Unacceptable Behavior: Robust PDF Malware Detection Using Abstract Interpretation

Alexander Jordan
François Gauthier
Behnaz Hassanshahi
Oracle Labs
Brisbane, Australia
firstname.lastname@oracle.com

David Zhao
Oracle Labs
Brisbane, Australia
University of Sydney
Sydney, Australia
d-z@outlook.com

ABSTRACT

The popularity of the PDF format and the rich JavaScript environment that PDF viewers offer make PDF documents an attractive attack vector for malware developers. PDF documents present a serious threat to the security of organizations because most users are unsuspecting of them and thus likely to open documents from untrusted sources.

State-of-the-art approaches use machine learning to learn features that characterize PDF malware, which makes them subject to adversarial attacks that mimic the structure of benign documents. In this paper, we instead propose to detect malicious code inside a PDF by statically reasoning about its *possible behavior* using abstract interpretation. A comparison with state-of-the-art PDF malware detection tools shows that our conservative abstract interpretation approach achieves similar accuracy, is more resilient to evasion attacks, and provides interpretable reports.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

Abstract interpretation, PDF, Malware, JavaScript

ACM Reference Format:
Alexander Jordan, François Gauthier, Behnaz Hassanshahi, and David Zhao. 2019. Unacceptable Behavior: Robust PDF Malware Detection Using Abstract Interpretation. In *14th ACM SIGSAC Workshop on Programming Languages and Analysis for Security (PLAS'19), November 15, 2019, London, United Kingdom.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3338504.3357341

1 INTRODUCTION

The Portable Document Format (PDF) allows for the embedding of interactive elements written in JavaScript. In PDFs, JavaScript allows document creators to support input validation in forms and to offer convenient shortcuts for common actions such as printing.

PLAS'19, November 15, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6836-0/19/11…$15.00
https://doi.org/10.1145/3338504.3357341

More elaborate use cases of PDF JavaScript[1] include controlling embedded multimedia objects and interacting with the file system or network. However, this rich and complex PDF JavaScript environment can also be used for illegitimate purposes. Indeed, previous work has shown that JavaScript is the vector of choice for PDF malware because: (1) implementation bugs in the PDF JavaScript extensions can be exploited to deliver and execute malicious payloads; (2) bugs in the JavaScript runtime and/or sandbox can be triggered with JavaScript code; and (3) JavaScript can be used as a facilitator to exploit vulnerabilities *outside* the JavaScript environment through techniques such as heap spraying [17, 37].

In 2008 and 2009 the number of Common Vulnerabilities and Exposures (CVEs) reported against the Adobe Reader rose alarmingly. Previous work showed how the vast majority of PDF malware uses JavaScript in one way or another [15, 17, 37]. The histogram in Figure 1 shows how the number of CVEs reported against Adobe Reader is still at all-time highs, with 175 CVEs already reported at the time of writing (July 2019), despite the introduction of sandboxing for increased safety in Adobe Reader X.

To lower the risk posed by PDF malware, several well-known static and dynamic analysis techniques are available. While most commercial tools still heavily rely on malware signatures, the research community has developed several approaches based on

[1] While the major parts of the PDF format are standardized as ISO 32000-1 [14], the specification of advanced features supported by Adobe's PDF software remains proprietary technology, and is referenced only by the ISO standard. The proprietary parts include JavaScript support and APIs available in Adobe's PDF software. Adobe provides an informal specification for this JavaScript part in their public documentation [12, 13].

