Catalog*. A PDF viewer conforming to the standard should read the file starting from this section in order to locate the cross-reference table and navigate to each object of the physical PDF structure. Within the trailer we can also find other relevant information like the number of revisions made to the document.

Document Structure — This layer describes the semantics of the components of the PDF file. This is a hierarchical structure that defines the relationships linking the various objects, i.e., how two object are connected. Decoupling the document structure from the file structure means that, given a document structure, it is possible to build different equivalent PDF files by simply shuffling objects order in the body. As long as the document structure does not change, the file rendering will not change as well.

At the root of the objects hierarchy there is the document's *Catalog* dictionary. A few of the nodes in the Catalog are scalar nodes, but many other nodes are the root for higher level objects. There are a lot of objects, but a minimal PDF Document will at least contain *Page* objects. Such objects are tied together in a logical structure called *page tree*, whose root is the first page object, which in turn is an indirect object referenced in the Catalog dictionary by using the entry having "`/pages`" as key.

Content Streams — These are PDF stream objects whose data consists of a sequence of instructions describing the appearance of any graphical entity that has to be rendered on a page. These objects are distinct from the basic types of data objects. The instructions can also refer to other indirect objects which contain information about resources adopted by the stream.

2.2 PDF Document Obfuscation techniques

Obfuscation is a well-known approach leveraged by malware coders to hide malicious code from inspection efforts. Code obfuscation is, in general, a legitimate technique that is widely used to protect proprietary code, however it is also one of the best evasion techniques used by malicious coders to fool malware detection systems (especially those based on signature matching) or to make the work of an expert analyst more complex and time consuming. Kittilsen listed several techniques [5] that, by mixing with the inherent complexity of the PDF format, are usually employed to hide JavaScript code in PDF files.

- **Separating Malicious Code over Multiple Objects**: the code embedded in the PDF document is fragmented among several objects and reassembled upon execution. This technique is made possible by exploiting the reference feature that is relevant to the indirect objects.
- **Applying Filters**: filters are used to compress and encode object streams of a PDF file. The parser of a detection software must be aware of the filter used, otherwise it will not even detect the presence of malicious code.
- **White Space Randomization**: randomly placed whitespace characters can be inserted in order to defeat very simple signature matching systems, like the ones based on calculating the *hash sum* of the whole document. This technique can be easily applied to JavaScript code, whose syntax is space-agnostic. Some of the solutions surveyed in the next sections (e.g. [4]) preprocess the code with a normalization phase in order to overcome this kind of obfuscation.
- **Comment Randomization**: similarly to space randomization, comments can be inserted at random to change the code without modifying its functioning.
- **Variable Name Randomization**: variable names are changed in order to overcome signature based detection systems which, for example, during a static analysis, look at the extracted code for suspicious variable names such as "`heapspray`", "`shellcode`", "`exploit`", etc.
- **String Obfuscation**: string manipulation is an obfuscation technique to fool and hinder security analysts and anti-malware software. This can be achieved in different ways. One of the most widely used techniques is to split a string in several substrings, and then merge them back at runtime.String format representation can be easily changed by employing different schemes, like the hexadecimal representation, unicode, base64 etc. An attacker can also use different formats and build hybrid representations. Another commonly used string obfuscation technique is the application of de-obfuscation functions upon strings at runtime, like substitution or XOR. Obfuscated code can be placed in any object and then deobfuscated only at runtime [13]. This kind of approach is extremely powerful against static analysis, while it is potentially subject to detection with dynamic analysis approaches.

- **Function Name Obfuscation**: this technique can be applied by creating pointers to functions using arbitrary names, like "`eval`"or "`unescape`".
- **Integer Obfuscation**: numbers can be obfuscated by representing them in a different way, like for example a mathematical expression. This is used in order to hide a specific hardcoded memory address or other kind of numbers, such as addresses related to ROP gadgets that are packaged in the code and used to exploit different versions of a reader software.
- **Block Randomization**: this involves modifying the embedded JavaScript code syntax and structure, while preserving its global behavior.
- **Dead code and Pointless code**: as a further element of obfuscation, real code can be augmented with dead code (routines that will never be executed) o pointless blocks (whose result do not impact the execution of the real malicious code).

3 Taxonomy of PDF Malware Detection Approaches

The approaches used in the state of the art to identify malicious PDF files vary widely from solution to solution. However it is possible to identify a general pattern that, with some specific variations, is commonly adopted:

- feature extraction;
- feature analysis and decision.

In the **feature extraction** phase the PDF file is analyzed to extract various features. Features can be extracted through an analysis of the PDF characteristics, or from the code that the file embeds. In this case standard static or dynamic techniques are leveraged to analyze and characterize the code behavior. Features are then analyzed in the **feature analysis** phase where several metrics of interest can be calculated. A discriminant function is then applied to decide if the input must be classified as malware or benign.

In order to conceptually organize the current state of the art in the field of PDF malware detection, we considered appropriate to apply this two-phases approach as the basis to build a taxonomy. In particular, we consider a taxonomy of the existing works with respect to two different aspects: considered *features*, and *approaches* used to analyze them. These two aspects provide orthogonal information about how the existing solutions tackle the problem of identifying malicious PDF files. In the next two sections we will detail these two taxonomies with more details.

