FEPDF: A Robust Feature Extractor for Malicious PDF Detection

Min Li[1], Yunzheng Liu[1, 2], Min Yu[1, 2*], Gang Li[3], Yongjian Wang[4*], Chao Liu[1]
[1]Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China
[3]School of Information Technology, Deakin University, VIC, Australia
[4]Key Laboratory of Information Network Security of Ministry of Public Security, the Third Research Institute of Ministry of Public Security, Shanghai, China
*corresponding author
yumin@iie.ac.cn, wangyongjian@stars.org.cn

***Abstract*—Due to rich characteristics and functionalities, PDF format has become the de facto standard for the electronic document exchange. As vulnerabilities in the major PDF viewers have been disclosed, a number of methods have been proposed to tame the increasing PDF threats. However, one recent evasion exploit is found to evade most of detections and renders all of the major static methods void. Moreover, many existing vulnerabilities identified before can now evade the detection through exploiting this evasion exploit. In this paper, we introduce this newly identified evasion exploit and propose a new feature extractor FEPDF to detect malicious PDFs. Based on the FEPDF and the JavaScript detection model, we test the performance of the proposed feature extractor FEPDF, and evaluation results show that FEPDF has a satisfactory performance in malicious PDF detection.**

Keywords—Malware Detection; Malicious JavaScript; PDF Documents; Code obfuscation

I. INTRODUCTION

Since the first critical Adobe Reader vulnerability was discovered in 2008, the Portable Document Format (PDF) has become one of the main attack vectors used by attackers [1]. Although PDF exploits target at various PDF readers, Adobe Reader is the most targeted one.

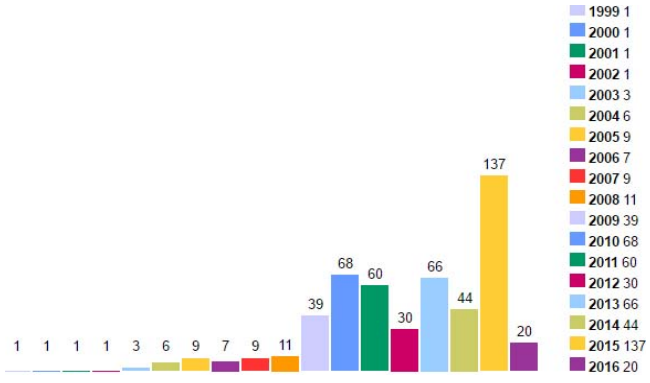

Fig. 1. Adobe Acrobat Reader's vulnerability by year.
Source: http://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53

Figure 1 shows the Adobe Acrobat Reader's vulnerabilities by year. The reason why the number of vulnerabilities decreased in 2016 is that the latest version of Adobe Acrobat Reader changed has been named as Adobe Acrobat Reader DC and its vulnerabilities were not included in the statistic. As shown in Figure 2, the number of Adobe Acrobat Reader DC's vulnerabilities increases dramatically, and it shows a peek in 2016 and still constitutes a relevant threat. In this paper, all the versions of Adobe Acrobat Reader are referred as Adobe Reader.

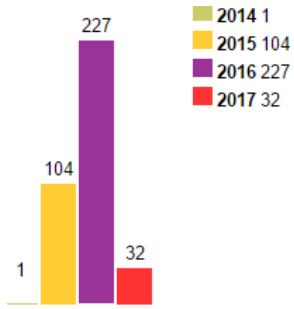

Fig. 2. Adobe Acrobat Reader DC's vulnerability by year.
Source: http://www.cvedetails.com/product/32069/Adobe-Acrobat-Reader-Dc.html?vendor_id=53

According to the Acrobat standard [2], content of many different kinds can be embedded in PDFs. Among them, JavaScript code is the most widely used, and perhaps the most threatening one [3]. To cope with malicious PDFs' increasingly security threats, a wide range of methods have been proposed [3-11]. Existing work can be categorized into two approaches: static methods and dynamic methods. The static methods have been shown to be simple, fast and accurate. Some of them focus on the PDF structure differences between the malicious and the benign documents [5-9], while others focus on the embedded JavaScript or Shellcode [3-4]. On the other hand, the dynamic methods perform well in de-obfuscation, but all of them have an overhead time expenses. Hence, some more recent techniques [10-11] attempt to extract JavaScript or Shellcode through static methods and then emulate the code execution during dynamic analysis.