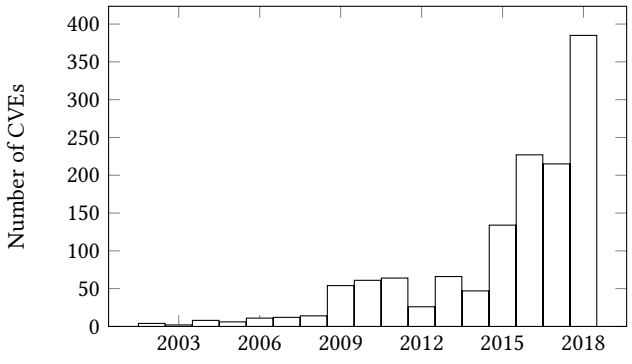

Figure 1: Number of CVEs per year containing "Adobe.*Reader" in their description.

dynamic analysis [40, 52, 53, 56, 59, 61, 62] and machine learning [43, 46, 47, 54, 58] to detect and report PDF malware.

Unsurprisingly, as a result of the ongoing arms race between malware developers and security experts, evasion approaches have been developed for each of the above techniques. Modern malware commonly uses packing and obfuscation to evade signature-based detection, and monitors its runtime environment to conceal its malicious behavior when executed in a sandboxed environment [23]. More recently, a plethora of adversarial machine learning approaches have been developed to evade learning-based detectors [16, 26, 57, 63]. As a result, the security community has developed various approaches to counter evasions through de-obfuscation and sandbox abstraction [19], and is now investigating ways to produce robust classifiers in the presence of adversarial inputs [25, 27, 49, 55].

While we strongly believe that research on detection, evasion, and counter-measures is beneficial to security, we also observe that all existing approaches for PDF malware detection suffer from the same fundamental limitation: they do not reason about the malicious code's semantics. Indeed, signature-based approaches rely on malware analysts inspecting the code and producing signatures that capture *structural* features of malware, a costly short-term solution. Dynamic approaches monitor side-effects of malicious code execution (e.g. network and file system access) but are limited to behavior observed in a special setting (e.g., a sandbox). Finally, the few machine learning approaches that reason about malicious code consider lexical patterns (e.g. n-grams) only, and can be easily evaded with packing and obfuscation. In an attempt to alleviate these limitations, recent machine learning approaches consider file structure and meta-data features, but fall short to evasion attacks that mimic benign documents, i.e., adversarial machine learning.

To address the limitations of signature-based, dynamic, and machine learning tools, we introduce *SAFE-PDF*, an abstract interpretation-based JavaScript malware detector for PDF documents. Abstract interpretation is a powerful framework for static program analysis that computes a sound over-approximation of all possible program behaviors. In other words, abstract interpretation allows us to check whether a JavaScript program can *ever* exhibit malicious behavior under *any* possible execution. Once the JavaScript code is made available to *SAFE-PDF*, there is very little malware developers can do to hide the malicious nature of their payload. This is confirmed by *SAFE-PDF*'s extremely low evasion rate.

This paper makes the following contributions:

- We present a robust and efficient approach to detecting PDF malware by analyzing the *behavior* of PDF-embedded JavaScript code in less than four seconds, on average.
- We develop one of the most robust PDF JavaScript code extractors to date, and test it on PDF documents that are *known* to defeat state-of-the-art extractors [17].
- We show how *SAFE-PDF* achieves a false positive rate and recall comparable to state-of-the-art PDF malware detectors, while detecting malware that *evades* these detectors.
- We show how *SAFE-PDF* produces highly interpretable results that help security practitioners to: understand how PDF malware operates, and troubleshoot and tune *SAFE-PDF*.

2 MOTIVATION

PDF malware frequently relies on embedded JavaScript code to carry its malicious payload. However, state-of-the-art approaches to PDF malware detection do not reason about the *semantics* of the embedded code; they reason about auxiliary features of the document only. For this reason, they are vulnerable to adversarial attacks where malicious payloads are injected in documents exhibiting features that mimic benign documents [41, 42, 57]. As Maiorca et al. aptly pointed out: "Looking at the Bag is not Enough to Find the Bomb". In this paper, we investigate how abstract interpretation can help reason about the semantics of embedded JavaScript code in PDF documents.