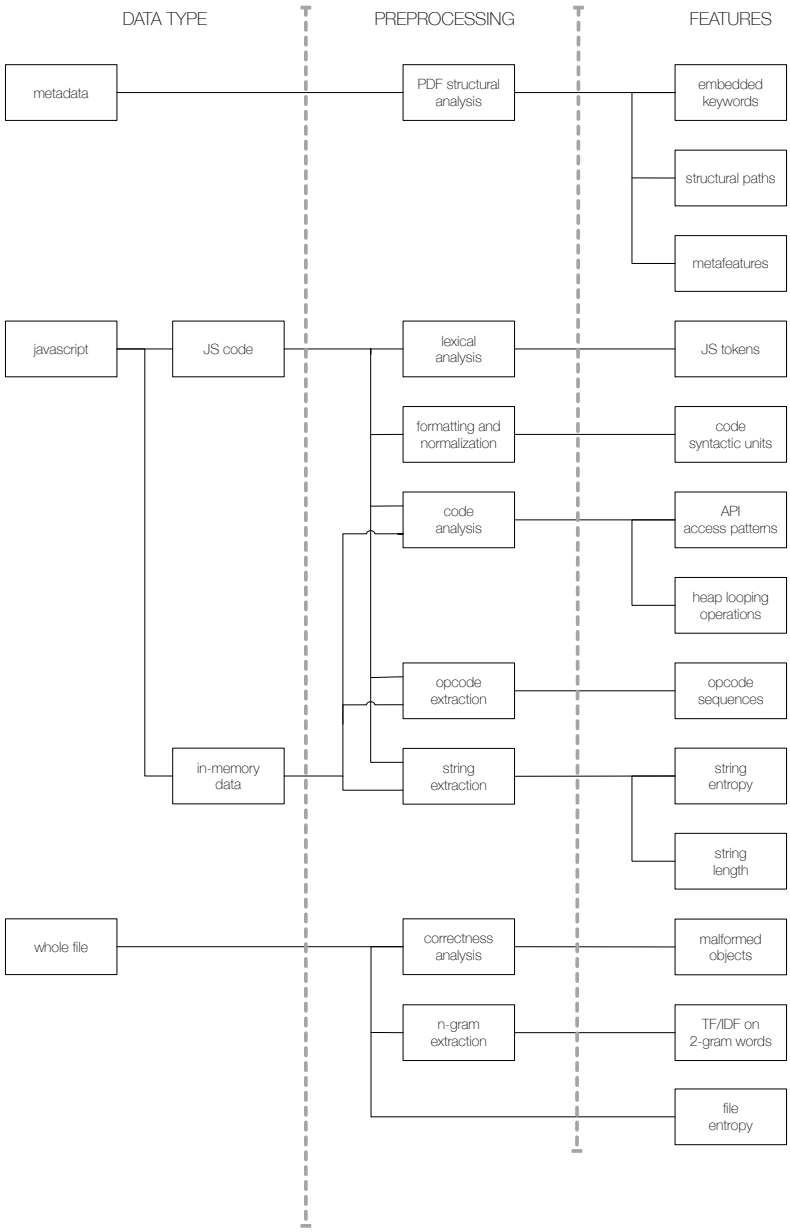

Figure 2: Taxonomy of features used in literature for PDF malware detection.

3.1 Features

This section describes which features have been proposed for PDF malware detection, and organizes them in a multi-level taxonomy (see Figure 2). The first level is the leftmost depicted in Figure 2 and represents what type of data is is extracted from the PDF document. The second level shows the preprocessing techniques used on these data to obtain the actual features used for PDF malware detection; these features are then reported in the third level.

This section is organized according to the data types identified at the first level: *metadata*, *javascript*, and *whole file*. A final subsection discusses *feature selection* techniques used by some solutions to improve detection performance.

3.1.1 Metadata

Some works focus on the *metadata* of a PDF (see Section 2.1) to determine its maliciousness [14, 11, 10, 24, 21]. They all perform *structural analyses* of the documents to extract the features they need.

Embedded keywords — A PDF reader uses the keywords embedded in the document to understand what actions to execute; therefore, the set of keywords embedded in a PDF file can be an effective indicator of its high-level behaviour. Pareek et al. [14] proposed a fixed reference set of keywords to look for in a document, while PDF Malware Slayer [11] and Slayer Neo [10] identify sets of *most characteristic keywords* by examining the occurrences of keywords in either benign and malicious PDFs included in the training set.

Structural paths — As detailed in Section 2.1, the internal structure of a PDF is organized hierarchically in a tree-like fashion. Investigating how objects are arranged in such a structure can unveil valuable clues to recognize malicious documents. Hidost [24] considers the structural paths of leaves in the analyzed documents as representing features. The obtained feature set is then processed through a technique called *structural path consolidation*(SPC) to merge together similar features. In this way the semantic of the document structure is better preserved reducing the dependency of the feature set from the specific dataset.

