Extract Me If You Can: Abusing PDF Parsers in Malware Detectors

Curtis Carmony
Syracuse University
curtis.carmony@gmail.com

Mu Zhang
NEC Laboratories America
mu@nec-labs.com

Xunchao Hu
Syracuse University
xhu31@syr.edu

Abhishek Vasisht Bhaskar
Syracuse University
abhaskar@syr.edu

Heng Yin
Syracuse University
heyin@syr.edu

***Abstract*—Owing to the popularity of the PDF format and the continued exploitation of Adobe Reader, the detection of malicious PDFs remains a concern. All existing detection techniques rely on the PDF parser to a certain extent, while the complexity of the PDF format leaves an abundant space for parser confusion. To quantify the difference between these parsers and Adobe Reader, we create a reference JavaScript extractor by directly tapping into Adobe Reader at locations identified through a mostly automatic binary analysis technique. By comparing the output of this reference extractor against that of several open-source JavaScript extractors on a large data set obtained from VirusTotal, we are able to identify hundreds of samples which existing extractors fail to extract JavaScript from. By analyzing these samples we are able to identify several weaknesses in each of these extractors. Based on these lessons, we apply several obfuscations on a malicious PDF sample, which can successfully evade all the malware detectors tested. We call this evasion technique a *PDF parser confusion attack*. Lastly, we demonstrate that the reference JavaScript extractor improves the accuracy of existing JavaScript-based classifiers and how it can be used to mitigate these parser limitations in a real-world setting.**

I. Introduction

Even though Adobe's Acrobat Reader, more commonly known as Adobe Reader, has become increasingly secure through the addition of advanced security mechanisms such as a sandbox [5], new exploits continue to be found with 44 CVEs published in 2014 [1] and 128 published in 2015 at the time of writing [2]. Due to the continued exploitation of Adobe Reader along with the ubiquity of the PDF format, the detection of malicious PDF files remains a concern, with Kaspersky reporting that Adobe Reader was the third most exploited target in 2014 and attracted 5% of the overall attacks [18].

Malicious PDF detection in commercial anti-virus products relies heavily on signature detection and is insufficient to detect PDFs containing zero-day exploits or advanced persistent threats. To address this limitation, two classes of systems have been proposed to detect malicious PDF files specifically: 1) structure and metadata based detectors [29], [32], [38] and 2) JavaScript-based classifiers [23], [25], [37], [26].

Structure and metadata based detection methods distinguish benign and malicious PDFs by determining which structural features and metadata are most associated with each class. However, the essential malice of PDF exploits does not originate in file structures but rather in embedded payloads (e.g., JavaScript code) that bear malicious intent. Therefore, these detectors can be easily evaded by the *mimicry attack* [39], [38] and the *reverse mimicry attack* [28], which hide harmful code in PDF files that exhibit structural features and metadata associated with benign files.

To fundamentally address PDF exploits, it is necessary to analyze the contents of documents and search for malicious payloads. Prior work [23] reveals that JavaScript is the most common malicious content in PDF exploits for two major reasons: 1) the implementation of the Adobe JavaScript APIs exposes vulnerabilities and 2) JavaScript code is used to enable advanced exploitation techniques, such as heap spraying. Almost all of the malicious PDF documents in our sample set collected from VirusTotal contain JavaScript, indicating that the extraction and analysis of embedded JavaScript is essential to malicious PDF detection.

To this end, prior JavaScript-based classifiers [23], [25], [37] have attempted to parse PDF documents, extract JavaScript from them, and then analyze this JavaScript to classify it as benign or malicious. These works all depend on their ability to accurately extract JavaScript from PDFs. With the exception of MPScan[26], which uses a modified version of Adobe Reader similar to the one presented in this work, each of these works rely either on open-source parsers or their own home-grown parsers. Because all of these parsers are incomplete and have oversimplified assumptions in regards to where JavaScript can be embedded, these detection methods are not accurate or robust.

In this paper, we aim to conduct a systematic study on a new evasion technique called the *PDF parser confusion attack*, which aims to confuse the PDF parsers in malware detectors in order to evade detection. In essence, this evasion attack exemplifies the *chameleon* and *werewolf* attacks that deliberately abuse file processing in malware detectors [22]. However, compared to other file types (e.g., ZIP, ELF and PE) that have been investigated in this previous work, the combination of the complexity of the PDF format and Adobe Reader's leniency in parsing these files potentially offers a much larger attack space. Unfortunately, this attack space has not been studied sufficiently in the security community.

NDSS '16, 21-24 February 2016, San Diego, CA, USA
Copyright 2016 Internet Society, ISBN 1-891562-41-X
http://dx.doi.org/10.14722/ndss.2016.23483

To enable a systematic study we have developed a *reference JavaScript extractor* by directly tapping into Adobe Reader, which is arguably the most popular and most targeted PDF viewer [19]. To develop this reference extractor, we present a mostly automatic dynamic binary analysis technique that can quickly identify a small number of candidate tap points, which can be further refined by simple manual analysis. We then perform a differential analysis on this reference extractor and several popular extractors, using over 160,000 PDFs collected from VirusTotal. For each extractor we identify hundreds of samples which it cannot correctly process, but that contain JavaScript according to the reference extractor.

By delving into these discrepancies between the reference extractor and the existing extractors we have identified several new obfuscations, and further quantified their impact when used in parser-confusion attacks on JavaScript extractors and malware detectors. By combining several of these obfuscations, we demonstrate that a malicious PDF can successfully evade *all* the malware detectors evaluated, including signature-based, structure/metadata-based, and JavaScript-based detectors.

These findings suggest that the key to effective countermeasures is a high-fidelity parser that closely mimics the parsing logic of Adobe Reader. One possible solution is to directly deploy our reference JavaScript extractor for JavaScript-based detectors. Our experiment shows that this deployment scheme not only incurs acceptable runtime overhead, but also produces much higher detection accuracy. Our experiments show that after replacing the original parser with our reference extractor, the detection rate of PJScan [23] increases from 68% to 96% for a specific version of Adobe Reader, based on a fairly rudimentary classifier.

Paper Contributions. In summary, this paper makes the following contributions:

- We propose a mostly-automatic, platform independent tap point identification technique to correctly identify tap points related to JavaScript parsing and execution in Adobe Reader which are used to develop a reference JavaScript extractor.
- Using our reference extractor we systematically evaluate the shortcomings of existing JavaScript extraction tools. We have identified hundreds of PDF samples (both benign and malicious), which existing extractors failed to extract JavaScript from. We manually investigate many of them, and identify their root causes.
- We construct several PDF parser confusion attacks by combining several of the obfuscations identified in our analysis. These evasions have proved to be effective in successfully evading all of the malware detectors we tested.
- We discuss several mitigation techniques. In particular, we demonstrate that with our reference JavaScript extractor, the detection rate of an existing classifier increases significantly from 68% to 96% on our sample set, and present a possible deployment scenario for the reference extractor.

We plan to release the complete data set and also launch a public service for our reference JavaScript extractor, to help security researchers conduct further research on this problem. A list of MD5 hash values are available for part of the data set and can be found at https://goo.gl/qtbuOC.

II. Background

A. Metadata and Structural Features Based Detection

Since signature-based malicious PDF detectors [31] are susceptible to various conventional malware polymorphism techniques [16], [17], [34], efforts have been made to find more robust malicious PDF detection methods. Based off of the observation that malicious files usually have little or no content aside from their payloads, and that benign files usually have an extensive set of diverse contents, several systems have been presented to quantify these structural differences to facilitate malicious PDF detection and classification.

PDF Malware Slayer [29] uses the PDF keywords identified in a sample by the popular PDFiD tool [35] as a feature set which they use to train a random forests PDF classifier. The primary limitation of this system lies in its use of the PFDiD tool which merely performs simple string matching to identify the existence of a subset of PDF keywords, and therefore cannot recognize strings encoded by a filter or differentiate the strings in the document structure from those in its contents.

PDFrate [32] similarly uses a random forests classifier, but utilizes a much more descriptive feature set. It parses PDF files to retrieve 202 different structural aspects of a sample such as the number and types of objects in the file, their size, aspects of their contents, pages in the document, and the size of the file. Again, this work is largely limited by its parser, a program developed by the authors which utilizes regular expressions to extract these features which cannot decode encoded streams in the file or parse their contents.

Taking a similar approach Šrndić and Laskov [38] developed a system which extracts the tree-like structure of the objects within a PDF as a feature set for classification. Despite its accuracy in an offline experiment, the use of this system in an operational test demonstrates this system cannot always correctly identify new threats.

While metadata and structure based malicious PDF detection systems have been shown to be both efficient and effective, they are fundamentally susceptible to evasion. Prior studies [38], [39] have demonstrated, either anecdotally or in a systematic study of PDFrate [9], that these classifiers are subject to so-called mimicry attacks. In these attacks, the structural features of a malicious sample are modified to resemble that of a PDF document already classified as benign. Because the malicious behaviors exercised by PDF malware do not necessarily depend upon specific structural features, this technique can evade these classifiers while preserving the efficacy of the original exploits.