Obviously, before any analysis of the embedded code can occur, it must be extracted from the PDF document, a task that is not as simple as it may appear. Indeed, the very complex nature of the PDF format allows attackers to hide their JavaScript payloads in many different ways. Previous research showed how JavaScript-based detectors could be evaded solely through clever use (and abuse) of the PDF format to hide embedded code from the extractor. Due to code extraction being the main challenge for *SAFE-PDF*, we invested significant efforts into developing one of the most robust PDF JavaScript code extractors to date, and validated it against 2952 documents that are *known* to cause issues in existing code extractors [17]. Section 4.1 details the PDF format, and highlights the "features" that must be supported to extract JavaScript code from PDF malware. Once the code is extracted, however, there is very little malware developers can do to hide their payload because our abstract interpreter will undo any JavaScript-based obfuscations.

An additional benefit of performing abstract interpretation of embedded JavaScript code is that the analysis naturally provides explanations. Indeed, when a malicious operation is detected, abstract interpretation can produce an exact sequence of operations and a set of (approximated) inputs that triggered it. Due to the nature of our analysis, false positive reports are possible, in which case, the analysis result can still provide useful insight into what triggered the false positive (e.g., an incorrect over-approximation in the model, or an implementation bug in the embedded JavaScript code). This allows for ongoing and targeted refinement of the analysis configuration, without the need for retraining. In a context where security practitioners tend to distrust machine learning approaches because of their black-box nature, being able to produce highly interpretable explanations is a huge advantage. While many recent research works address the problem of explaining classifiers' decisions, the produced explanations are often imprecise [30].

Higher robustness to evasion attacks and explicability, however, come at the price of lower performance, especially when compared to machine learning-based approaches that leverage heavily optimized, production-grade machine learning toolkits. While our prototype was not optimized for performance, it seems unrealistic that abstract interpretation will ever reach the same performance as a *trained* classifier. For this reason, we believe *SAFE-PDF* could be used in combination with other tools, in situations where: (1) resilience to evasion is paramount; (2) other tools cannot reach a definitive conclusion; and (3) it helps analysts understand new malware.

```
function urpl(sc) {
    var keyu = "%u";
    var re = /XY/g;
    sc = sc.replace(re, keyu);
    return sc;
}
var unes = unescape
var pGvRIJZpqdN
for (i = 0; i < 18000; i++)
    pGvRIJZpqdN = pGvRIJZpqdN + 0x77;
var s = "XY104CXY106FXY1072XY1065XY106DXY1020XY" +
 ↪  "1069XY1070XY1073\x75XY106DXY1020XY1064" +
 ↪  "XY106FXY106CXY106FXY1072XY1020XY1073XY" +
 ↪  "1069XY1074XY1020XY1061XY106DXY1065XY10" +
 ↪  "74\x25XY1020XY1063XY106FXY106EXY1073XY" +
 ↪  "1065XY1063XY1074XY1065XY1074\x75XY1072" +
 ↪  "XY1020XY1061XY1064XY1069XY10. . . ";
pGvRIJZpqdN = unes(urpl(s));
```

Listing 1: Artificial malware example: obfuscated binary payload

2.1 Complementing existing approaches

The most common way for anti-virus software to identify PDF malware is to search files for signatures or patterns of known malware. While fast and computationally cheap, signature-based methods are easily evaded through packing and obfuscation. PDF-based packing uses hard-to-analyze features of the PDF format to prevent extraction of the JavaScript code. Note that previous work refers to PDF-based packing as "parser confusion" attacks [17], and hence we will use this term throughout the rest of this paper. Apart from parser confusion attacks, the JavaScript language itself also offers many possibilities for obfuscation. Listing 1 shows an example that aliases and composes function objects, uses string replacement, and uses the built-in `unescape` function to decode its malicious payload.