Metafeatures — Other works look at more general characteristics of a PDF, which we refer to as *metafeatures*. We want to stress the fact that metafeatures are different from embedded keywords. Indeed, some works [11] use a set of specific keywords as they are, simply extracting them from the structure of the file (this may imply that some keywords are closely associated to some vulnerability or malicious behavior). Conversely, metafeatures are features that somewhat reflect the properties of the metadata, like "the count of some keywords", or "the ratio of the number of pages to the size of the whole document", or "the number of uppercase characters in the author field" or other similar properties that parametrize the metadata and the file structure as much as possible. As an example, Slayer Neo [10] considers a number of statistics about the structure of a document, such as its size and the number of contained

streams. Similarly, PDFrate [21] gathers many numeric data representing aspects such as the occurrences of specific strings or the length and position of particular sections. Also Pareek [14] consider the frequency of some specific keywords, like `/js` (i.e. the number of launched javascript) or `/JavaScript` (i.e. the number of embedded javascript).

Despite systems relying on these kind of features are both efficient and effective, they are possibly subject to two kinds of evasion, namely *mimicry* and *reverse mimicry* attacks. The first attack has been demonstrated in a more systematic way by [21] and [10], and more theoretically by [6] and [23]. The second attack has been widely addressed by [10].

The peculiarity of these attacks reside in the way they prepare the malicious PDF file. In particular, the *mimicry attack* adds benign metadata-based attributes to malicious samples, while the *reverse mimicry attack* starts instead from a sample classified as benign and renders it malicious in an incremental fashion by trying to not cross the boundary line that divides the goodness and maliciousness of the file form the metafeatures standpoint.

3.1.2 JavaScript

The most common attack vector for malicious PDFs derives from embedded *JavaScript* code that can be executed by the PDF reader software. Indeed, many surveyed papers consider features derived in different ways from embedded JavaScript code [26, 7, 4, 25, 19, 9, 2]. As Figure 2 shows, features linked to JavaScript can be extracted from two distinct sources: the *JavaScript code* itself, that is actually executed when the PDF file is opened, and the *in-memory data* that is generated during code execution.

JavaScript code can be extracted from the PDF either statically or dynamically. In the former case, the code is directly extracted from the file, while in the latter the PDF is opened and parsed through a reader software to observe which code is actually executed. The dynamic approach is generally more robust against obfuscation techniques (see Section 2.2), but requires a secure sandboxed environment for execution and is, in general, more resource demanding.

In-memory data is generated by the execution of the embedded JavaScript code and can thus be observed only by running the code through dynamic analysis. Features extracted from in-memory analysis can unveil malicious activities such as the preparation of memory areas (e.g. *heap spray*) to use for buffer overflow attacks.

Different preprocessing techniques have been proposed on either JavaScript code and in-memory data to compute the required features: *lexical analysis*, *formatting and normalization*, *runtime analysis*, *code method extraction*, *opcode extraction* and *string extraction*.

Differently from previous surveys [12], our taxonomy assumes that data from which features are derived have been extracted from the analysis target with either a static or dynamic analysis process. Depending on the sample under analysis, the choice of the right tool for data extraction is either implicit in the taxonomy (i.e. in-memory data can be obtained through dynamic analysis only), or is left to the analyst (i.e. JavaScript

code extraction may be performed statically or dynamically, depending on the nature of the analyzed sample.)

Lexical Analysis — Examining possibly complex and obfuscated JavaScript codes calls for some form of abstraction to get rid of unnecessary details and isolate what is actually relevant for the detection. A lexical analysis of the code can support the automation of such an abstraction process.

Both the approaches proposed by Vatamanu et al. [26], i.e., Hierarchical Bottom-up Clustering and Hash Table Clustering, use PDF fingerprints as features, where a fingerprint is the set of pairs $\langle \text{token}, \text{frequency} \rangle$ obtained from a lexical analysis of JavaScript code extracted statically from a document. They consider *JavaScript tokens* identified using a grammar for ECMA Script.

Also PJScan [7] performs lexical analysis on JavaScript code extracted in a static way. It relies on SpiderMonkey [2] to extract JavaScript tokens, and also recognizes further tokens on the basis of their length and whether they represent invocations of suspicious functions, such as `eval()` and `unescape()`.

Formatting and Normalization — In case some kind of comparison between JavaScript code fragments is required to decide on the maliciousness of a document, a conversion to some canonical form is usually needed to enable the evaluation of possible similarities or differences.

Karademir et al. [4] use *code syntactic units* as features. As syntactic unit they consider a block or a function in JavaScript code. Code is extracted statically from a document, then it is parsed to identify syntactic units. These are extracted and encapsulated in a XML file with additional metadata, such as start and end line of each syntactic unit with respect to the original file (*formatting* phase). A later *normalization* phase includes three types of transformations aimed at abstracting the actual structure of the unit, i.e., control and assignment statements: (i) renaming of identifiers (to remove any reliance of naming conventions), (ii) filtering of undistinguishable elements, such as variable declarations, and (iii) replacing elements by their abstract name (e.g., replacing any expression with a unique symbol).