A second type of attack, the *reverse mimicry* attack, has also been presented [28]. Whereas a mimicry attack adds benign attributes to malicious samples, the reverse mimicry attack takes a sample classified as benign and makes it malicious. The launch of such attacks is even easier because it does not

TABLE I: Existing PDF Classifiers

Technique	Detectors	Detection Capability	Parser Requirement	Evasion Techniques
Signature-based	AV Scanners Shafiq et al. [31]	Varies	Low - Medium	Malware Polymorphism [16], [17], [34]
Metadata & Structure -based	PDF Malware Slayer [29] PDFrate [32] Šrndić and Laskov [38]	Medium	Medium	Mimicry Attack [39], [38] Reverse Mimicry Attack [28]
JavaScript-based	Liu et al. [25] MDScan [37] PJScan [23]	Varies	High	

require knowledge of the targeted classification system, which mimicry attacks largely depend on.

B. *JavaScript Based Detection*

The evasions of these systems demonstrate that malicious PDF detection techniques that rely only upon structural and metadata similarities are insufficient. Given that most malicious PDFs use JavaScript to either trigger or set up exploits, prior classifiers focus on the extraction and analysis of embedded JavaScript instead. Since these detection methods depend on JavaScript analysis for classification, an accurate parser is essential to correctly interpret the entire PDF file and precisely locate JavaScript.

Liu et al. [25] attempt to identify and instrument automatically executing JavaScript in a document, so as to attribute suspicious behavior exhibited by Adobe Reader at runtime to the executing JavaScript. The runtime observations along with other heuristics are then used to compute a score for classification. This system is limited by its overly simplistic JavaScript extraction, only associating JavaScript with two keywords, and assuming that they must always appear in plaintext. In fact JavaScript can be embedded in multiple layers, using extensions to the format (e.g., XFA), and can be encoded by using diverse PDF features, such as object streams and filters.

Realizing that malicious JavaScript often utilizes the Adobe JavaScript API to read the contents of objects in a PDF, MDScan [37] parses a PDF not only to extract embedded JavaScript, but to load the internal structure of the document as well. The extracted JavaScript is then run in a modified JavaScript engine, augmented to support certain elements of the Adobe JavaScript API which the authors reverse engineered, so that calls to these supported API functions mimic the behavior of Adobe Reader. While this system provides a more complete platform for dynamically analyzing this JavaScript, it is inherently incomplete due to partial API support, which is non-trivial, error-prone, and considerably time-consuming to improve. This work realizes that JavaScript can be encoded in various ways, but it only associates JavaScript with one keyword, which is a severe limitation.

PJScan [23] extracts JavaScript and uses the tokenized JavaScript as the feature set used to train a One-Class Support Vector Machine. This system falls short of accuracy primarily due to its PDF parser libpdfjs, which is built upon a third-party parser, Poppler [10]. While Poppler claims to implement the entire PDF ISO 32000-1 specification [24], it does not claim to address the discrepancies between the specification and the closed-source implementation of Adobe Reader, all of the addendums to the specification, or all of the specifications extensions such as XFA.

Without considering the shortcomings specific to each detector, these JavaScript-based PDF classifiers are all limited by their JavaScript extraction capabilities. Not only must PDFs be parsed correctly, but these detectors have to statically identify all of JavaScript components embedded in the document. Because JavaScript can be embedded in many different ways, or even using extensions to the specification which these detectors do not implement, they are unlikely to always produce all of the JavaScript in a document, especially those which have been obfuscated.

To resolve these parsing issues, Lu et al introduced MPScan which hooks Adobe Reader's JavaScript engine to produce the JavaScript executed by Adobe Reader when opening a document, which is then classified as malicious or benign using shellcode and heap-spray detection techniques[26]. While the authors are able to mitigate all of these parsing issues, they are only able to hook one version of Adobe Reader, and do not present any technique for identifying the points to hook the binary or describe how they did so. In the absence of this information, one must assume they did so through manual analysis, which is an arduous task for a program as large and complex as Adobe Reader, and which must be repeated for every new version of Adobe Reader.

C. *Summary and Hypotheses*

As presented in Table I, prior PDF classifiers have been evolving to grasp the semantics of malicious payloads to defeat rudimentary attacks which create polymorphic malware or imitate benign file structures.

Hypothesis 1: A key observation is that all previous detection methods rely on parsing and interpreting PDF files to a certain extent. Consequently, their detection accuracy critically depends on the quality of their PDF parsing and JavaScript extraction. Thus, we hypothesize that a delicate attack can be launched to evade all these classifiers, provided it can successfully confuse the PDF parsers that are utilized in detection.

Hypothesis 2: We also realize that, in order to perform accurate and robust PDF classification, it is crucial to actually examine the embedded JavaScript payloads in PDF files. As a result, we hypothesize that the improvement of JavaScript

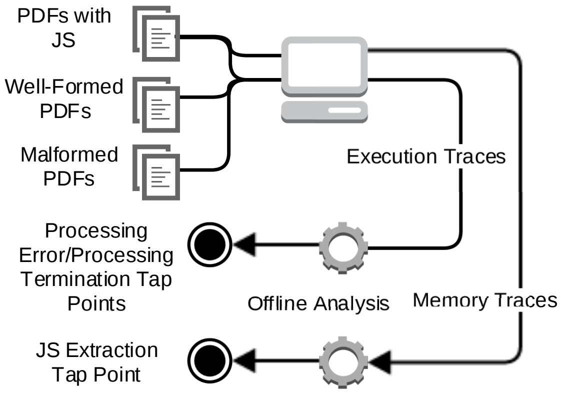

Fig. 1: Tap Point Identification.

extraction can facilitate the detection of malicious documents that are not detected by existing JavaScript-based PDF classifiers.

III. Reference JavaScript Extractor

To verify our hypothesis, we need to develop a reference JavaScript extractor so as to quantitatively measure the discrepancies between existing PDF parsers deployed in detectors and Adobe Reader.

A. The Need For a New Technique

MPScan demonstrated that a JavaScript extraction tap point can be manually identified in Adobe Reader. While the amount of effort required is not described, given the size and complexity of Adobe Reader–IDA Pro identifies 91,753 and 133,835 functions in the main AcroRd32.dll component alone, for the 9.5.0 and 11.0.08 versions of Adobe Reader we worked with, respectively–it could not have been a simple task.

Since a reference extractor can only precisely mimic the behavior of a single version of Adobe Reader, our three tap points must be identified for each version of Adobe Reader that is to be protected, making the manual application of this technique infeasible as a general solution to the problem of malicious PDF detection. By developing a technique which is repeatable and automatable we can reliably produce reference extractors for many versions of Adobe Reader with minimal effort. Such a technique can also conceivably be applied to extract executable code from other file formats, such as embedded VBA macros in Microsoft Office documents. While we are unable to automate this process completely, we have been able to automate the majority of the analysis. Overall, we have found that once the technique has been implemented to develop one reference extractor, it only takes a few hours of manual effort develop a new reference extractor based on a different version.

B. Overview

Figure 1 depicts the workflow to build our reference extractor. We first open three classes of labeled PDF samples (i.e., well-formed PDFs with JavaScript, well-formed PDFs without JavaScript and malformed PDFs) with Adobe Reader in an execution monitor [20] to collect memory access and execution traces. In the end, we perform offline analysis on the traces to identify three *tap points* that are associated with JavaScript extraction, PDF processing termination and processing error. We can then create the reference JavaScript extractor by modifying Adobe Reader at these three tap points.

In particular, we identify these points by comparing the distinctive traces of multiple classes. By processing the memory access traces of well-formed PDFs with automatically executing JavaScript, we identify *JavaScript extraction tap points*, where embedded JavaScript code is extracted and executed. By examining the divergent execution traces of well-formed and malformed files, we discover *processing termination tap points* and *processing error tap points*, which represent the end of file processing in these two classes, respectively.

It is worth noting that we define the well-formedness of a PDF file based on the behavior of Adobe Reader when opening file. We do not rely on the PDF specification for such a determination, because 1) some specification items are vague and cannot be easily interpreted, and 2) the implementation of Adobe Reader in fact deviates from the specification in order to increase compatibility. Thus, we consider PDFs which Adobe Reader opens successfully to be well-formed. Conversely, we consider a PDF to be malformed if Adobe Reader opens an alert indicating it was unable to open the sample. The sample sets used to perform our analysis were manually constructed by opening samples with Adobe Reader to determine if they are well-formed or malformed according to our definition, and through manual analysis to determine whether or not they contained any automatically executing JavaScript.

While we use an existing work as the basis for JavaScript extraction [15], this technique is insufficient for the development of the reference extractor for three reasons. Firstly, because it relies on an execution monitor for extraction it is too slow to process any significant number of samples. Secondly, the technique focuses on extracting all targeted information while the monitoring system is run and so provides no mechanism for determining when all of the data has been extracted, which we need in order to expediently process samples. Lastly, the technique does not handle the situation where no targeted information exists or it cannot be extracted, i.e. a PDF contains no JavaScript or Adobe Reader fails to process the PDF because it is malformed.

To address these limitations, our technique is different from the previous one in three aspects. Firstly, in contrast to this previous work, which focuses on the instructions that access data and thus performs analysis solely on memory traces, we take into account both data accesses and control transfers. The addition of control flow analysis is necessary because not all of the information we wish to extract from Adobe Reader can always be determined by monitoring only memory accesses. For example, determining if Adobe Reader has encountered an error or has finished opening a PDF can likely only be determined by examining the program's state. Secondly, this previous work monitors memory accesses on an "operation" level and only groups contiguous memory accesses within a fixed number of memory accesses. However, this grouping is not well-suited to segmented memory access patterns, which are likely to appear in JavaScript processing. We instead keep track of memory access on the granularity of "operation groups" so as to capture contiguous operations spread over many access. Lastly, the original technique selects instructions as tap points, which can only yield the targeted data to a full system emulator. In the interest of performance we adapt the technique to locate functions so that existing function hooking techniques can be used.