To overcome the limitations of signature-based techniques, lexical [37, 60], metadata-based [54], and structure-based [43, 58] learning approaches have been proposed. These approaches mainly differ in the way they perform feature extraction. Lexical approaches tokenize the JavaScript code and use n-grams in an attempt to distinguish between benign and malicious JavaScript code. Metadata-based approaches use features such as file size, number of JavaScript components, or number of embedded fonts to detect malicious documents. Finally, structure-based approaches classify documents based on paths in the PDF document tree. Because machine learning approaches are limited to learning which features of the embedded code or whole document are *correlated* with malicious behaviors, they are susceptible to adversarial attacks where a malicious payload is embedded in an environment that *exhibits* mostly benign features [42, 57].

Sandbox-based techniques, on the other hand, monitor the execution of a PDF viewer application opening a suspicious file in an attempt to detect malicious behaviors at runtime [39, 40, 59]. The Cuckoo sandbox [3], for example, can log system calls and network traffic, and perform memory analysis in an attempt to detect traces of malicious behavior. While dynamic analysis techniques are oblivious to parser confusion and obfuscation, they are inherently limited by the fact that they target a specific runtime environment. Hence, they may miss malicious behavior due to simple checks that probe the environment,

Table 1: Detection approaches vs. evasion techniques (✗ indicates that an approach is vulnerable to an evasion).

	Signature	Sandbox	ML	Abs.int.
Parser confusion	✗		✗†	✗
Obfuscation	✗		✗†	
Sandbox detection		✗		
Adversarial attacks			✗	

† Lexical features-based learning only.

To summarize, Table 1 associates existing approaches with evasion techniques. For the sake of clarity, we distinguish between evasions based on parser confusion and obfuscation. While all approaches can be evaded, we argue that parser confusion, which *SAFE-PDF* is vulnerable to, is probably less evil (conservative treatment of parsing errors and moving towards strict parsing as proposed by the *Caradoc* approach [29] are possible remedies). Indeed, for a parser confusion attack to be effective, the runtime under attack must still be able to *parse* the input (document). Hence, an extractor that faithfully reproduces the parsing functionality of the runtime completely thwarts parser confusion-based evasions. Obfuscation, sandbox detection and adversarial attacks, on the other hand, allow for infinite variations of a single malware, which has led to a seemingly never-ending arms race. Table 1 makes it clear, however, that no single approach is bulletproof, and we strongly believe that a comprehensive solution to malware detection should use a combination of tools with orthogonal strengths.

We propose the use of *conservative* abstract interpretation of JavaScript as a static analysis to detect malware in PDF documents. By means of abstract interpretation, *SAFE-PDF* hits a sweet spot in the analysis landscape where it can *statically* consider all possible executions of the JavaScript code, and detect malicious behavior without relying on a special runtime environment or requiring any user interaction. By conservative, we mean that when in doubt, our analysis will err on the safe side. In other words, we are willing to accept that our analysis may regard a harmless PDF as malicious (i.e., a *false positive*), but we do not accept the opposite (i.e., a *false negative*).

3 BACKGROUND

Abstract interpretation is a mathematically well-founded framework for static analysis introduced by Cousot and Cousot in [21]. It addresses the challenge of computing non-trivial properties of a program, which is known to be undecidable when the concrete language semantics is used (c.f., Rice's theorem [31, chapter 9]). When concrete values and operations are approximated with abstract values and abstract operations, however, such an abstract interpretation of a program becomes computable. This comes at the cost of losing precision for some properties of a program due to the abstraction (approximation) that is applied. In the context of static analysis, such a (partially) imprecise result means that some of our questions about a program (e.g., *"is it malicious?"*) have to