Code Analysis — The extracted JavaScript code can be either analysed or executed in a real or virtual environment, to understand in details which APIs are invoked and with which parameters. The execution environment can be instrumented to capture relevant events and information, depending on the specific desired features. PDF Scrutinizer [19] looks at runtime for operations that add elements to an array to verify whether many identical and large data blocks are inserted, which can be seen as an attempt of heap spraying. It also inspects the code statically to find any match with known signatures of malicious vulnerable method calls and parameters. Lux0R [2] uses PhoneyPDF [3] for executing both static and dynamic analyses to extract all the API references that appear in the considered JavaScript code (i.e., *API access patters*).

[2] `https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey`

[3] `https://github.com/smthmlk/phoneypdf`

Opcode Extraction — A common practice for malicious PDFs is to build the shellcode at runtime by copying the correspondent *sequence of opcodes* in some variable. Therefore, some detection approaches execute dynamic analysis to identify variables that possibly contain malicious or suspicious opcode sequences. PDF Scrutinizer [19] employs a dedicated heuristic to properly choose which values to analyze, for example by focussing on the output of `unescape` method invocations, or on strings "with length between reasonable lower and upper bounds" or having many occurrences of the pattern "`%u`". Indeed, the `unescape` method can be used to decode previously encoded strings where the malicious opcode sequence was stored, while shellcode is usually encoded using the "`%u`" pattern. MDScan [25] chooses the strings where to look for shellcode by observing that such strings are commonly built at runtime, for example by decoding or deciphering other strings. This kind of transformations requires new strings to be allocated, because strings are immutable objects in JavaScript. Hence, MDScan scans memory areas of newly allocated strings. In a similar way, also MPScan [9] identifies new strings by hooking where they are created, and subsequently examines them to spot shellcodes. Furthermore, it hooks the JavaScript engine of Adobe Reader where opcodes are actually executed, so as to reconstruct the real opcode flow.

String Extraction — Besides being analyzed to identify sequences of opcodes that may correspond to known shellcodes, strings can be also extracted for other types of analyses, for example for heap spray detection. MPScan [9] computes the entropy of strings to verify whether they can be used for heap spraying. Indeed, since a memory area to be used for heap spraying mostly contains NOP opcodes, its entropy should result relevantly lower compared to any other string. Conversely, PDF Scrutinizer [19] looks at the length of used strings; by relying on the observation that strings prepared for heap spraying are likely to have significant dimensions, it verifies whether their length is greater than 100000 bytes.

3.1.3 Whole File

In addition to inspecting document metadata or contained/executed JavaScript code, other approaches look at the PDF file *as a whole*, i.e. without considering its internal structure. The underlying rationale is that a malicious PDF holds somewhere inside specific elements designated to run some exploits and deliver a desired payload, which makes the document as a whole have some distinguishing traits overall. Thus the key idea is analyzing the entire PDF with the aim of catching any feature possibly attributable to malware. Slayer Neo [10] employs two distinct tools, PeePDF [4] and Origami [5], to parse PDF documents and observe whether any *malformed object* is found. The presence of malformed objects, streams, actions, code or filters is a valuable information to evaluate the maliciousness of a PDF. Pareek et al. [15] computes the byte-level *entropy of the entire file* to obtain a representative data to recognize

[4] `https://github.com/jesparza/peepdf`
[5] `http://esec-lab.sogeti.com/pages/origami.html`

malware. They also extract word-level *2-grams* from the hexadecimal dump of a PDF, and apply a term frequency-inverse document frequency (*TF/IDF*) analysis on the obtained 2-grams.

3.1.4 Feature selection

Feature extraction is often followed by a technique called *Feature selection* (sometimes also known as *attribute selection*). It is an automatic selection of attributes that are most relevant to the predictive modeling problem under consideration.

What is worth mentioning is that some works employ this technique with widely different approaches. PDFMS [11] and SlayerNeo [10] use a clustering approach in order to reduce the number of features to those appearing in the higher frequency cluster. Conversely, Luxor [2] uses a specifically crafted function to select a set of features, which in this case are represented by API references, that specifically characterize malicious samples. In particular, it checks the result of the function against a predefined threshold t, where t must be chosen in order to reflect a good tradeoff between classification accuracy and robustness against possible evasion. Hidost [24] performs feature selection in order to find the minimum set of features required for a successful machine learning application. Specifically the huge extracted feature set passes through a technique called *structural path consolidation* (SPC) with the aim of merging similar features. In this way the semantic of the document structure is better preserved reducing the dependency of the feature set on the specific dataset.

3.2 Detection approaches

The features extracted according to the techniques described in Section 3.1 are then used to determine whether a specific PDF is malicious or not. This section reports on the approaches used in the literature to elaborate available features for malware detection. We grouped existing approaches in four macro-classes (see Figure 3): Statistical analysis, Machine learning classification, Clustering (for family identification) and Signature matching. A subsection is dedicated for each macro-class.

3.2.1 Statistical Analysis

A common way to study a dataset of interest consists in employing well-known *statistical analysis* tools; indeed, they allow to easily find trends and relationships that otherwise would remain hidden and unexploited.

Pareek et al. [15] extract the byte-level entropy on the entire file for a set of PDFs including both benign and malicious PDFs, then calculate the confidence interval for the entropy of malevolent documents A new document to analyze is recognized as malicious if the entropy of its content is within such interval. As underlined by the authors, using the entropy only does not lead to acceptable detection accuracy.