Detailed explanations are presented in the following subsections. In all, we have tailored the existing technique to identify the JavaScript extraction tap point and extended it to identify the new processing error and processing termination tap points which are needed by the reference extractor.

C. JavaScript Extraction Tap Points

Definition. We consider a *JavaScript extraction tap point* to be a function, in which Adobe Reader extracts and executes JavaScript code from PDF documents. Formally, such a tap point is defined as a triple:

$$(caller, function\_entry\_point, argument\_number),$$

where the caller indicates the calling context of the function and the argument number is the index of the function parameter which holds a reference to a null-terminated string containing JavaScript.

We maintain the calling context of the tap point function to increase the accuracy of identification. Some common functions, such as *memcpy*, are likely to be invoked by multiple callers in a program. Only some of these calls are associated with JavaScript operations, however, so the introduction of context awareness can significantly help eliminate false identification.

We define tap points at the function level instead of the instruction level because function entry points are more resilient to conditional execution and provide a cleaner interface for hooking. For example, depending on the length of a string, different instructions in *memcpy* are used to copy the string.

Despite the advantage of function-level tap points, in practice, we have to initially identify instruction-level tap points, which we call *raw tap points*. Each raw tap point is defined formally as a pair:

$$(caller, program\_counter^{1}),$$

where the caller is also the caller of the host function and the program counter uniquely represents the address of the instruction.

Once a raw tap point is discovered, data-flow analysis is needed to correlate the identified instruction with a certain argument of the host function. Hence, we can eventually retrieve the function-level tap points.

Memory Access Trace. The identification of raw JavaScript extraction tap points is performed by analyzing the memory accesses made by Adobe Reader while opening PDFs which contain automatically executing JavaScript. All of these *memory accesses* are logged in a memory trace, where each access *m* is formatted as a tuple:

$$m = (caller, program\_counter, type, data, addr)$$

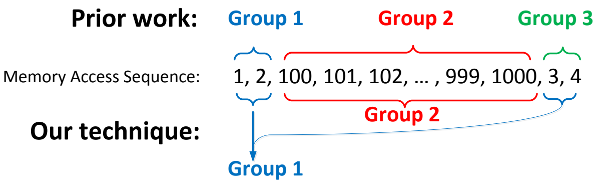

Fig. 2: A Comparison of Identifying Contiguous Memory Operations between Prior and This Work

That is, the calling context, the address of the instruction producing the access, the type of the access (either a read or a write), the data written or read, and the address of memory being accessed. In the cases where one layer of calling context is not sufficient, we can increase context sensitivity by adding another layer of caller information. We refer to the set of memory traces collected for these samples as $\mathbb{M}_{JS} = [M_0, M_1, ..., M_n]$, where each M_i denotes the trace for an individual sample.

Identification of Raw Tap Points. Once the memory traces in $\mathbb{M}_{JS}$ have been collected, we perform offline analysis in two steps to identify the raw tap points. First, we group the memory accesses in each trace into *contiguous memory operations* then we examine these memory operations to search for JavaScript strings.

We keep track of memory accesses on the "operation group" level instead of the individual operation level in order to tolerate intermittent memory accesses that often happen in JavaScript processing where strings are likely to be parsed and executed segment by segment. The prior work only monitors a limited window (i.e., five) of *operations* at once and any operations beyond this limit, even if contiguous to a previous one, cannot be correlated with the previous operations. Figure 2 demonstrates the advantage of our approach compared to the prior work [15]. In this example, only three groups (i.e., {1,2}, {100...1000} and {3,4}) are identified by the prior work, though the accesses to memory region 3 and 4 continue the operations on 1 and 2. To address this limitation, we keep track of memory operations at a higher granularity and directly monitor several access groups at the same time. Thus, the sequential accesses to the memory regions from 100 to 1000 becomes one single group; both accesses to {1,2} and {3,4} can be observed within a window and therefore can be further grouped as one. While it is possible for the prior work to increase the window size in order to properly group the segmented memory accesses, this drastically increases the computation overhead.

We define *contiguous memory operations* as a list of instructions with the same calling context which access consecutive locations in memory in the same way. Formally, a group of contiguous memory operations, *g*, is defined as a triple:

$$g = (start, end, m\_list),$$

where the start and end are the beginning and ending addresses of the contiguous access, respectively, and m_list is a list of the individual memory accesses $[m_{start}, ..., m_{end}]$. To group these memory operations, we present Algorithm 1.

[1] For the brevity of presentation, we assume Address Space Layout Randomization (ASLR) is disabled. When ASLR is enabled, we in fact use module name plus offset to specify this raw tap point.

The algorithm takes a memory access trace M as an input and outputs a list of contiguous memory access groups. We also introduce a working list, WQ, which serves as a LRU cache to store a fixed number of access groups. In practice, a cache size of ten was sufficient, though it is possible that potential tap points are lost as a result of being prematurely pushed out of the cache. While this size can be increased, it increases the amount of memory necessary to perform the analysis.

To identify these groups, we iterate over each memory operation m in the trace M. If a read operation m matches an existing group, g in cache WQ in terms of calling context and the address read by m exactly succeeds the last one accessed by g, we perform $Extend()$, which inserts m at the end of g. If a read operation m falls in the middle of an existing group we move that group to the front of the cache to avoid discarding groups which have been recently accessed. Otherwise, m falls out of the boundary of each existing group which indicates the presence of a new contiguous memory access group. We create this new group using m and add it to the front of the cache, removing and saving the least recently used group if necessary. In the cases where m in fact writes into an existing group g in the cache, we invalidate the old group by removing it from the cache and saving it to output.

Once the contiguous memory operations have been collected for a sample, they are examined to find the automatically executed JavaScript code. Since the in-memory encoding of the JavaScript is not known, both UTF-8 and UTF-16 representations have to be searched for. If a JavaScript string is discovered, these memory operations are considered to be raw tap points.

Algorithm 1 Contiguous Memory Operation Identification

```
$M \leftarrow [m_0, m_1, ...m_n]$
$WQ \leftarrow$ an empty list of $g$
for each memory operation $m$ in $M$ do
    if $m.type = read$ then
        if $\exists g \in WQ \mid g.end+1 = m.addr$ and $g.caller = m.caller$ then
            $Extend(g, m)$
        else if $\exists g \in WQ \mid g.start \leq m.addr \leq g.end$ and $g.caller = m.caller$ then
            $WQ.move\_to\_front(g)$
        else    #$m$ falls out of all $g$ in $WQ$.
            $g_{new} \leftarrow CreateNewGroup(m)$
            $WQ.add\_to\_front(g_{new})$
        end if
    else    #$m$ is a write.
        if $\exists g \in WQ \mid g.start \leq m.addr \leq g.end$ then
            $WQ.remove\_and\_save(g)$
        end if
    end if
end for
```

Tap Points Refinement. Once raw tap points are discovered, we conduct a use-def chain analysis to see if the identified memory operations can be traced back to any function arguments. If so, the candidates of JavaScript extraction tap points are found. Since static analysis may introduce inaccuracy, we then perform runtime testing to validate these candidates.

The search for candidates is repeated for every memory trace with the set of potential tap points being reduced to those which produce the embedded JavaScript in all memory traces processed. A small amount of manual analysis is then used to determine which of these candidates is to be used and how the JavaScript can be extracted from them. In practice, this amounts to examining the tap points in a disassembler to identify one which easily yields the JavaScript. If no suitable tap point can be found, this offline analysis can be repeated including an additional level of calling context.

D. Processing Termination and Processing Error

Definition. In addition to the extraction of automatically executing JavaScript, it is also necessary to determine if Adobe Reader can successfully open a file or if will fail to do so because of some error. We define these states as

Processing Termination: The point at which Adobe Reader has successfully completed all of its initial processing associated with opening a PDF. Any automatically executing JavaScript must have executed before this point, and all elements on the page must be completely rendered.

Processing Error: Before completing its initial processing of a PDF Adobe Reader has encountered an error causing it to abort its processing. In this case, Adobe Reader will open an alert describing to the user the error it has encountered.

Adobe Reader's behavior will always fall into one of these two cases. Aside from producing useful information about the sample, Adobe Reader's reaching of one of these cases indicates when there is no more automatically executing JavaScript to extract. By terminating the process whenever one of these is reached cases is reached, we can expediently process samples.

We define the processing termination and processing error tap points to simply be the program counters of instructions, which when executed, indicate that Adobe Reader has reached each of these states. In practice, we found that the program counter alone was sufficient to identify these points but calling context can be added if needed.

Execution Trace. The identification of the the processing termination and processing error tap points is performed by analyzing only the instructions executed by Adobe Reader. Since only the execution of these tap points is used to reveal information about the program's state, this analysis can be simplified by performing it on the basic blocks executed, without loss of generality. The traces produced are thus lists containing the addresses of the first instruction of every basic block executed.