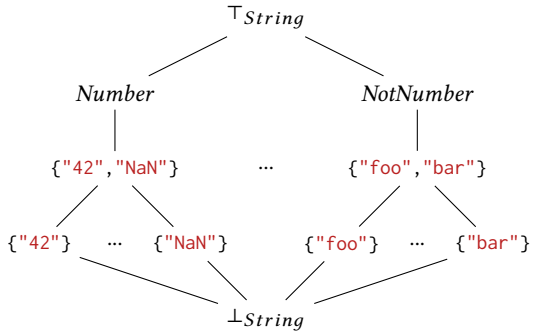

Figure 2: SAFE's string abstract domain. It keeps track of sets of, at most k, different (number-) strings before approximating them as *Number* or *NotNumber*. ($k = 2$ in this figure.)

be answered with *"We don't know"* for the analysis to be sound. In the following paragraphs we explain some of the concepts behind abstract interpretation that are required later on, and we give a step-by-step example. A formal introduction to abstract interpretation can be found in [44, chapter 4].

Abstract interpretation From a program analysis point of view, abstract interpretation gives us the ability to statically (i.e., without running the program) build an abstract state for every point in the program. These states capture information about possible concrete executions and we can use them to validate assertions or find problems in the programs we want to analyze. It is up to us, as the designer of the abstract interpretation analysis, to choose an appropriate abstraction. An abstraction consists of (1) abstract domains and (2) abstract semantics. The abstract domains capture the program states in our analysis as abstract values. It is important to note that an abstract value must be able to represent more than one concrete value at a time. A domain maintains information in the form of a *lattice*, which may have up to infinite elements and infinite height. (See Figure 2 for an example of the abstract domain for string values used by *SAFE-PDF*; this is a simple *powerset lattice* ordered by *set inclusion*.) Depending on the abstraction function, a domain can capture information about possible concrete values using wide approximations. For example, if we were interested only to find out whether a numeric value can be negative, we could abstract an integer value as its sign; or we abstract it as a set or a range of concrete integer values. We also need to encode the semantics of abstract operations, which approximates concrete operations on abstract values. Given an abstraction, we compute a fixpoint for a program in the abstraction. Note that in practice, an analysis might not reach a fixpoint for all programs within feasible time and memory bounds.

Example We present a simple JavaScript code snippet in Listing 2. Lines that start with `//` show updates to the abstract state after every instruction. The `var` keyword at line 1 creates a local variable, which our abstract state stores directly, together with the right-hand side string-value, in the current *local* scope. The array `arr` (created at line 3) contains two function objects, which are stored in the abstract heap at address *#1* and *#2* respectively. The abstract array object itself is stored at address *#3* and referenced

```
1  var msg = "hello world";
2  // local: msg := "hello world"
3  var arr = [
4    function(s) { console.println(s); },
5    function(s) { doc.getField("msgField").value = s; }
6  ];
7  // heap: #1 := Func
8  //       #2 := Func
9  //       #3 := Obj { "0" := #1, "1" := #2, length := 2 }
10 // local: arr := #3, msg
11 var i = doc.getField("inputField");
12 // local: arr, i := ⊤String, msg
13 var n = Number.parseInt(i);
14 // local: arr, i, msg, n := ⊤Number
15 var fn = arr[n];
16 // local: fn := {#1, #2, undefined}
17 if (fn === undefined)
18   // local: fn := {undefined}
19   app.alert("Unexpected input: " + i);
20   // calls app.alert()
21 else
22   // local: fn := {#1, #2}
23   fn(msg);
24   // calls either #1 or #2
```