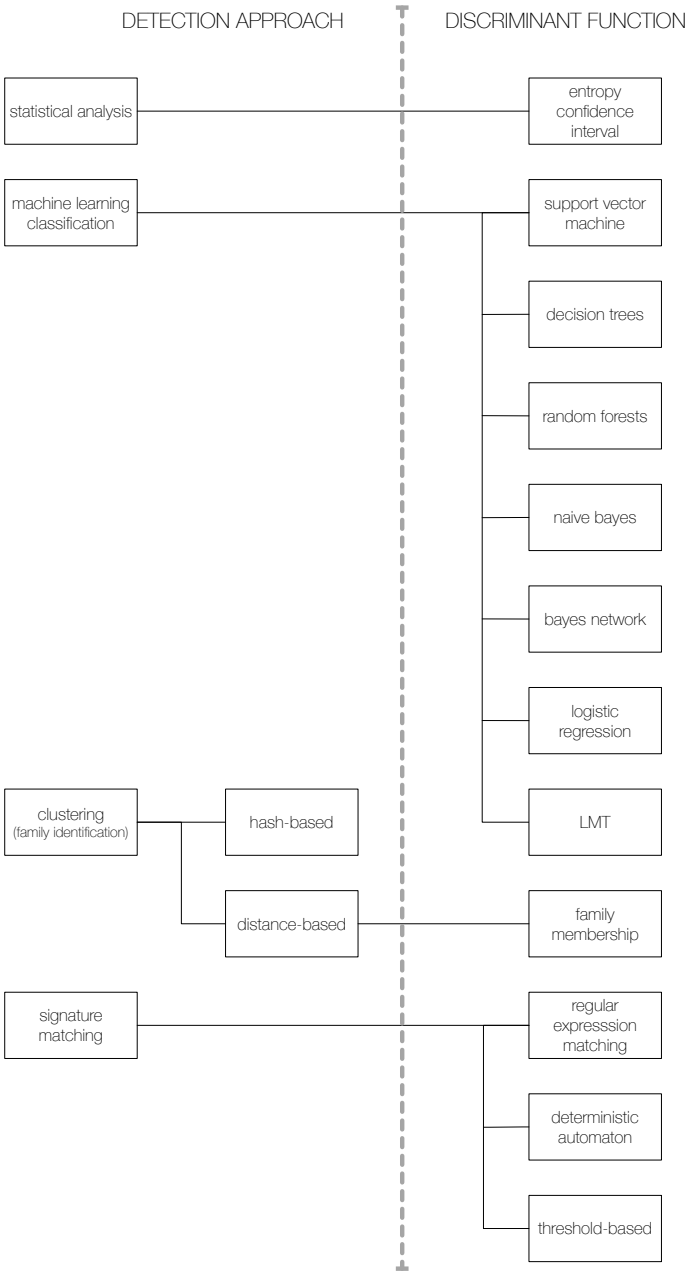

Figure 3: Taxonomy of approaches used in literature for PDF malware detection.

3.2.2 Machine Learning Classification

A natural and nowadays really widespread approach to malware detection consists in extracting a set of features from a training set, balanced between benign and malevolent samples, and training a binary classifier to detect new malicious samples with the highest possible accuracy. Several *machine learning classification* techniques are used in the literature for malware detection in PDF files. Often, reviewed papers report evaluation results on employing distinct classification algorithms and discuss which one performs best.

Two-class Support Vector Machines (SVM) are used by PDF Malware Slayer [11], PDFRate [21] and Lux0R [2]. PJScan [7] uses instead a one-class SVM, trained with a set of malicious PDFs only.

Decision Tree algorithms are the most widely employed; they are used, in fact, by Slayer Neo [10], Pareek et al. [14, 15], PDF Malware Slayer [11], Hidost [24] and Lux0R [2].

A Random Forest is an ensemble of decision trees, which usually provides better accuracy than single decision trees. PDF Malware Slayer [11], PDFRate [21] and Lux0R [2] feed their features to a Random Forest. Naive Bayes classifiers are utilized by Pareek et al. [14], PDF Malware Slayer [11] and PDFRate [21]. Pareek et al. [14] also employ other classifiers, i.e., Bayesian Networks, Logistic Regression and Logistic Model Tree (LMT).

3.2.3 Clustering

An interesting goal, within the general field of malware analysis, is grouping together samples that behave similarly or that share strong commonalities among each other. Analyzing new unknown or suspicious samples by understanding how much it behaves similarly to known malware is fundamental task to simplify the security analyst job. Indeed, this quickly gives to analysts many relevant information about analyzed samples, e.g., what actions we can expect they execute, and how to neutralize them. A group of similar malware is usually referred to as *malware family*. Given a set of malicious samples, each represented by a feature vector, it is possible to group them on the basis of the similarities they have on those features. *Clustering* algorithms are usually employed at this regard, and also some surveyed papers use them.