In particular, sets of these traces, $\mathbb{ET}_{JS}$, $\mathbb{ET}_{WF}$ and $\mathbb{ET}_{MF}$, were collected for well-formed PDFs with JavaScript, well-formed PDFs without JavaScript, and malformed PDFs (i.e. those which cause Adobe Reader to produce an error), respectively. More concretely, $\mathbb{ET}_{JS}$ consists of traces collected from the samples used to generate the memory traces as well as additional malicious PDFs which were manually identified as containing JavaScript. The traces in $\mathbb{ET}_{WF}$ were obtained from benign samples which were manually verified to not contain JavaScript. Lastly, $\mathbb{ET}_{MF}$ contains traces for malformed samples both with and without JavaScript, and for files which are of different formats entirely, such as PE and DOCX.

Each trace in $\mathbb{ET}_{JS}$ and $\mathbb{ET}_{WF}$ was collected until Adobe Reader had appeared to finish its initial processing of a sample, while traces in $\mathbb{ET}_{MF}$ were collected until Adobe Reader raised the alert indicating it could not open the file.

Processing Termination. A basic block selected for the processing termination tap point must meet three requirements: 1) the basic block is always executed once and only once when processing well-formed PDFs; 2) it is never executed by Adobe Reader when processing malformed PDFs; 3) it is only executed after the JavaScript extraction tap point if the PDF contains JavaScript.

According to requirement 1, we compute a set $BB_{WFUnique}$ which contains basic blocks common to all traces in $\mathbb{ET}_{WF}$ that appear only once in each trace. Based on requirement 2, we need to then exclude from $BB_{WFUnique}$ all the basic blocks executed in any trace in $\mathbb{ET}_{MF}$. We collect these basic blocks in a set BB_{MF} and then exclude it: $(BB_{WFUnique} - BB_{MF})$. Due to requirement 3, we have to first find the set $BB_{JSTrunc}$ that holds all basic blocks executed in any trace in $\mathbb{ET}_{JS}$ *after* the JavaScript extraction tap point is reached, and then compute the intersection as the set of potential processing termination tap points:$(BB_{WFUnique} - BB_{MF}) \cap BB_{JSTrunc}$. Any of the basic blocks in this set can be selected as the processing termination tap point.

Processing Error. A basic block associated with the processing error tap point must meet two requirements: 1) the basic block is always executed once and only once when processing malformed PDFs; 2) it is never executed by Adobe Reader when processing well-formed PDFs. To meet these two requirements, we compute three intermediate sets of basic blocks. Specifically, we first compute a set $BB_{MFUnique}$ that contains all basic blocks, common to all traces in $\mathbb{ET}_{MF}$, which appear only once in each of these traces. Then we exclude all basic blocks that are executed for well-formed PDFs, both with and without JavaScript. To this end, we collect two sets BB_{JS} and BB_{WF}, which represent all basic blocks in the sets $\mathbb{ET}_{JS}$ and $\mathbb{ET}_{MF}$, respectively. Thus, the set of potential processing error tap points is computed as $BB_{MFUnique} - (BB_{JS} \cup BB_{WF})$. Again, any of the basic blocks in this set can be used as the processing error tap point.

E. Tap Point Action

Once the tap points have been identified, Adobe Reader needs to be modified so as to log the JavaScript produced at the JavaScript extraction tap point and to terminate when the processing termination and processing error tap points are reached. If the tap point definition includes calling context, these modifications also need to determine at runtime if their current execution matches that context, otherwise the program's behavior should not be altered. By checking the existence of the log file and examining the process' exit code, which is specific to each tap point, it is easy to programmatically determine which tap points were reached and if any JavaScript was extracted.

It is often the case that multiple JavaScript statements are executed automatically during the initial processing of a file. Usually we expect to catch all of them, and thus, Adobe Reader is allowed to run until it reaches either the processing error or the processing termination tap point. However, to handle unexpected cases, we set a timeout every time the JavaScript extraction tap point is reached. This configurable timeout limits the amount of time Adobe Reader spends processing a specific JavaScript statement by terminating the process if the time limit is reached. A timeout of one second was chosen for our evaluation to allow sufficient time for execution while allowing us to process a large number of samples.

We choose to perform hot patching instead of dynamic instrumentation (e.g., Pin [27]) due to performance concerns. The Microsoft Detours library [21] was selected to modify the binary primarily because of its simplicity and ease of use, but a small amount of manual effort is required to interface this library with the tap points. In the case of the JavaScript extraction tap point, this amounts to identifying the calling convention and arguments of the target function. For the processing error and processing termination tap points we do not need to be concerned with this analysis since they cause the process to terminate immediately. Note that Adobe Reader's sandbox mechanism which prevents the process which processes PDFs from performing certain actions, such as file creation, must first be disabled.

IV. Differential Analysis

A. Experiment Setup

To evaluate the effectiveness of the reference JavaScript extractor and to identify the limitations of existing extractors, we produced two different extractors based on Adobe Reader versions 9.5.0 and 11.0.08, and then compared them against each other and several other open source tools which provide similar functionality. The libpdfjs [7] tool is the JavaScript extractor which utilizes the Poppler PDF rendering library and powers the PJScan malicious PDF detector. Origami [8] is a framework for PDF parsing, analysis, and creation which is packaged with a JavaScript extraction tool using this framework. The JavaScript unpacking tool jsunpack-n [6] attempts to extract and analyze JavaScript from many formats, but contains a module which specifically extracts JavaScript from PDFs. Lastly, the PDFiD tool is not a JavaScript extractor but merely scans a document for the appearance of certain keywords. VirusTotal uses this tool to tag samples as containing JavaScript or not [36], among other things, and the results included here for this tool are those reported by VirusTotal.

All of these tools were run against a set of 163,306 PDF files procured from VirusTotal over the month of February, 2015. Every tool evaluated was given twenty seconds to process each sample before the tool was considered to timeout and terminated. The Origami and jsunpack-n tools were run on a machine with 6GB of RAM and a 2.93GHz CPU running Ubuntu 14.04.

The most recent operating system supported by the libpdfjs tool is Ubuntu 11.04, and so it was run in a virtual machine running this version. The Adobe Reader tool must also be run inside of a Windows virtual machine, which was running XP service pack 3 specifically. Both of these virtual machines were

TABLE II: JavaScript Extractions

	Version 9.5.0					Version 11.0.08				
	Reference Extractor	**libpdfjs**	**jsunpack-n**	**Origami**	**PDFiD**	**Reference Extractor**	**libpdfjs**	**jsunpack-n**	**Origami**	**PDFiD**
Total	4397	4625	5053	4508	4398	4704	4625	5053	4508	4398
Matches	-	3940	4247	3863	3721	-	4269	4537	4167	3904
Invalid (ben./mal.)	-	7 (7/0)	26 (10/16)	23 (0/23)	-	-	0 (0/0)	16 (0/16)	23 (0/23)	-
Zero (ben./mal.)	-	450 (20/430)	124 (113/11)	511 (76/435)	676 (253/423)	-	435 (6/429)	151 (140/11)	514 (80/434)	800 (377/423)
Inconclusive	-	356	500	318	677	-	356	500	318	494

run on the same bare metal machine as the other extractors and were each given 4GB of RAM. Though this is slightly less than given the other tools, in practice, memory usage does not appear to be a limitation for any of the extraction tools.

Since the Adobe Reader tool actually executes the JavaScript extracted from a sample, we must be careful to prevent any potential exploit of Adobe Reader from having an impact on the processing of future samples. In order to prevent this, a snapshot of the virtual machine in a clean state was taken and then restored before processing each new sample. A list of hashes corresponding to certain sections of the results can be found at https://goo.gl/qtbuOC with descriptions for each.

B. *Summarized Results*

The total number of samples for which each tool extracts at least one JavaScript item and the comparison of their results against each reference extractor is listed in Table II. Notice that in the case of PDFiD, the number of samples it identifies as containing JavaScript are listed. Combined, the open-source extractors we evaluated produced JavaScript for 5250 unique samples. Our version 9.5.0 reference extractor produces JavaScript from 4397 samples of which 2956 are benign and 1441 are malicious, and our version 11.0.08 reference extractor produces JavaScript from 4704 samples, of which 3261 are benign and 1443 are malicious, when considering samples with at least 15 detections on VirusTotal as malicious. In total, all of the extractors produced JavaScript for 5267 unique samples.

Table III shows the comparison between the two different reference extractors. The files that each extractor could produce JavaScript for are largely similar and all of the extractions produced matched the similarity metric described below. Of the extractions unique to a specific reference extractor, all but two were benign, which were only produced by the version 11.0.08 extractor. Both of these samples are malformed in ways which allow for multiple interpretations, and it appears that the different versions of Adobe Reader select different ones. The large number of extractions unique to the version 11.0.08 extractor are largely caused by samples using features not supported by the older version of Adobe Reader and the extractions unique to the 9.5.0 extractor are largely caused by samples using features which are now deprecated.

It is extremely difficult to verify the correctness of these extractions due to the lack of ground truth. Therefore, we instead attempt to identify which extractions produce at least a partial match against the JavaScript discovered by the reference extractor. Even though we assume the JavaScript produced by our reference extractor is correct, it extracts only the JavaScript code that is automatically executed upon opening a PDF, which is possibly only a subset of the JavaScript in the document.