Listing 2: PDF JavaScript snippet with abstract interpretation state in comments

by a new variable in *local*. The `arr` object contains three properties: *"0"* and *"1"* point to the function objects at their respective addresses; the internal *length* property approximates the number of items contained in an array (or any JavaScript object). The next two instructions (lines 11 and 13) call a PDF API and JavaScript built-in function respectively. The call to `getField` returns a user input that cannot be known statically. But based on the specification of `getField`, our abstract interpretation can approximate the returned value with the abstract value $\top_{String}$, which represents *any string*. This value is then passed to `parseInt` (l. 13), for which our analysis has a semantic model: it returns an integral number if the string argument can be parsed and `NaN` otherwise. In this case, for *any string* as input, it returns $\top_{Number}$, i.e., *any number*. The resulting value stored in `n` is then used in a property lookup on the array object `arr` (l. 15). The information in our abstract state at this point is precise enough to give a close approximation for the lookup with a key value of $\top_{Number}$, which is converted to a string value during lookup according to JavaScript semantics. Consequently, the values with property names *"0"* and *"1"* match the lookup because they are number strings, and are thus returned as results to `fn`. The value `undefined` is also returned because $\top_{Number}$ includes numbers that are not in *arr* (e.g., 2, 42.3, ...). Note, however, that because $\top_{Number}$ matches number strings only, the lookup does not match *length* or any of the internal array functions (e.g. `find()`), which can be accessed from a JavaScript array object by following its prototype chain. To approximate the control-flow behavior of the if-statement starting in line 17, we perform its test against our abstract state. Abstract interpretation determines that because `fn` may be `undefined`, the then-branch (l. 19) can be taken, and the call to `app.alert` may be executed. In the else-branch (also reachable

due to the abstract value of `fn`), abstract interpretation can remove `undefined` from the possible values of `fn` (before reaching line 23) because it contradicts the if-condition. This proves that the call to the function object `fn`, can invoke only the two functions defined inside `arr`. It cannot call any other function, and cannot fail due to `fn` not being a function object.

4 MALWARE DETECTION

SAFE-PDF starts by extracting the JavaScript code from the input document. Then, it complements the extracted code with a model of the JavaScript runtime environment inside a PDF viewer following Adobe's specification [12, 13], hereafter called the *PDF-JS* model, and performs abstract interpretation. If abstract interpretation does not complete (i.e., it cannot reach a fixpoint) within a certain time, the document is immediately reported as malicious. If abstract interpretation terminates, *SAFE-PDF* then checks if the document exhibits potentially malicious behavior. If so, the document is reported as malicious. Otherwise, it is reported as benign.

4.1 Pre-processing step: code extraction

Our approach requires JavaScript code to be extracted from PDF documents before it can be analyzed. While conceptually simple, the PDF format makes extraction extremely tricky. Indeed, JavaScript code can be embedded in different PDF constructs, encoded with various uncommon encodings, compressed, and encrypted, meaning that JavaScript code extraction requires a full-fledged PDF parser. Moreover, Carmony et al. [17] showed that PDF viewers often deviate from the specification, in an attempt to "just work", and that existing open-source and commercial JavaScript code extractors all fail to extract code from various PDF documents. As a result of their work, they compiled a set of 2952 PDF documents that are known to cause extraction issues in one or more JavaScript code extractors. Starting from those documents with hard-to-extract JavaScript code, we extended an existing commercial extractor [5] until it could successfully extract JavaScript code from *all* documents in the set. Because static code extraction can reach code that might not be loaded dynamically (e.g form actions that are only triggered when interacting with the document), we claim that our approach analyzes a strict superset of the code that can be extracted by dynamically loading a PDF document in a sandbox.

In the following paragraph, we briefly introduce the PDF format, focusing only on elements that are relevant to code extraction. We refer the interested reader to [29] for an extensive description of the Portable Document Format (PDF). For code extraction purposes, the four most important elements of the PDF syntax are: (1) direct objects, which are the basic building blocks of a PDF; (2) indirect objects, which are uniquely identified, and can be referenced from elsewhere in the document; (3) cross-reference tables, which contain the positions of objects in the file; and (4) content streams, which store various parts of the document content. Content streams are composed of two parts: a stream that is an optionally compressed and encrypted byte sequence, and a meta-data dictionary object that carries information about the stream's encoding and how to uncompress it. Because Adobe Reader can cope with partially broken compressed streams, and unspecified encodings, we extended our extractor with the same capabilities.