Vatamanu et al. [26] propose two approaches to cluster malware with the aim of understanding what families can be identified in the considered dataset of malicious PDFs. The first approach is *hash-based* and is called *Hash Table Clustering*, where for each document of the dataset the hash of the PDF fingerprint is computed, and two PDF files are considered in the same family if their hashes are in the same bucket, i.e., each bucket represents a malware family. Since this approach does not allow the detection malware (i.e. only knwon malware is categorized in families), the *hash-based* block in Figure 3 is not linked to any block of the *Discriminant Function* level of the taxonomy. The second approach is *distance-based*, the Hierarchical Bottom-up Clustering, where clusters are built iteratively in a bottom-up fashion, starting from one cluster for

each sample and then gradually merging clusters having higher similarity. Such similarity is measured using a distance metric computed on token frequencies.

Karademir et al. [4] also use a distance-based approach and compute a similarity metric between two samples by using the *NiCad* clone detection tool [18]. Each sample is represented by its code syntactic units, and, if two samples result less than 30% different from each other, then they are considered in the same family. Rather than using such clustering methodology only for family identification, they take one step further by realizing a malware detection method based on *family membership*. After a training phase where available malicious PDFs are clustered in families, when a new sample has to be analyzed, its similarity is computed with respect to identified families and, if the most similar family results less than 30% different, then the new sample is assigned to that family and hence considered as malicious.

3.2.4 Signature Matching

One among the oldest but still widely employed approaches for malware detection is *signature matching*. A knowledge base is maintained where distinguishing signatures of known malware are stored. When a new sample needs to be analyzed, it is verified against these signatures and, if any match is found, the sample is marked as malicious. We recognize three distinct classes of approaches based on signature matching: *regular expression matching*, *deterministic automation* and *threshold-based*.

Regular Expression Matching — Rather than relying on a fixed and un-flexible signature, approaches based on *regular expression matching* result more powerful and effective in identifying variants of a same malware.

PDF Scrutinizer [19] matches signatures against patterns specified with regular expressions. In particular, JavaScript code is checked for occurrences of the signatures, represented by vulnerable method calls and including parameters often used in known exploits. It also employs another kind of regular expression, consisting in the set of words more commonly used by known malicious JavaScript code, such as suspicious variable names (e.g., "`shellcode`", "`heapspray`", "`exploit`"). Vulnerable API methods calls are checked against crafted regular expression. Furthermore, it executes the code in an emulated environment and uses a basic *endless loop detection* mechanism to recognize situations where a malicious PDF realizes it is being executed in some analysis environment and reacts by not executing its malicious payload.

PDF Scrutinizer [19], MPScan [9] and MDScan [25] use external tools such as Nemu [16] and libemu to perform a pattern matching of extracted opcode sequences against signatures of known shellcodes.

Deterministic Automation — When a signature represents specific patterns of opcodes which denote known malicious activities, it can be useful to model such patterns by using finite state machines (FSM). MPScan [9] adopts this approach and relies on a knowledge base of signatures, each of them being an FSM instance modelling a malicious pattern of opcodes. To verify whether a specific opcode sequence extracted from a

PDF matches a particular signature, the correspondent FSM instance is used to check the feasibility to obtain the opcode sequence according to the allowed transitions. If the sequence can be exactly rebuilt, and the FSM instance terminates in a final state, then a matching is found and the sequence is considered malicious.

Threshold-based — A particular type of signature can consist in a threshold value, to be used to determine if a document contains malware. Its simplicity of use usually comes at the cost of limited effectiveness in terms of achievable accuracy.

PDF Scrutinizer [19] puts in place a threshold-based mechanism on string lengths. If the length of a variable string value exceeds a predefined threshold, the document is marked as malicious. This is because long strings in a malicious JavaScript code are usually instantiated for the construction of NOP-sleds, to be used in heap spray exploitations. MPScan [9] selects all the strings longer than a certain threshold, under the assumption that very long strings are likely linked to heap spray activities. The entropy of these long strings is then computed and, because heap spray mostly includes repeated characters, the result should be much lower with respect to normal and harmless strings. Hence, a maximum threshold value (1, in this case) is chosen, to determine whether the string should be considered suspicious.

4 State of the art discussion

In the previous sections we described a taxonomy that explores two aspects separately, namely features and detection approaches. For each aspect we detailed several building blocks, grouped in conceptually homogeneous families and organized in a hierarchical structure. This taxonomy helped us in clearly defining how each specific building block is considered by the works that use it. However, considering these building blocks separately does not provide the reader with a global view about how each work in the state of the art analyzes a given set of feature.

Figure 4 provides a cross-reference matrix where the two aspects of this taxonomy are represented as different axes. At the intersection of features with detection approaches within this matrix we reported references to the systems where that specific combination is used. This global view allows the reader to appreciate two details that are evident. Firstly, some works mixes the usage of different detection approaches with several distinct features. This is a common solution to improve the overall detection effectiveness of a system. The second details that is worth noticing is that several systems share similar approaches or work on the same feature sets. This is an important information as it may indicate that these features or approaches have been found to be particularly effective in detecting malware by independent researchers.

We want also to point out how Pareek et al. developed two systems, both introduced in [15], that focus on the whole file as a data extraction mean, with the difference that one is based on entropy measure while the second is n-gram-based.