However, almost all above methods extract JavaScript, Shellcode or others features using the third-party libraries designed for reading or writing PDFs. One potential problem is that libraries such as Poppler [12] strictly obey the rules of

2324-9013/17 $31.00 © 2017 IEEE
DOI 10.1109/Trustcom/BigDataSE/ICESS.2017.240

Acrobat standard, but the attackers won't. Although, methods like JSUNPACK [13] designs their own feature extractors to parse the PDFs, but those extractors are not robust enough. In this work, we will report a vulnerability that can evade all the detection methods with weak feature extractors.

Different from existing research on malicious PDFs, this work aims to design a robust feature extractor for robust detection of malicious PDFs. The extractor, named as FEPDP, will not only parse the PDF documents that follow the PDF standards [2] but also scan every suspicious segment. Experiment results show that FEPDF extracts more effective PDF features for detection. In summary, this work makes the following contributions:

- Identified an evasion exploit that can evade existing malicious PDF detection methods.
- Designed a robust and reliable feature extractor FEPDF.

The rest of this article is organized as follows. Section 2 presents the background knowledge. The vulnerability exploit we find will be introduced in section 3. System design is detailed in section 4. In section 5, we present the evaluation results. Finally, we conclude in section 6.

II. PRELIMINARIES

In this section we will introduce the key elements that constitute a PDF document, and then review the feature extractors used by the current malicious PDF detection models.

A. PDF File Format

File structure: According to the PDF references, the PDF structure contains four parts (Fig. 3):

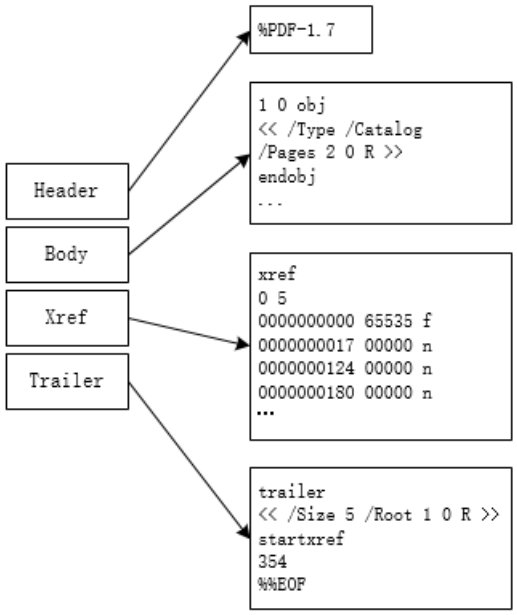

Fig.3 Structure of a PDF document

- **Header:** a line which gives the information of the version.
- **Body:** the main portion of the file, which contains all the PDF objects.
- **Cross-reference table:** indicates the position of every indirect object in memory.
- **Trailer:** enables an application to read the file for quick finding the cross-reference table and certain special objects.

Direct objects: PDF document's body consists of a hierarchy of objects that linked together to describe pages, multimedia, etc. PDFs support eight types of objects:

- **Boolean:** PDF provides Boolean objects with value **true** and **false**.
- **Numeric:** PDF provides two types of numeric object: integer and real.
- **String:** A string object consists of a series of bytes. There are two conventions, described in the following sections, for writing a string object in PDF:

  As a sequence of literal characters enclosed in parentheses ( ). For example: (So does this one.\n)

  As hexadecimal data enclosed in angle brackets < >. For example: <901FA3>
- **Name:** A name object is an atomic symbol uniquely defined by a sequence of characters.

  A slash character (/) introduces a name and indicate that the following sequence of characters constitutes a name. For example: /AB
- **Array:** An array is a one-dimensional collection of objects arranged sequentially.

  An array is written as a sequence of objects enclosed in square brackets ([and]):

  [549 3.14 false (Ralph) /SomeName]
- **Dictionary:** A dictionary is an associative table containing pairs of objects.

  It is written as a sequence of key-value pairs enclosed in double angle brackets (<< and >>):

  << /Type /Page /Version 1.0 /StringItem (a string) >>

  Dictionary objects are the main building blocks of a PDF document. They are commonly used to collect and tie together the attributes of a complex object.

  Stream: A stream object, like a string object, is a sequence of bytes. All streams must be indirect objects and the stream dictionary must be a direct object.

  Because a stream can be of unlimited length, large amounts of data, such as images and page descriptions, are represented as streams. It consists of a dictionary that describes a sequence of bytes bracketed with the keywords **stream** and **endstream**:

stream

this is the stream data

this is also the stream data and it can be more lines

endstream

- **Null:** The null object is used to fill the empty or uninitialized positions in an array or dictionary.