Since the other extractors attempt to produce all JavaScript embedded in the document, the JavaScript produced by a reference extractor should match at least some of the JavaScript produced by the other extractors. If such a partial match is found for a PDF sample, we call it a "match". If an extractor produces nothing for a sample from which an extraction is produced by our reference extractor, we refer to this as a "zero". If an extractor produces an extraction which does not match the extraction of our tool at all, and which is manually verified to be invalid, this is considered as an "invalid extraction". Since PDFiD only does not actually extract JavaScript, the notion of an invalid extraction does not apply to this tool.

To be safe, we do not reason about the validity of extractions produced for samples for which the reference extractor produced nothing. As a result, the amount of invalid extractions presented represents only the lower bound. We call these instances "inconclusive".

To identify matched extractions, we first look for a substring match. Since different extractors produce extractions with slight variations in whitespace, such as different end-of-line sequences, and the existing extraction tools often have trouble handling non-ASCII encodable characters, we do not consider them in this search. If a substring match cannot be found, we then use the Python difflib [4] library to efficiently compute a similarity heuristic between the extractions. If an extraction appears to have a least 50% in common with the one produced by the our tool, it is considered a match.

While in majority of the samples, the JavaScript produced by the existing extractors match that of our tool, the results do reveal that prior extractors miss JavaScript extractions for a significant amount of PDF documents. On average these extractors miss JavaScript code in 10.1% of the files and PDFiD cannot detect the existence of JavaScript in 17.01% of the samples when compared to the version 11.0.08 reference extractor. The situation is even more severe for malicious PDFs. On average, each existing extractor fails to extract JavaScript from 22.47% of the malicious samples identified as containing JavaScript by the reference extractor.

Note that the samples in the "invalid" and "zero" categories for each of the extractors are broken down into benign and malicious categories, as identified by the detectors from VirusTotal. Many of the malicious samples analyzed from these categories have obfuscations which appear to be mounting parser confusion attacks, indicating that attackers are already aware of several parser weaknesses and are actively exploiting them in an attempt to evade detection. The jsunpack-n extractor is the most effective in processing these samples, and appears to have been coded to specifically address many of the evasions.

TABLE III: JavaScript Extractions (Version 9.5.0 vs 11.0.08)

9.5.0 Only	11.0.08 Only	Common	Total
21	328	4376	4725

While these samples have been classified by many detectors on VirusTotal as malicious, the weaknesses they exploit to evade extraction are still effective against these detectors as demonstrated in Section V, indicating that this classification is likely based on factors outside of JavaScript analysis. The fact that JavaScript cannot be extracted from many benign samples either indicates that these extractors are incomplete or parse PDFs differently from Adobe Reader.

C. Tap Point Verification

Our reference extractor does not produce JavaScript from 546 samples (only 10 of which meet our definition of malicious) which at least one of the other extractors is able to. To demonstrate that the reference extractor is only unable to extract JavaScript from PDFs which do not contain automatically executing JavaScript or are malformed, and not because of a failing of the technique or because of an incorrectly selected tap point, we further perform a verification. By opening these samples with the original Adobe Reader binary in an execution monitor, we can observe the program's behavior to determine if it processes any of the JavaScript produced by the other extractors.

Each of these samples is opened by Adobe Reader and then given thirty seconds to load the EScript.api module, knowing that it contains the JavaScript engine. If the module is loaded within this time, Adobe Reader is allowed to run for another four minutes to collect a memory trace.

Of the 546 samples, only 274 actually loaded the EScript.api module. For the remaining 274 samples, we group memory operations from their traces into contiguous operations and then compare the data accessed by these operations against the extractions produced by the other tools. Of all of the contiguous operations produced in these samples, only 1006 unique matching strings were identified.

The vast majority of these strings are module or function names from the Adobe JavaScript API or JavaScript keywords. Some small JavaScript fragments were identified. Manual analysis indicates that these fragments are only substring matches between JavaScript statements produced within the module, possibly executed as part of the JavaScript engine's initialization. No complete JavaScript statement from any extraction was identified in these contiguous operations, indicating that the reference extractor has correctly captured all of the automatically executed JavaScript in well-formed PDFs.

D. Lessons

Having identified samples which contained JavaScript but which were not processed correctly by one of the extractors, we set out to determine the causes of these failings. In many cases, the cause of these failings was easily identified as an incomplete parser implementation, as many of the tools are aware of some of their limitations and will output messages indicating that they have encountered an aspect of the specification they cannot handle. In other cases, the source code of the extractor and the sample itself were manually analyzed to determine the cause of the failing, which was usually the result of a design error or an implementation bug

```
trailer << /Root 1 0 R /Size 8 >>
```

(a) Original Trailer

```
trailer << /Root %!@#!@#
1 0 R /Size 8 >>
```

(b) Trailer With Injected Comment

```
/XFA `[(config)42 0 R(template)%195 0 R
111 0 R(datasets)44 0 R(localeSet)45 0 R]’
```

(c) Abbreviated XFA Entry With Injected Comment

Fig. 3: Comment Injections

By examining the samples for which a single extractor uniquely produced the correct output, we were able to identify why the extractor was successful whereas the others were not. Thus, while only performing this analysis on a relatively small subset of these samples, we were able to identify several weaknesses in these extractors. Table IV outlines these limitations and their impacts on the JavaScript extractors, which can be generally broken down into four categories.

Implementation Bugs. Often an extractor will interpret the specification correctly but will have a programming error in its implementation. While many of these bugs and errors are quite easy to fix, their enumeration is difficult.

The PDF specification states that comments, which begin with a "%" and end with a newline sequence, shall be ignored and treated as a single whitespace character. While this aspect of the specification is straightforward, neither the jsunpack-n or Origami tools always parse comments correctly. The injection of a comment into a PDF's trailer, as seen in figures 3a and 3b, causes the Origami tool to terminate prematurely and prevent the extraction of JavaScript. Similarly, comments injected into dictionaries thwart the jsunpack-n tool. An abbreviated XML Forms Architecture (XFA) entry in a dictionary, shown in figure 3c, demonstrates this bug. In this case, jsunpack-n does not realize that the "%195 0 R" string should be ignored. Instead of looking for the object with ID "111 0" which contains the malicious payload, it in fact looks for an object "195 0" which does not exist.

Stream data is to be terminated by an end-of-line marker followed by the "endstream" keyword. However, the regular expression jsunpack-n uses to identify the stream data matches zero or more newline characters before "endstream". This means that trailing bytes in streams which happen to have values associated with newlines will incorrectly be considered whitespace instead of stream data.

The specification allows for several different encryption schemes and algorithms. Generating and applying the encryption keys in each of these algorithms is fairly complicated and can depend on several other features of the document. The

TABLE IV: Failings and Limitations

		Affected Extractors		
		libpdfjs	jsunpack-n	Origami
Implementation Bugs	Comment in trailer	✗	✗	✓
	Comment in dictionary	✗	✓	✓
	Trailing whitespace in stream data	✗	✓	✗
	Security handler revision 5 hex encoded encryption data parsing	✗	✓	✗
	Security handler revision 3, 4 encryption key computation	✗	✓	✗
	Hexadecimal string literal in encoded objects	✗	✓	✗
Design Errors	Use of orphaned encryption objects	✗	✓	✓
	Security handler revision 5 encryption key computation without encrypted metadata	✗	✓	✗
Omissions	No XFA support	✓	✗	✗
	No security handler revision 5 support	✓	✗	✗
	No security handler revision 6 support	✓	✓	✗
Ambiguities	No cross-reference table and invalid object keywords	✗	✗	✓

sets of specifications and algorithms governing how this is performed are referred to as "security handlers" with several revisions being developed as the specification has evolved. According to the PDF specification, encryption algorithms can be applied using a blank "default" password, which means that even though certain contents of the file are stored in ciphertext, any parser which correctly implements the algorithm can decrypt and examine them. Jsunpack-n appears to have interpreted the encryption key generation algorithms correctly for the revisions 3 and 4 security handlers; however, typos in the function which computes them cause the extractor to crash when they are used.

Hexadecimal string literals sometimes are not correctly handled by jsunpack-n. When these string literals are placed inside of encoded objects they are not properly parsed after the object is decoded. Additionally, jsunpack-n fails to parse hexadecimal strings that are used to store encryption data with the revision 5 security handler.

Design Errors. These are instances where the extractor appears to have interpreted the specification incorrectly or where shortcuts have been intentionally taken to simplify development. These errors are common in more complicated aspects of the specification, such as document encryption, which are hard to interpret properly or in corner cases which break the developer's assumptions.

For example, jsunpack-n and Origami scan a document for objects which define encryption parameters, and if found, use them to decrypt all content which the specification states should be encrypted. However, the incremental update mechanism in the PDF specification allows for the creation of an updated file which is no longer encrypted. Such an update does not remove old content, but produces a new document structure which stops referencing older objects. Thus, the existence of an object defining encryption parameters does not necessarily mean the current version of the document is encrypted. These extractors will then incorrectly decrypt data which is already in plain-text, producing "junk" data.

The revision 5 security handler has two slightly different key generation algorithms depending on whether or not the document's metadata is encrypted. These algorithms are not correctly interpreted by jsunpack-n. Thus, it can only produce the correct key when this metadata is encrypted.

Omissions. None of the extractors evaluated claims to have completely implemented all of the PDF specification and its extensions. By using these unimplemented aspects of the specification, it is trivial for attackers to hide malicious content from the extractor.