			Features													
			Metadata			Javascript							Whole File			
						JS Code		JS Code or In-Memory Data								
			PDF Structural Analysis			Lexical Analysis	Formatting and Normalization	Code Analysis		Opcode Extraction	String Extraction		Correctness Analysis	N-Gram Extraction		
			Embedded Keywords	Structural Paths	Metafeatures	JS Tokens	Code Syntactinc Units	API Access Patterns	Heap Looping Operations	Opcode Sequences	String Entropy	String Length	Malformed Objects	TF/IDF on 2-Gram Words	File Entropy	
Detection Approaches	Statistical Analysys	Entropy Confidence Interval													[15]	
	Machine Learning Classification	Support Vector Machines	[11]		[21]	[7]		[2]								
		Decision Trees	[10,11,14]	[24]	[10,14]			[2]					[10]	[15]	[14]	
		Random Forests	[11]		[21]			[2]								
		Naïve Bayes	[11,14]		[14,21]										[14]	
		Bayes Networks	[14]		[14]										[14]	
		Logistic Regression	[14]		[14]										[14]	
		LMT	[14]		[14]										[14]	
	Clustering	Family Membership					[4]									
	Signature Matching	Regular Expression Matching						[19]		[9,19,25]						
		Deterministic Automaton								[9]						
		Threshold-based							[19]		[9]	[19]				

Figure 4: Cross-reference matrix

Furthermore, the two systems introduced by Vatamanu et al.[26] have not been included in the matrix as they mainly propose a method for malware family identification and do not explicitly define a discriminant function to identify benign/malicious input (they assume that all input is malicious).

The matrix shows that most detection systems take advantage of machine learning techniques for file classification. What is worth mentioning is how all of them, but [7] [10] and [15], exploit more than just one classification algorithm to train several models and select the ones providing the best accuracy on the available training datasets. By a quick visual inspection of the matrix, it seems apparent that no system has explored, so far, the power of machine learning classification techniques in combination with *OpCode sequences* as possible features vector, whereas all the systems that use them take advantage of an instrumented javascript interpreter which keeps track of runtime operations and variable values together with specific dynamic heuristics monitoring the control flow for malicious operations (e.g. shellcode detection using *GetPC* heuristics). In general, three broad groups are present in the matrix, two in the upper extremities and one on the lower middle part of the matrix. The empty parts of this matrix could possibly provide hints for future research directions.

4.1 Related works

This section briefly reports other solutions that are strictly related to the analysis performed in this survey, but don't fit adequately the proposed taxonomy. This may happen because these works focus on specific solutions that per-se do not constitute a fully fledged malware detection system. In some other cases, it is possible that the proposed work provides a fundamental building block that can be used to build a malware detection system. In any case, we think a survey like this one could not be considered complete without briefly citing these solutions.

NOZZLE — Ratanaworabhan et al. [17] presented a runtime heap-spraying detector which examines individual objects in the heap, interpreting them as code and performing a static analysis on that code to detect malicious intent. In particular the NOZZLE lightweight emulator scans heap objects to identify valid x86 code sequences, disassembling the code and building a control flow graph. This analysis technique is mainly focussed on the detection of *NOP sleds*. Through the development of an attack surface metric they try to figure out the likelihood that a random jump on an object allocated in the heap would end up executing a possible shellcode. As we know, in the heap spray technique, any jump that lands in the NOP sled will eventually transfer control to the shellcode. Through the development of a control flow graph, made of blocks with disassembled code, NOZZLE calculates the reachability of the various blocks. If one of them contains the shellcode, most likely, by jumping randomly on a different block (containing arbitrary instructions or NOP instructions), it will be eventually reached. The heap spray technique is widely employed within malicious PDF files to give exploits a higher chance of success. For this reason, blocking a part of the attack, the heap spray in this case, It

would mean stopping the attack itself.

ShellOS — Presented by Snow et al. [22], ShellOS is an open source framework that leverages hardware virtualization to better enable the detection of code injection attacks with respect to software-based emulation techniques. It is based on code analysis at runtime. The framework uses hardware virtualization to execute instruction sequences directly on the CPU, significantly improving the speed of code analysis and the execution efficiency. ShellOS kernel, runs as a guest OS using *Kernel-based Virtual MAchine* (KVM). It communicates with the host operating system by mean of shared memory address space regions, through witch it receives the stream of code to analyze and writes back the results

Active Learning Framework — Nissim et al. [13] proposed an Active Learning (AL) based framework, specifically designed to efficiently assist anti-virus vendors focussing their analytical efforts aimed at acquiring novel malicious content. The objective is to identify and acquire both new PDF files that are most likely malicious and informative benign PDF documents. These files are used for retraining and enhancing the knowledge bases of both the detection model and anti-virus. The model is built by employing a SVM classifier on the same features used by [24], namely the structural paths.

Advanced parsers — Carmony et al. [1] highlight how all existing detection techniques rely on the PDF parser to a certain extent. The problem is mainly due to the complexity of the PDF format specification. Parser implementations, built ad-hoc by anti-virus software developers are often limited in functionality, are less precise that other full-fledged parsers, and are often vulnerable to possible evasion. In order to prove that this problem is actually compelling in the field of malware detection, they implemented a *javascript reference extractor* which directly taps into Adobe Reader, and compared it with publicly available parsers, showing their inability at extracting malicious javascript code from several samples.