Indirect objects: Objects may be labeled so that they can be referred to by other objects. A labeled object is called an indirect object. The object identifier consists of two parts:

- **Object number:** A positive integer.
- **Generation number:** A nonnegative integer.

The definition of an indirect object in a PDF file consists of its object number and generation number, followed by the value of the object itself bracketed between the keywords **obj** and **endobj**. For example:

2 0 obj

<< /Type /Catalog /Pages 2 0 R>>

endobj

PDF Contains JS Code: According to the PDF reference, different kinds of content can be embedded in PDFs, and JavaScript code is the most widely used. Figure 4 shows a simple PDF document embedded with JavaScript. The sequence number shows the steps of how JavaScript code is executed by the reader applications.

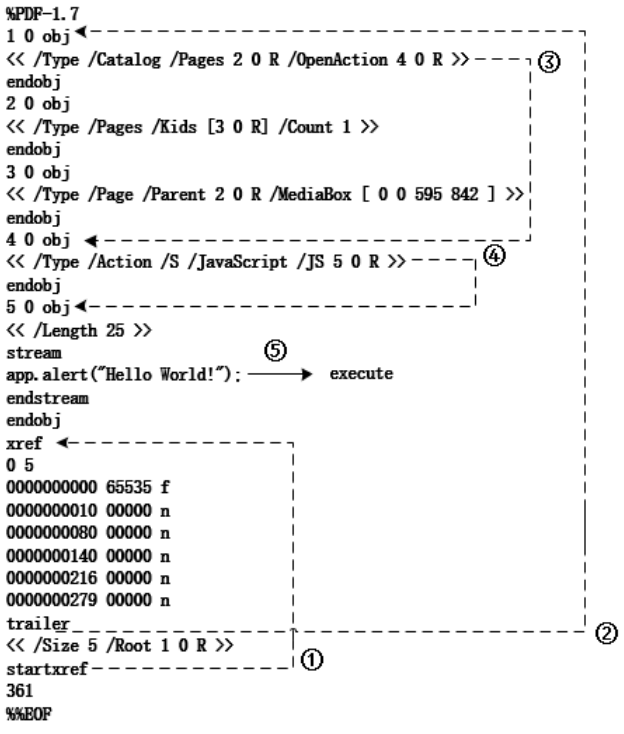

Fig.4 A simple PDF embed with JavaScript code

B. The Current Feature Extractors

Poppler: Poppler is a PDF rendering library based on the xpdf-3.0 code base. It is used by Hidost [9], a malicious document detection model focusing on the structure features of the Document. This model's feature extractor combines Poppler with regular expression to parse the PDF documents and to extract the structure features.

iTex: iText is a software develop toolkit that allows users to integrate PDF functionalities within their applications, processes or products [14]. It is also used to extract features from PDFs.

JSUNPACK: Strictly speaking, JSUNPACK is not a feature extractor. It is a generic JavaScript Unpacker which can detect the malicious PDFs by extracting and analyzing the JavaScript code. Before extracting the JavaScript code, it will scan and record all the objects using regular expressions.

PdfJsExtractor: This feature extractor is based on Poppler and designed for PJScan [1], a malicious PDF detection model which focus on the JavaScript code embedded in the PDF documents. Before extracting the JavaScript code, the extractor will check the **catalog** and the **xref** at first. According to the PDF references, it needs **xref** to search all the indirect objects and after that it will be able to parse the whole PDF documents and extract the JavaScript code.

III. THE VULNERABILITY EXPLOIT

In this section, we will present the vulnerability and analyze the vulnerability exploit to evade the current detection methods in detail.

One of the biggest issues in the current PDF readers is that they accept flawed files and try to correct the errors by themselves [15]. Although the PDF references recommend many rules, many PDF readers including the Adobe Reader do not obey the rules strictly. Figure 4 has shown a normal PDF file embedded with JavaScript code. If the JavaScript code is malicious, the current detection method can extract and detect it.

But attackers can easily evade most detection models by replacing the keyword **endobj** with any non-empty strings like **xxxxxx**. Adobe Reader can also execute the JavaScript code, but most existing feature extractors fail to extract them.