The libpdfjs extractor has the most omissions largely due to its dependence on an older version of the Poppler parser, which does not implement newer additions to the specification such as the revision 4 and 5 security handlers. The Poppler parser does also not support the XFA extension to the PDF specification which is often used to embed JavaScript in PDFs. While neither the Origami or jsunpack-n extractors fully support XFA, they support enough of the specification to identify and extract JavaScript embedded in this way.

Only the Origami tool supports the revision 6 security handler which is part of the PDF 2.0 specification still under development [14]. Even though this algorithm is not officially part of the PDF specification, it is still used by Adobe products and so any effective malicious PDF detector must do so also.

Ambiguities. The PDF specification is vague in certain cases, leaving space for multiple interpretations. Similarly Adobe Reader, in an attempt to "just work", will often process PDFs deviating from the specification. Since the specification does not cover these cases, it is unclear how they should be handled. Finding these ambiguities is very difficult, as well as determining how they should be handled.

For example, the PDF specification states that all PDF documents are to include a "cross-reference" table or stream which contains information about all objects in the file and their locations. If a document does not have this table, Adobe Reader and all of the extractors we evaluated will attempt to reconstruct this table by scanning the document for objects, which is usually successful when the objects in the document are well-formed.

Another ambiguity, which is often seen in malicious PDFs, is to use the malformed "objend" keyword to terminate objects. The specification states that objects should be terminated with the "endobj" keyword, but Adobe Reader and all of the existing extractors deviate from the specification by accepting both. When these two ambiguities are combined in a document which contains objects terminated with the incorrect "objend" keyword and no cross-reference table, Origami fails to identify the objects and cannot parse the document.

```
3 0 obj
<< /JS 6 0 R /S /JavaScript /Type /
     Action >>
endobj
...
6 0 obj
<< /Length 3907 >>
stream
function heapSpray(str, str_addr,
     r_addr) {
...
}
endstream
endobj
```

(a) Malicious JavaScript and reference are unobfuscated.

```
3 0 obj
<< /JS 6 0 R /S /JavaScript /Type /
     Action >>
endobj
...
6 0 obj
<< /Length 1552 /Filter /FlateDecode
     >>
stream
<encoded JavaScript>
endstream
endobj
```

(b) Malicious JavaScript is encoded, but reference is unobfuscated.

```
2 0 obj
<< /Type /ObjStm /Length 1696 /Filter
     /FlateDecode /N 4 /First 20 >>
stream
<encoded objects>
endstream
endobj
```

(c) Objects placed in streams, then encoded. Malicious JavaScript and reference are obfuscated.

Fig. 4: Stream Obfuscations

V. Parser Confusion Attacks

A. Attack Definition

By systematically studying these weaknesses of the extractors and identifying their root causes, we are able to make modifications, which we term obfuscations, to other PDF files which exploit these weaknesses and prevent JavaScript extraction. Our understanding of the parser limitations, which caused these weaknesses, also allowed us to develop new obfuscations which also prevent extraction. Since Adobe Reader's processing of the file and its execution of the embedded JavaScript is not affected, the application of these obfuscations prevents any JavaScript based malicious PDF detection while not affecting the efficacy of the exploit. We call the application of these obfuscations with the specific intent of allowing a malicious PDF to evade detection, *PDF parser confusion attacks*.

While we only specifically analyzed the weaknesses of the open source extractors which we evaluated, we were able to identify several common weaknesses in these extractors which were likely to be present in other PDF parsers. To determine the prevalence of these weaknesses and the strength of parser confusion attacks, we applied our obfuscations to a PDF containing a working exploit and then evaluated whether or not different JavaScript extractors and malicious PDF detectors were still able to correctly extract the JavaScript or classify the sample as malicious. By identifying several strong obfuscations and combining them, we were able to thwart all evaluated JavaScript extractors, all commercial AV products on VirusTotal, and the metadata based PDFrate detector.

B. Attack Construction

To demonstrate the effectiveness of these attacks in evading detection, a malicious PDF was created using a Metasploit module [3] which exploits a "use after free" vulnerability present in versions 9.0.0 to 11.0.3 of Adobe Reader [12] as the payload. The payload we embedded in this malicious sample opens a reverse shell on another machine on the network, and for each modification applied to the original sample the functionality of the exploit was verified.

The original Metasploit module generated some obfuscations which were first removed before any of ours were added, which produces a sample clearly identified as malicious and which can be correctly processed by all extractors. With the exception of embedding a comment in the document trailer, which is easily applied using a hex editor, all of these modifications were made using the open-source qpdf tool [13] or the PyPDF2 Python library [11].

In general, the aim behind these attacks is to obscure the payload of a malicious sample using aspects of the specification which a detector's parser does not support or handles incorrectly. Figure 4 outlines a simplified example demonstrating how this attack works starting with the unobfuscated malicious content in figure 4a.

By applying filters or encodings to object streams in the document, as seen in figure 4b, the malicious JavaScript is obscured from all detectors which do not support the encoding used. However, the fact that the document contains JavaScript is not hidden, which may be used with other heuristics to classify the sample as malicious. If the objects are first placed in streams before the stream data is encoded, as seen in figure 4c, no indication of the embedded malicious JavaScript remains for malware detectors which do not correctly handle these aspects of the specification.

To evaluate the effectiveness of these attacks against the commercial malware detectors on VirusTotal, we started with the base unobfuscated malicious file and then applied different obfuscations, including Flate compression, R5 & R6 security handlers, hexadecimal encoding, etc. By determining which obfuscations were successful and which detectors they evaded, we were able to combine several obfuscations so as to maximize the number of evaded detectors. Although the application of many of these obfuscations is quite easy, they are powerful in practice.

Knowing that these parser confusion attacks are likely insufficient to thwart metadata based detection systems on their own, since the core content of the sample is not changed, we mounted our parser confusion attacks in combination with a reverse mimicry attack[28]. To mount this attack, we removed the payload from the malicious sample and applied it to a benign root file, a tax form taken from the IRS' website[2]. We

[2] http://www.irs.gov/pub/irs-pdf/fw4.pdf

TABLE V: Parser Confusion Attacks on Commercial Detectors and JS Extractors

Obfuscation	**MD5 Hash**	**Detection Ratio**	**O**[1]	**l**[2]	**P**[3]	**j**[4]
None	ae91ec6a96dc4d477beba9be6b907568	30/55	✓	✓	✓	✓
Flate Compression, objects streams	eb64df4dbd733b5aa72fb0c41995f247	24/56	✓	✓	✗	✓
Flate Compression, R5 security handler	2b1071b27f96d9cdcfc59e35040d28b7	19/56	✓	✗	✓	✗
Flate Compression, R5 security handler, objects streams	8887439e33d15bcc8716634cbcbb392e	14/54	✓	✗	✗	✗
Flate Compression, R6 security handler	4e05ad44febe26f25629f27c155a7a0e	4/57	✓	✗	✓	✗
Flate Compression, R6 security handler, object streams	c82643a1388a2645409395ef3420d817	0/56	✓	✗	✗	✗
Flate Compression, R6 security handler, objects streams, comment in trailer	6b6abbce700027f7935e3eeacd43618d	0/57	✗	✗	✗	✗
JS encoded as UTF-16BE in hex string	ab09a01fe61a1066f814e3ffc2548f0a	23/55	✓	✓	✓	✓
JS encoded as UTF-16BE in hex string. Flate compression, object streams	b21e264efbb14b928f0121b22030c3a7	10/55	✓	✓	✗	✗
JS encoded as UTF-16BE in hex string, Flate Compression, R5 security handler, objects streams, comment in trailer	5039c273435300a46cd42ad0de0bb4ff	1/57	✗	✗	✗	✗

[1]**Origami** [2]**libpdfjs** [3]**PDFiD** [4]**jsunpack-n**

TABLE VI: PDFrate Evasion

Sample	**MD5 Hash**	**Contagio Malware Dump**	**George Mason University**	**PDFrate Community**
Unobfuscated malicious file	ae91ec6a96dc4d477beba9be6b907568	86.4%	89.6%	91%
Malware w/parser confusion attack only	6b6abbce700027f7935e3eeacd43618d	70%	65.8%	82.2%
Benign root file	303b209708842adf30b81f437c5ec0ed	0.7%	13.9%	13.5%
Root file w/**parser confusion + reverse mimicry attacks**	d48a343058503f931eadec99f3a89e70	7.8%	2.3%	11.0%

then applied obfuscations which were successful in thwarting the commercial malware detectors to this sample to evaluate its effect on the detector. This attack is evaluated against the PDFrate classifier[32] which has been published as a publicly available online service[9].

C. Attack Effectiveness

Commercial Detectors and JS Extractors. Table V lists a series of different parser confusion attacks which were mounted using this sample and their effects on both the detectors on VirusTotal and the JavaScript extractors which we used in our differential analysis. This table lists the "Detection Ratio" for the AV scanners, which is the number of scanners which identify the file as malicious over the number of scanners which could return a result in a timely manner. Additionally, each sample's MD5 checksum is provided which can be used to get additional information about the file and the scan from VirusTotal. Researchers with sufficient access to the VirusTotal API can also obtain copies of these files, though caution must be exercised as they do contain working exploit.