5 Conclusions

In this work we presented a comprehensive overview of existing solutions for PDF malware detection. We conveniently organized reviewed solutions along two orthogonal axes: one for the considered features, and one for the approach used to analyze these features to decide whether a PDF is malicious or benign. By structuring in this way the surveyed solutions, we provided a general taxonomy which can be used by practitioners to identify the best solutions for their needs. Furthermore, the same taxonomy may be of interest for researchers as it hints at clear gaps in the current state of the art that may pave the way for new interesting research directions. More in general, PDF malware analysis represents a fundamental building block for threat intelligence platforms that aim at protecting systems from diverse attacks.

Acknowledgments

This present work has been partially supported by a grant of the Italian Presidency of Ministry Council, and by CINI Cybersecurity National Laboratory within the project *FilieraSicura: Securing the Supply Chain of Domestic Critical Infrastructures from Cyber Attacks* (`www.filierasicura.it`) funded by CISCO Systems Inc. and Leonardo SpA.

References

[1] C. Carmony, M. Zhang, X. Hu, A. V. Bhaskar, and H. Yin. Extract me if you can: Abusing pdf parsers in malware detectors. 2016.

[2] I. Corona, D. Maiorca, D. Ariu, and G. Giacinto. Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references. In *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, pages 47–57. ACM, 2014.

[3] Document management – portable document format – part 1: Pdf 1.7. Standard, International Organization for Standardization, Geneva, CH, Mar. 2008.

[4] S. Karademir, T. Dean, and S. Leblanc. Using clone detection to find malware in acrobat files. In *Proceedings of the 2013 Conference of the Center for Advanced Studies on Collaborative Research*, pages 70–80. IBM Corp., 2013.

[5] J. Kittilsen. Detecting malicious pdf documents. Master's thesis, 2011.

[6] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[7] P. Laskov and N. Šrndić. Static detection of malicious javascript-bearing pdf documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 373–382. ACM, 2011.

[8] K. Liu. Dig into the attack surface of pdf and gain 100+ cves in 1 year. White paper at Black Hat Asia 2016, 2017.

[9] X. Lu, J. Zhuge, R. Wang, Y. Cao, and Y. Chen. De-obfuscation and detection of malicious pdf files with high accuracy. In *System sciences (HICSS), 2013 46th Hawaii international conference on*, pages 4890–4899. IEEE, 2013.

[10] D. Maiorca, D. Ariu, I. Corona, and G. Giacinto. 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)*, pages 27–36, 2015.

[11] D. Maiorca, G. Giacinto, and I. Corona. A Pattern Recognition System for Malicious PDF Files Detection. In P. Perner, editor, *MLDM*, volume 7376 of *Lecture Notes in Computer Science*, pages 510–524. Springer, 2012.

[12] N. Nissim, A. Cohen, C. Glezer, and Y. Elovici. Detection of malicious pdf files and directions for enhancements: a state-of-the art survey. *Computers & Security*, 48:246–266, 2015.

[13] N. Nissim, A. Cohen, R. Moskovitch, A. Shabtai, M. Edri, O. BarAd, and Y. Elovici. Keeping pace with the creation of new malicious pdf files using an active-learning based detection framework. *Security Informatics*, 5(1):1, 2016.

[14] H. Pareek, P. Eswari, and N. S. C. Babu. Malicious PDF Document Detection Based on Feature Extraction and Entropy. *International Journal Journal of Security, Privacy and Trust Management*, 2(5), 2013.

[15] H. Pareek, P. Eswari, N. S. C. Babu, and C. Bangalore. Entropy and n-gram analysis of malicious pdf documents. *International Journal of Engineering*, 2(2), 2013.

[16] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 287–296, New York, NY, USA, 2010. ACM.

[17] P. Ratanaworabhan, V. B. Livshits, and B. G. Zorn. Nozzle: A defense against heap-spraying code injection attacks. In *USENIX Security Symposium*, pages 169–186, 2009.

[18] C. K. Roy and J. R. Cordy. Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In *Proceedings of the 2008 The 16th IEEE International Conference on Program Comprehension*, ICPC '08, pages 172–181, Washington, DC, USA, 2008. IEEE Computer Society.

[19] F. Schmitt, J. Gassen, and E. Gerhards-Padilla. Pdf scrutinizer detecting javascript-based attacks in pdf documents. In *Privacy, Security and Trust (PST), 2012 Tenth Annual International Conference on*, pages 104–111. IEEE, 2012.

[20] K. Selvaraj and N. F. Gutierrez. The rise of pdf malware. *Symantec Security Response*, 2010.

[21] C. Smutz and A. Stavrou. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248. ACM, 2012.

[22] K. Z. Snow, S. Krishnan, F. Monrose, and N. Provos. Shellos: Enabling fast detection and forensic analysis of code injection attacks. In *USENIX Security Symposium*, pages 183–200, 2011.

[23] N. Šrndic and P. Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, 2013.

[24] N. Šrndić and P. Laskov. Hidost: a static machine-learning-based detector of malicious files. *EURASIP Journal on Information Security*, 2016(1):22, 2016.

[25] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.

[26] C. Vatamanu, D. Gavriluţ, and R. Benchea. A practical approach on clustering malicious pdf documents. *Journal in Computer Virology*, 8(4):151–163, 2012.