According to the PDF reference, the root of a document's object hierarchy is the **catalog** dictionary, located via the **Root** entry in the **trailer** of the PDF file. The **catalog** contains references to other objects defining the document's content, outline, article threads, named destinations, and other attributes. One finding is that we can copy the root object with the keyword **catalog** and make some modifications. When inserting the modified root object into the PDF, Adobe Reader will recognize it as the root object and ignore the original one. The precondition is that, the modified root object should be inserted after the original root object and before the Cross-reference table. The modified PDF looks normally, with no warning when opened by Adobe Reader.

In addition, the Cross-reference table that indicates the offset of the indirect objects is allowed to be damaged or missing. Hence, the offset of the indirect object can be incorrect. This means that we can add some objects into the PDF more easily,

because the **xref** can be deleted or unmodified. Some existing feature extractors reviewed above will lose the ability to parse the PDFs because of the mistakes or the missing of the Cross-reference table.

Accordingly, an exploit template can be used to evade the current detection models. As figure 5 shows, the red component is a simple instance of this template. When the modified PDF is opened by the Adobe Reader, JavaScript code will be executed automated. But all existing feature extractors mentioned above cannot extract the JavaScript code.

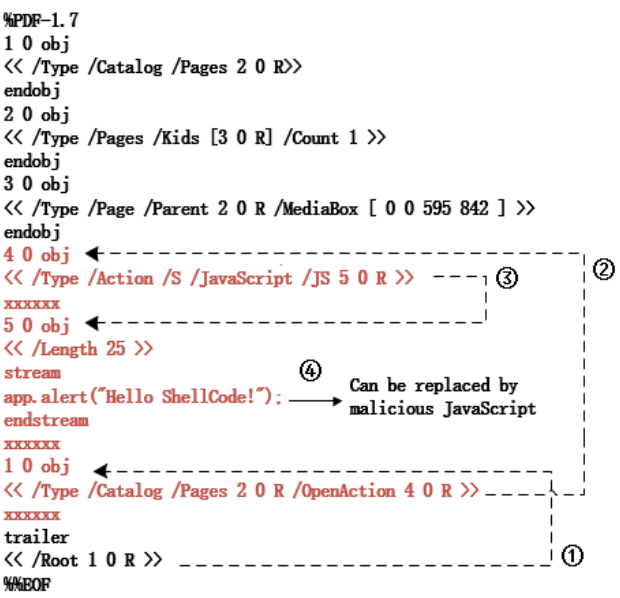

Fig.5 An instance of the template

In this template instance, we replace the keyword **endobj** with **xxxxxx**. In this way, the current feature extractors cannot identify the objects in the template and ignore them instead. While the Adobe Reader will still recognize them as objects.

The new root object in this template instance makes the Adobe Reader recognize it as the real and the unique root object. While the current feature extractors ignores the template because of the replacement of keyword **endobj,** and still recognizes the original one as the root object. The keyword **OpenAction** in the new root object enables that the JavaScript code can be executed automatically while opening the PDF by Adobe Reader**.**

In this template instance, the cross-reference table is missing, but Adobe Reader can also parse the PDF document and execute the JavaScript code. We can embed the malicious template into any benign PDFs easily, because we do not need to modify the original objects in the PDF and the offset of the objects in the template do not need to appear in the **xref**.

Above method enables two attacks: one is the malicious JavaScript code in the template can be executed; and the other one is that the features extracted by the current extractors will be same as the features extracted from the original benign PDFs. This is a disaster for the detection methods based on classifiers. The template can be created with the malicious JavaScript code found in the known malicious PDFs. It should also be noted that not only the malicious JavaScript code but also other malicious segments can be used to create the malicious template.

To imitate an attack scene, we download paper [1] and embed a test instance into it. As figure 6 shows, when the modified paper opened by Adobe Reader, it is still able to read but the JavaScript code is also executed. This means that this template can be used to attack those users who are interested in its content. Because attacks in reality won't pop up warning box, so the modified PDF looks like a normal document to users.

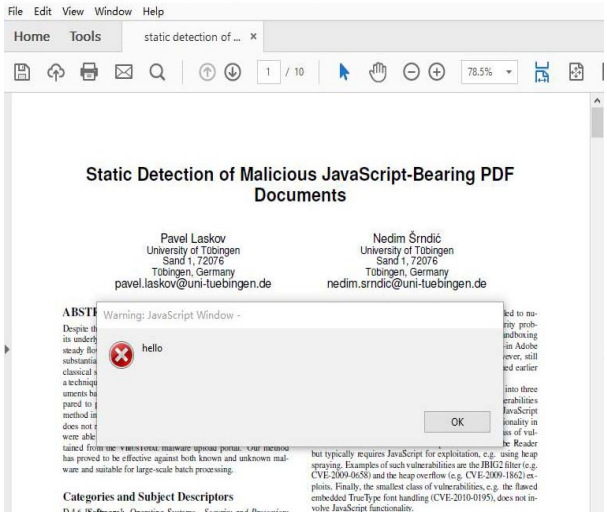

Fig.6 Embedding the template into a normal PDF document

IV. FEPDF: SYSTEM DESIGN

FEPDF is designed to parse and extract features from the PDF documents. It can extract those features that may be ignored by existing PDF parsers or feature extractors. It will not only parse the PDFs according to the PDF references but also search every suspicious segment. FEPDF can be divided into five parts as follows:

A. Matching Method

The matching method is the basis of FEPDF for parsing the PDF documents. We use the keywords to build a Trie [16] for matching the keywords in PDFs and record the offsets of the keywords while scanning the PDFs. By this way, we can partitioning the PDF documents according to the offset of the keywords like **obj**, **endobj**, **stream**, **endstream**, **trailer**, **xref**, and so on.

B. Detecting the PDF Header

The PDF specifications only require the header to appear somewhere within the first 1024 bytes of the file [2]. Usually, benign documents do not need to obfuscate PDF header, but malicious documents will do. Actually, it has been shown that manipulating the file type can evade anti-virus software [17].

Raynal et al. [18] produced *polyglot* files that are also valid PDF files. The trick is to store the PDF content inside the JPEG file, in which the PDF header appears in the first 1024 bytes of the file. Similarly, Albertini published numerous variants of PDF files that are– among others – valid ZIP, Flash, HTML or Java files [19] [20].

Algorithm 1 shows the algorithm FEPDF for inferring the file type. Our system will record all the file-types matching the regex signatures and record the position where the header appears.

Algorithm 1 File-type inference algorithm

```
Read first 1024 bytes of input file into buf, Check buf for regex file-type signature pdf and signature r
type_list = []
type_offset = []
if buf matches the regex signature pdf then
    type_offset.push_back ( pdf.offset )
    type_list.push_back ( pdf.filetype )
end if
for buf matches the regex signature r then
    type_offset.push_back ( r.offset )
    type_list.push_back ( r.filetype )
end for
return type_list, type_offset
```

C. Detecting All Objects

As we shown in section 2, a template exists to prevent the traditional PDF parsers from extracting the malicious objects. Hence, FEPDF will find the objects not only by the traditional ways but also scanning every suspicious segment that is potentially malicious. Although it seems trivial to find the template, but some problems associated with the template make it harder to be detected in reality. Here we list them as follows:

Reference Chain: Indirect objects may exist between the root and the object with real data. These objects form a reference chain, and it makes the extraction of JavaScript more complicated. FEPDF will not only record the objects but also the reference chain, which is very important for existing malicious PDF detection methods.

Let I be the set of all possible objects, O the set of possible indirect objects reference chains, and $S_c : I_S \rightarrow O_S$ the specification for chain c, where $I_S \subseteq I$ and $O_S \subseteq O$. A traditional document parser or feature extractor E_{tra} can only process the compliant objects:

$$E_{tra}(x) = \begin{cases} I_S(x) & \text{if } x \in I_S \\ Error & \text{if } x \in I - I_S \end{cases}$$

In reality, some objects like the proposed exploit template lying in $I - I_S$ can evade the detection. We design the extractor to scan the suspicious segments according to the regular expression and the objects' reference relations. So, FEPDF has its idiosyncratic way Sd of parsing these objects:

$$E_{our}(x) = \begin{cases} I_S(x) & if\ x \in I_S \\ S_d(x) & if\ x \in I_d \ \text{ where } S_d: I_d \rightarrow O_S \\ Error & if\ x \in I - (I_S \cup I_d) \end{cases}$$

Sometimes, there may be some segments cannot be recognized as objects by FEPDF but these segments will be recorded as unknown segments.

Hexadecimal Digit in keywords: In PDF 1.2 and higher version, any character except null (character code 0) may be included in a name by writing its 2-digit hexadecimal code, preceded by the number sign character (#). Many malicious PDFs use this trick to hide keywords. For example, we can change "Catalog" to "C#61t#41log". FEPDF will normalize these characters while parsing the PDF documents, and the original characters will also be recorded as features.

Compression: Filter is an optional part of a stream, describing how the data in the stream must be decoded before it is used. The PDF malware is typically decoded, so that signature matching method will not find it. FEPDF provides methods to decode the streams in the PDF documents according to the standard filters introduced in PDF references.