Starting with the completely unobfuscated malicious file, which the majority of the detectors identify as malicious, we started applying different obfuscations. Each parser has its own individual weaknesses, and by combining the obfuscations in different combinations, we can exploit the weaknesses in several parsers to reduce the detection ratio. We can also reduce the detection ratio by layering the obfuscations in ways which our analysis indicates a parser is unlikely to anticipate or handle correctly. By combining these techniques with the use of object streams to increase the amount of information in the document which is encoded, we are able to ensure that detection requires successful parsing while simultaneously decreasing the chances that a detector is able to do so.

While common encodings such as Flate compression only slightly reduce the detection ratio, the application of the more complicated revision 5 (R5) security handler and the obscure revision 6 (R6) handler are very effective in evading detection. Though the Origami tool correctly implements the revision 6 security handler, it is easily thwarted by injecting a comment into the document trailer, producing a sample which completely evades detection. Eventually, the sample obfuscated using the combination of Flate compression, the revision 6 security handler, object streams and the comment injected in the trailer produces zero detections on VirusTotal and can defeat all the JavaScript extractors.

Similarly, by only applying the relatively simple and well-understood UTF-16BE and hexadecimal encodings, the detection ratio is only slightly reduced. Knowing from our analysis that many parsers handle relatively simple encodings but do not correctly handle complex combinations, we combine these with Flate compression and encryption to almost completely evade detection. The fact that this can be done without the use of the unratified revision 6 security handler indicates that even within specifications which have existed for years there are still many failings which need to be addressed.

Metadata Based Detectors. Table VI shows the effectiveness of parser confusion attacks against PDFrate. We evaluated the version of this classifier which is publicly available as an online service. The classifier produces the likelihood that a sample is malicious using models trained against three different data sets, the Contagio malware dump, samples collected at George Mason University, and samples submitted by the PDFrate community [9]. No threshold is provided by PDFrate for determining whether or not a sample is malicious, but the effect of the attack is clear.

The classifier correctly identifies the unobfuscated malicious file generated by Metasploit as malicious, and though we can reduce its malicious rating using the same obfuscations which completely thwart the detectors on VirusTotal (i.e., R6 security handler, objects streams, comment in trailer), we cannot do so significantly. This is due to the fact that PDFrate does not detect malware based on the contents of

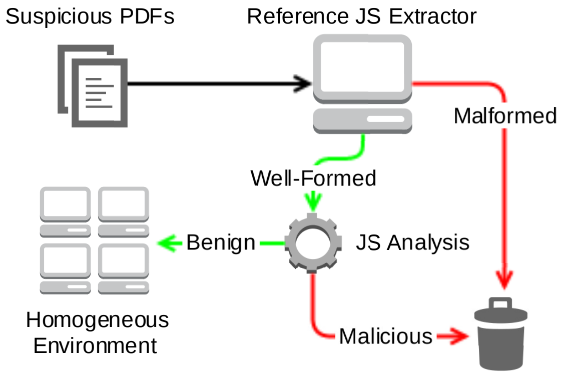

Fig. 5: Possible Deployment Scenario

embedded JavaScript but rather on the file's metadata and that these obfuscations cannot significantly alter many of the features used for classification (e.g. file size). In contrast, the application of the reverse mimicry attack produces a sample which is classified as much less malicious. However, the sample might still appear as suspicious depending on the threshold values used for classification.

By mounting a parser confusion attack in addition to a reverse mimicry attack on this file, the ability of the classifier to recognize the sample as malicious is significantly reduced, to the point where it is classified as more benign than the original sample by two of the classifiers. Since PDFrate's regular expression based parser cannot perform the computation necessary to decrypt the document, much of the document appears to the classifier to be large streams of random data. Thus, many of the features of the sample which would be used to classify the sample as malicious are obscured by the parser confusion attack.

VI. Mitigations

To mitigate parser confusion attacks, three classes of mitigation methods can be utilized.

A. *Runtime-Based Exploit Detection*

The first possible solution is to capture the JavaScript execution completely at runtime. Since parser confusion attacks only defeat static parsing and further classification efforts, it does not prevent JavaScript from being executed and thus, observed when Adobe Reader opens the hosting PDF files.

Prior malware detection systems have been presented which, though not designed to detect malicious PDFs specifically, can detect the existence of malware by analyzing the runtime behavior of the target application. Nozzle [30] and ShellOS [33], for example, have both been shown to be effective in detecting attacks against Adobe Reader specifically.

While these tools are able to completely circumvent issues related to parsing, they have significant overhead. These systems also not only depend on the execution of malicious JavaScript, but on the actual execution of an exploit in order to classify a PDF as malicious meaning that malicious JavaScript which selectively executes or fails to execute correctly cannot be correctly classified.

TABLE VII: PJScan Performance

Tool	True Positive	False Positive
Original PJScan	68.34% (1453)	0.18% (3814)
PJScan & Adobe Reader 9.5.0	96.04% (1441)	0.32% (3521)
PJScan & Adobe Reader 11.0.08	94.02% (1021)	0.20% (3677)

TABLE VIII: Average Runtime

Tool	Avg. Runtime (s)
libpdfjs	0.05
jsunpack-n	0.78
Origami	1.86
Reference JS Extractor	3.93

B. *Improvement of Parsers*

The second potential solution is to improve the existing parsers that are used in malicious PDF classifiers. Since the proposed attacks focus on the discrepancies between 3rd party PDF parsers and Adobe Reader, the attacks can essentially be defeated provided we can improve the quality of the parsers and resolve these discrepancies.

However, it is in general very difficult to make one program precisely mimic the behavior of another. Even though our reference extractor can facilitate the identification of the weakness of other PDF parsers, there can never be any guarantee that an improved parser faithfully follows the parsing logic of Adobe Reader. Even if a perfect parser could be developed, this work would have to be repeated for each version of Adobe Reader given that they have unique parsing behaviors.

C. *Deployment of Reference Extractor*

Given that the improvement of existing parsers depends on the existence of a reference extractor, it seems much more straightforward to just use the reference extractor instead. Figure 5 demonstrates the third possibility for attack mitigation, which deploys the reference JavaScript extractor along with a JavaScript based classifier in a real world scenario. Since the reference extractor can only precisely mimic the behavior of the version of Adobe Reader upon which it is based, it is best suited for controlled, relatively homogeneous environments (e.g., enterprises) in which the versions of Adobe Reader being used are known.

To demonstrate that the use of the reference extractor can improve the accuracy of existing JavaScript based classifiers, we compared the performance of the original PJScan detector, which uses libpdfjs as its extractor, against a modified version which uses the reference JavaScript extractor.

Since the PJScan tool can only classify samples which contain JavaScript we evaluated it against only the samples for which any extractor was able to produce JavaScript. While we cannot be certain that each of these samples actually contains JavaScript, the detection improvement can still be demonstrated. Since the reference extractor is also able to identify which samples are malformed, we precluded those samples in its evaluation, arguing that malformed files can be blocked without adversely affecting the end-user.

Since PJScan utilizes a One-Class Support Vector Machine which needs to be trained against a malicious set before any samples can be classified, a two-fold cross-validation was performed. Table VII shows the results of this evaluation, with the number of samples in each set shown in parenthesis (for the reference extractors these are the number of samples they considered well-formed). As can be seen, we are able to greatly improve PJScan's ability to detect malicious PDFs by using the reference extractors.

The obvious reason for this improvement is the fact that the reference JavaScript extractors are able to extract JavaScript from more of the malicious samples than libpdfjs. Of the 1453 malicious samples any tool reported as containing JavaScript, libpdfjs is only able to produce 1021 extractions whereas the version 9.5.0 and 11.0.08 reference extractors produce extractions for 1429 and 1013 out of the 1441 and 1021 malicious files they identify as well-formed, respectively.

Additionally, since only the samples which can actually be processed by each version of Adobe Reader are used to train PJScan there is less noise in the training data and a better model can be produced. For example, the versions of Adobe Reader we used do not open samples containing many older exploits which have been patched. By discarding these older malformed samples using the reference extractors, the classifier can be trained against and evaluated against only newer exploits, increasing its accuracy. This also appears to be why the version 9.5.0 extractor has slightly better performance– since we are able to filter out newer PDFs which cannot be opened by this version there is more similarity between the remaining malicious samples.

We then use our obfuscated samples to test the effectiveness of PJScan plus the reference JavaScript extractors. Results show that when PJScan is paired with the version 11.0.08 reference extractor it can now detect all the samples used in the parser confusion attacks provided a PDF containing the same malicious payload is used in the training set. When paired with the version 9.5.0 reference extractor, PJScan can detect all of these samples except those using the R6 security handler since it is not supported by that version.

Table VIII shows the average runtime for each of the evaluated JavaScript extractors for all of the samples obtained from VirusTotal. As can be expected, the Adobe Reader tool pays a significant penalty for having to restore the virtual machine to a clean state after every iteration. Note that the performance of the reference extractor is comparable to MPScan even though they do not appear to reset the system between samples [26]. The use of a reference monitor instead of dynamic hooking would also require this system reset and is significantly slower than running a VM.

In a real world implementation of this system, certain optimizations can be performed. For instance, by placing the virtual machine on a RAM disk instead of on a hard drive, we can save approximately 2 seconds on VM snapshot restoration. Since the snapshot restoration can be performed after the extraction, the latency for receiving a sample's analysis can be greatly reduced and a pipeline of analyzers could be produced to mitigate the remaining overhead.