Redundant Information in JavaScript: Attackers usually add many comments into the embedding JavaScript code, which makes the extracted JavaScript full of useless information. This obfuscation overwhelms the extracted features of malicious code. In order to cope with this situation and improve the static detection accuracy, we remove that useless information. The algorithm is described as follows:

Algorithm 2 Removing useless information algorithm

```
Read the JavaScript string s
while i < s.length do
  if s[i:i+2] == '/*' then
    for j = i + 2 to s.length - 1 do
      if s[j:j+2] == '*/' then
        Break
      end if
    end for
    s = s[0:i] + s[j+2:]
  else then
    ++i
  end if
end while
record s
```

Invalid Objects: Malicious PDFs detection methods based on the machine learning rely on the features extracted from the PDFs such as the frequency of some keywords. Laskov et al. [21] proposed to add content directly after the cross-reference table but before the end-of-file marker [15]. Such content will be ignored by Adobe Reader, but not by some feature extractors. FEPDF does not ignore them either, but it will record the offset of every essential element in the PDF such as the offset of object, cross-reference tables and trailer, etc. Invalid objects will be found according to the offset.

D. Detecting Cross-reference

According to the PDF reference, when Adobe Reader opens a PDF file with a damaged or missing cross-reference table, it attempts to rebuild it by scanning all the objects in the file. But most malicious PDF detection models do not rebuild the cross-reference table, and some of them directly scan objects according to the cross-reference table. Instead, FEPDF will check the cross-reference table according to the objects scanned, and it will record the original cross-reference table and the rebuild cross-reference table.

E. Detecting Trailer

The trailer enables the application finding the cross-reference table and certain special objects quickly. As shown in Figure 4, the trailer section starts with keyword **trailer** and end with keyword **%%EOF**. The dictionary list << /Size 5 /Root 1 0 R>> defines the root object and the keyword **startxref** declares that the Byte offset of the keyword **xref** in the last cross-reference section is **361**.

The offsets of the **xref** and the special objects are important for many detection models. FEPDF will record the original trailer section, and it will also check and record the real offset of the special objects and the **xref**.

V. EVALUATION AND ANALYSIS

First, we collect a number of malicious PDFs and benign PDFs from the wild. Then we extract the malicious segments from those malicious PDFs for creating malicious template instances and embedded the instances into the collected benign PDFs. In this way, we create many new malicious PDFs for testing the current Anti-virus engines and the feature extractors. To be more intuitive, we implemented a PDF analyzing system based on FEPDF and the traditional malicious JavaScript detection model JSUNPACK [13]. The only difference is that our system parses the PDFs and extracts the JavaScript code by FEPDF. In this section, we present the experimental evaluation result of our prototype system.

A. Performance of Antivirus Engines

For testing the current antivirus engines, we select 5 benign PDFs and 5 malicious PDFs which are embedded with malicious JavaScript. Then we extract the malicious JavaScript code from the 5 original malicious PDFs and combine them with the template to create the instances. By embedding the instances into the 5 collected benign PDFs, we create 5 new malicious PDFs. Then we send these 10 malicious PDFs to VirSCAN.org, a free on-line scan service. The service has 39 antivirus engines. We count the number of engines that successfully identify the malicious PDFs.

TABLE 1. The detection rate of the original and new mal-pdfs

Samples	MD5	Result(m/N)
O_Sample 1	dad3ac62d35176cbc37731bded9353f1	17/39
N_Sample 1	e96dbd496bee8af264fe3c513669d15f	4/39
O_Sample 2	c0a77705b5e45a4da3b6a0a6a80f476f	17/39
N_Sample 2	bbde5615b9c7cc4504606db2cf2ba0a3	5/39
O_Sample 3	0adfb51aa465bfe077adfa52d1aef6f9	18/39
N_Sample 3	fa7d89e5c873092cd30c7735baa0b3fd	5/39
O_Sample 4	c16f4cc276701b869a92753751b03356	15/39
N_Sample 4	aca7e3658fb0b422ce1d3094c3b0c76a	5/39
O_Sample 5	61f2e2a576fc4e4e534078679d8e18ba	13/39
N_Sample 5	1fc19a26960420944c8cff1fafc28a45	5/39

As table1 shows, the original malicious PDFs are named as O_Sample *x* and the new malicious PDFs are named as N_Sample *x*. We can find that those documents exploiting old vulnerability can evade the detection of anti-virus engines using the exploit template.

B. Performance of the Feature Extractors

To evaluate the efficiency of the current feature extractors and the FEPDF, we design a controlled experiment. We select those previous 5 original benign PDFs used to create the new malicious PDFs as benign samples, and select those 5 newly created malicious PDFs as malicious samples. Then we modify the current feature extractors to print the features they extracted. By this way, we can identify whether the feature extractor can extract different features between the benign PDFs and the malicious PDFs.

TABLE 2. Performance of feature extractors

Feature Extractor	Can extract different features
Poppler	No
iText	No
JSUNPACK	No
PdfJsExtractor	No
FEPDF	**Yes**

As shown in table 2, the current feature extractors cannot find the differences between the benign and the malicious PDFs. Because the exploit template prevents the malicious JavaScript code or other features like keyword **OpenAction** from being extracted. Hence, any PDF detection models that use those feature extractors cannot deal with this evasion exploit. In comparison, our proposed feature extractor, FEPDF, can extract the malicious JavaScript code and other features embedded in the PDFs. The types of features that can be extracted by the feature extractors are listed as follows.

TABLE3. Types of features

Extractors	JSUNPACK	iTex	Poppler	PJExtactor	**FEPDF**
Header type	**Yes**	No	No	No	**Yes**
Refer-Chain	No	No	No	No	**Yes**
Hex-Digit	**Yes**	No	**Yes**	**Yes**	**Yes**
Compression	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Redundant-Information	No	No	No	No	**Yes**
Invalid-Obj	No	No	No	No	**Yes**
Xref feature	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**
Trailer feature	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

C. Performance of JavaScript detecting

JavaScript code is the most widely used in PDF attack, and we designed an analyzing system based on the traditional malicious JavaScript detection model JSUNPACK. The only difference between our system and JSUNPACK is that we use FEPDF to extract the JavaScript code. After that, JavaScript code will be sent to JSUNPACK's original JavaScript detection model. Using this way, we can test whether the new system can be more effective in feature extracting than the original system.

We select 1500 malicious PDFs including 100 new malicious PDFs created by the template and 2000 benign PDFs embedded within normal JavaScript code as the test samples. Recall, precision and accuracy are used as the evaluation indicators. TP is the number of malicious PDFs which were identified successfully. FP is the number of benign PDFs which were identified as malicious. TN is the number of benign PDFs which were identified successfully. And FN is the number of malicious PDFs which were identified as benign. Table 4 and table 5 show the test results.

TABLE 4. Test results

SYSTEM	TP	FP	TN	FN
JSUNPACK	1257	134	1866	243
OUR-SYS	1363	34	1966	137

TABLE 5. Evaluation results

SYSTEM	Precision	Recall	Accuracy
JSUNPACK	90.37%	83.80%	89.23%
OUR-SYS	97.57%	90.87%	95.11%

There are 243 malicious PDFs which JSUNPACK cannot identified, 106 of them are structure obfuscated and the embedded JavaScript cannot be extracted by JSUNPACK's extractor, the other 137 PDFs' JavaScript code can be extracted, but the JavaScript detection model failed to identify them as malicious. Our system can extract all the JavaScript code embedded in the PDFs, the 137 unidentified PDFs are same as the 137 unidentified PDFs by JSUNPACK for the same reason that the detection model relying on the designed rules of YARA which limits the accuracy of the detection.

There are 134 benign PDFs were identified as malicious by JSUNPACK, these PDFs were embedded with benign JavaScript code which contains long variable names and JavaScript APIs like "util.printf" and some long strings in its comment. The 34 misjudged benign PDFs by our system contain long variable names and JavaScript APIs.

Different from the original feature extractor of JSUNPACK, FEPDF will scan every segment and try to extract the suspicious JavaScript code and remove the comment in the code. Hence, JavaScript in the structure obfuscated PDFs can also be extracted and the comment in the code will not confuse the JavaScript detection model. This enables that more malicious PDFs can be detected and less benign PDFs would be misjudged as malicious. As the results show, FEPDF is more robust and accurate than JSUNPACK, and this also indicates the effectiveness of the designed feature extractor.

VI. Conclusion

In this paper, by reviewing existing PDF feature extractors and analyzing the implementation of the malicious template, we learned the weakness of existing feature extractors. Then, we designed a robust feature extractor FEPDF, which can record the realistic information of the elements in the PDFs and extract features that may be ignored by other feature extractors. For testing the current Anti-virus engines and the feature extractors, we created many new malicious PDFs as samples. The results show that many existing Anti-virus engines failed to identify the new malicious PDFs, but FEPDF can extract the essential features for improved malicious PDF detection.