VII. Limitations

The primary limitation of the reference extractor is its ability to only extract code which is automatically executed by Adobe Reader. Additionally, since we cannot afford to process any single sample indefinitely we can fail to extract JavaScript which delays its execution or which does not finish in the time allotted. In practice, however, these issues do not appear to be significant in terms of malicious PDF detection. Of the 10 malicious files which the reference extractor did not extract JavaScript from (out of 1453) one was malformed and the remaining nine depended on user interaction. Fundamentally these limitations are caused by the use of dynamic analysis, but as our evaluation has shown static analysis also has its own limitations.

Malware often uses "anti-VM" or "anti-sandbox" techniques to avoid detection by electing to not exhibit malicious behavior in virtual environments. Although we are unaware of any such techniques used by malicious PDFs and think that the limited amount of information about the system available through the Adobe Reader JavaScript API would make them difficult to implement, we cannot claim that such checks are impossible. For example, it might be possible for an advanced attacker to test if the sandbox is disabled, which is required for reference extractor to function. However, any such check would have to depend on the execution of some JavaScript which would be extracted and could be used to classify the document as malicious or at least suspicious.

VIII. Conclusion

In this paper, we conducted a systematic study on a new evasion technique called a *PDF parser confusion attack*, which aims to confuse the PDF parsers in malware detectors in order to evade detection. To enable a systematic study we have developed a *reference JavaScript extractor* by directly tapping into Adobe Reader and presented a mostly-automatic technique for developing it. By delving into these discrepancies between the reference extractor and the existing extractors we have identified several new obfuscations and further quantified their impact when used in parser confusion attacks on JavaScript extractors and malware detectors. By combining several of these obfuscations, we produced a malicious PDF which can successfully evade *all* the malware detectors evaluated, including signature-based, structure/metadata-based, and JavaScript-based detectors. To address parser confusion attacks, we discuss several mitigation techniques. In particular, we demonstrate that with our reference JavaScript extractor the detection rate of an existing classifier has increases significantly from 68% to 96% on our sample set and present a possible deployment scenario for the reference extractor.

Acknowledgement

We would like to thank anonymous reviewers and our shepherd Dr. Guofei Gu for their insightful feedback. This research was supported in part by National Science Foundation Grant #1054605, Air Force Research Lab Grant #FA8750-13-2-0115 and #FA8750-15-2-0106, and DARPA Grant #FA8750-14-C-0118. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] "Adobe acrobat reader : Security vulnerabilities published in 2014," http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-497/year-2014/Adobe-Acrobat-Reader.html.

[2] "Adobe acrobat reader : Security vulnerabilities published in 2015," http://www.cvedetails.com/vulnerability-list/vendor_id-53/product_id-497/year-2015/Adobe-Acrobat-Reader.html.

[3] "CVE-2013-3346 Adobe Reader ToolButton Use After Free," http://www.rapid7.com/db/modules/exploit/windows/browser/adobe_toolbutton.

[4] "difflib – helpers for computing deltas," https://docs.python.org/2/library/difflib.html.

[5] "Introducing adobe reader protected mode," http://blogs.adobe.com/security/2010/07/introducing-adobe-reader-protected-mode.html.

[6] "jsunpack-n," http://code.google.com/p/jsunpack-n/.

[7] "libpdfjs," http://sourceforge.net/projects/libpdfjs/.

[8] "origami," https://code.google.com/p/origami-pdf/.

[9] "PDFrate A machine learning based classifier operating on document metadata and structure," http://pdfrate.com/.

[10] "Poppler," http://poppler.freedesktop.org/.

[11] "PyPDF2," https://github.com/mstamy2/PyPDF2.

[12] "Vulnerability Details : CVE-2013-3346," http://www.cvedetails.com/cve/2013-3346.

[13] J. Berkenbilt, "QPDF: A Content-Preserving PDF Transformation System," http://qpdf.sourceforge.net/.

[14] G. Delugré, "The undocumented password validation algorithm of adobe reader x," http://esec-lab.sogeti.com/post/The-undocumented-password-validation-algorithm-of-Adobe-Reader-X.

[15] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee, "Tappan zee (north) bridge: mining memory accesses for introspection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 839–850.

[16] P. Fogla and W. Lee, "Evading network anomaly detection systems: Formal reasoning and practical techniques," in *Proceedings of the 13th ACM Conference on Computer and Communications Security*, 2006.

[17] P. Fogla, M. Sharif, R. Perdisci, O. Kolesnikov, and W. Lee, "Polymorphic blending attacks," in *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, 2006.

[18] M. Garnaeva, V. Chebyshev, D. Makrushin, R. Unuchek, and A. Ivanov, "Kaspersky security bulletin 2014." http://securelist.com/analysis/kaspersky-security-bulletin/68010/kaspersky-security-bulletin-2014-overall-statistics-for-2014/.

[19] D. Goodin, "It's official: Adobe reader is world's most-exploited app," http://www.theregister.co.uk/2010/03/09/adobe_reader_attacks/, 2010.

[20] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: Building a platform-neutral whole-system dynamic binary analysis platform," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 248–258. [Online]. Available: http://doi.acm.org/10.1145/2610384.2610407

[21] G. Hunt and D. Brubacher, "Detours: Binary interception of win32 functions," in *Third USENIX Windows NT Symposium*. USENIX, July 1999, p. 8. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=68568

[22] S. Jana and V. Shmatikov, "Abusing file processing in malware detectors for fun and profit," in *Proceedings of the 2012 IEEE Symposium on Security and Privacy*, ser. SP '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 80–94. [Online]. Available: http://dx.doi.org/10.1109/SP.2012.15

[23] P. Laskov and N. Šrndić, "Static Detection of Malicious JavaScript-bearing PDF Documents," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 373–382. [Online]. Available: http://doi.acm.org/10.1145/2076732.2076785

[24] M. Lee, "GNU PDF project leaves FSF High Priority Projects list; mission complete!" https://www.fsf.org/blogs/community/gnu-pdf-project-leaves-high-priority-projects-list-mission-complete.

[25] D. Liu, H. Wang, and A. Stavrou, "Detecting Malicious Javascript in PDF Through Document Instrumentation," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, ser. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 100–111. [Online]. Available: http://dx.doi.org/10.1109/DSN.2014.92

[26] X. Lu, J. Zhuge, R. Wang, Y. Cao, and Y. Chen, "De-obfuscation and Detection of Malicious PDF Files with High Accuracy," in *46th Hawaii International Conference on System Sciences, HICSS 2013, Wailea, HI, USA, January 7-10, 2013*, 2013, pp. 4890–4899. [Online]. Available: http://dx.doi.org/10.1109/HICSS.2013.166

[27] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005.

[28] D. Maiorca, I. Corona, and G. Giacinto, "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*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 119–130. [Online]. Available: http://doi.acm.org/10.1145/2484313.2484327

[29] D. Maiorca, G. Giacinto, and I. Corona, "A Pattern Recognition System for Malicious PDF Files Detection," in *Proceedings of the 8th International Conference on Machine Learning and Data Mining in Pattern Recognition*, ser. MLDM'12. Berlin, Heidelberg: Springer-Verlag, 2012, pp. 510–524. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-31537-4_40

[30] P. Ratanaworabhan, B. Livshits, and B. Zorn, "Nozzle: A defense against heap-spraying code injection attacks," in *Proceedings of the Usenix Security Symposium*. USENIX, 2009. [Online]. Available: http://research.microsoft.com/apps/pubs/default.aspx?id=81085

[31] M. Z. Shafiq, S. A. Khayam, and M. Farooq, "Embedded malware detection using markov n-grams," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2008.

[32] C. Smutz and A. Stavrou, "Malicious PDF Detection Using Metadata and Structural Features," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12. New York, NY, USA: ACM, 2012, pp. 239–248. [Online]. Available: http://doi.acm.org/10.1145/2420950.2420987

[33] K. Snow, S. Krishnan, F. Monrose, and N. Provos, "Shellos: Enabling fast detection and forensic analysis of code injection attacks," in *USENIX Security Symposium*, 2011. [Online]. Available: http://static.usenix.org/events/sec11/tech/full_papers/Snow.pdf

[34] Y. Song, M. E. Locasto, A. Stavrou, A. D. Keromytis, and S. J. Stolfo, "On the infeasibility of modeling polymorphic shellcode," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, 2007.

[35] D. Stevens, "PDF Tools," http://blog.didierstevens.com/programs/pdf-tools/.

[36] ——, "PDFiD On VirusTotal," http://blog.didierstevens.com/2009/04/21/pdfid-on-virustotal/.

[37] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos, "Combining static and dynamic analysis for the detection of malicious documents," in *Proceedings of the Fourth European Workshop on System Security*, ser. EUROSEC '11. New York, NY, USA: ACM, 2011, pp. 4:1–4:6. [Online]. Available: http://doi.acm.org/10.1145/1972551.1972555

[38] N. Šrndić and P. Laskov, "Detection of malicious PDF files based on hierarchical document structure," in *In Proceedings of the Network and Distributed System Security Symposium, NDSS 2013*. The Internet Society, 2013.

[39] N. Šrndic and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, ser. SP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 197–211. [Online]. Available: http://dx.doi.org/10.1109/SP.2014.20