This research highlights once more the importance of developing a robust feature extractor for the malicious PDFs detection system. As a counter measure, we proposed the feature extractor FEPDF, which has shown to be effective and robust. It extracts more malicious features than existing feature extractors, and it can be easily integrated with different detecting models such as JSUNPACK.

Acknowledgment: This work is supported by Strategic Pilot Technology Chinese Academy of Sciences (No. XDA06010703), National Natural Science Foundation of China (No. 61173008, 61402124, 61303244), Young Scholar Foundation of Institute (No. 1104005704) and Key Lab of Information Network Security, Ministry of Public Security (No. C15607).

VII. References

[1] P. Laskov, Šrndic Nedim, "Static detection of malicious JavaScript-bearing PDF documents," in Twenty-Seventh Computer Security Applications Conference, Orlando, 2011, pp. 373-382.

[2] Acrobat standard. http://wwwimages.adobe.com/content/dam/Adobe/en/devnet/pdf/pdfs/PDF32000_2008.pdf, 2017

[3] I. Corona, D. Maiorca, D. Ariu, "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, Scottsdale, Arizona, USA, 2014, pp. 47-57.

[4] D. Maiorca, G. Giacinto, I. Corona. "A pattern recognition system for malicious pdf files detection," in Machine Learning and Data Mining in Pattern Recognition, Springer Berlin, Heidelberg, 2012, pp. 510-524.

[5] C. Smutz, A. Stavrou. "Malicious PDF detection using metadata and structural features," in Annual Computer Security Applications Conference (ACSAC), 2012, pp.239-248.

[6] N. Šrndic, P. Laskov, "Detection of malicious pdf files based on hierarchical document structure," in Proceedings of the 20th Annual Network & Distributed System Security Symposium, San Diego, 2013, pp. 27-36

[7] D. Maiorca, D. Ariu, I. Corona, "A structural and content-based approach for a precise and robust detection of malicious PDF files," in International Conference on Information Systems Security and Privacy, Angers, Loire Valley, France, 2015, pp. 27-36.

[8] A. Cohen, N. Nissim, L. Rokach and Y. Elovici, "SFEM: Structural Feature Extraction Methodology for the Detection of Malicious Office Documents Using Machine Learning Methods." Expert Systems with Applications, vol. 63, pp. 324-343, Nov. 2016

[9] N. Šrndić, P. Laskov, "Hidost, a static machine-learning-based detector of malicious files," Eurasip Journal on Information Security, pp. 1-20, 2016.

[10] Z. Tzermias, G. Sykiotakis, M. Polychronakis, "Combining static and dynamic analysis for the detection of malicious documents," in Proceedings of Workshop on European Workshop on System Security Eurosec, Salzburg, 2011, pp. 1-6.

[11] L. Xun, J. Zhuge, R. Wang, "De-obfuscation and Detection of Malicious PDF Files with High Accuracy," in Hawaii International Conference on System Sciences (HICSS), Hawaii, 2013, pp. 4890-4899.

[12] Poppler. https://poppler.freedesktop.org/, 2017

[13] JSUNPACK. http://jsunpack.jeek.org/, 2017

[14] iText. http://itextpdf.com/, 2017

[15] G. Endignoux, O. Levillain, J Y. Migeon, "Caradoc: A Pragmatic Approach to PDF Parsing and Validation," in IEEE Security and Privacy Workshops (SPW), 2016, pp. 126-139.

[16] Trie. https://en.wikipedia.org/wiki/Trie/, 2017

[17] S. Jana, V. Shmatikov, "Abusing File Processing in Malware Detectors for Fun and Profit," in IEEE Symposium on Security and Privacy (S&P), 2012, pp. 80-94.

[18] F. Raynal, G. Delugré, D. Aumaitre, "Malicious origami in PDF," Journal of computer virology and hacking techniques, vol. 6, pp. 289-315, Nov. 2010.

[19] A. Albertini. This PDF is a JPEG; or, this proof os concept is a picture of cats. *PoC or GTFO ox03*, 2014.

[20] A. Albertini. Abusing file formats; or, Corkami, the Novella. *PoC or GTFO 0x07*, 2015.

[21] N. Rndic, P. Laskov, "Practical Evasion of a Learning-Based Classifier: A Case Study," in IEEE Symposium on Security and Privacy (S&P), 2014, pp. 197-21.