Table 2: Extractor limitations and associated PDF constructs

Limitation	Problematic PDF construct
Implementation Bugs	Comment in document trailer
	Comment in dictionary object
	Trailing whitespace in stream data
	Null object reference
	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 key computation with clear metadata
Omissions	No XFA support
	No security handler revision 5 support
	No security handler revision 6 support
Ambiguities	Invalid object keywords
	No cross-reference table
	Wrong or missing entries in the cross-reference table
	Partially broken compressed streams

Furthermore, because JavaScript code in PDF documents is usually broken into snippets and spread across several content streams, a code extractor must not only extract the various snippets from streams, it must also parse the document in order to recover its structure and reassemble the snippets into a semantically valid program. In [17], the authors list several constructs that are known to cause PDF parser failures and extraction errors. We report those constructs in Table 2, along with additional problematic constructs we addressed in our extractor (in bold). In Table 2, security handlers refer to various encryption algorithms that can be used to encrypt streams, and XFA refers to the XML Forms Architecture, which is supported by the PDF specification, and that allows the embedding of JavaScript actions in XML forms. After meticulous extensions, our extractor now supports all constructs listed in Table 2 and extracts JavaScript code from *all* of the original 2952 PDF documents with hard-to-extract JavaScript code that contain non-empty JavaScript code and that do not cause our extractor to fail. Indeed, during manual investigation of the documents with no extracted JavaScript, we observed that all of them contain an empty JavaScript string (e.g. `/JS()/S/JavaScript`). We believe that because the approach in [17] detects JavaScript code by monitoring loads of the `EScript.api` module, which would be triggered by the `/JavaScript` keyword, it mistakenly tags those files as containing

JavaScript code. We also observed that most of the hard-to-extract files that cause extraction failure are so broken that they cannot be opened with Acrobat Reader DC 2018.011. Hence, we are highly confident that our extractor retrieves all the JavaScript code that can be extracted.

4.2 Main analysis step: abstract interpretation

During the main analysis step, we perform abstract interpretation of the extracted JavaScript code. As secondary input, we provide a model of the JavaScript runtime environment emulating that of a concrete PDF viewer, which we refer to as our *PDF-JS model*. The role of the PDF-JS model is to provide extracted JavaScript code with an abstract environment for analysis emulating that of a PDF viewer application. It thus captures (a subset of) the PDF-JavaScript specification [12, 13]. For example, the global static objects app and doc are made available as part of the JavaScript environment according to Adobe's documentation. Unlike a concrete JavaScript-based environment, which would be used for dynamic analysis, our model can make use of abstract semantics (i.e. not all API functions must provide concrete results), as long as it remains conservative, i.e., it must not under-approximate the behavior of the JavaScript API. We present the PDF-JS model in detail in Section 4.4.

To support JavaScript code in XFA and its interactions with objects specified as XML entities, we provide additional modeling. In accordance with available documentation [4], the analysis extracts and dynamically models XFA entities as JavaScript objects and, using the same principles as the PDF-JS model, provides an environment to analyze XFA JavaScript code.

The result we receive from this analysis step must represent an over-approximation of the JavaScript code's behavior. In situations where abstract interpretation cannot reach a fixpoint, no valid over-approximation is available, and thus the analysis immediately reports a potential malware. Causes for not reaching a fixpoint are: (1) the analysis reaching a timeout or exhausting the available memory; and (2) syntactic or semantic errors in the extracted JavaScript code causing the analysis to fail. Only if the analysis reaches a fixpoint, we pass on the result of abstract interpretation to the final *whitelisting* step.

4.3 Post-processing step: semantic whitelist

The last step of our static analysis classifies the extracted JavaScript as either *safe* or *malicious*. This is done by inspecting the result of abstract interpretation and whitelisting documents that exhibit *known* safe behaviors only. To be conservative, *SAFE-PDF* has to reject a PDF document as malicious if abstract interpretation of its extracted JavaScript code yields any of the following:

(1) A call to a vulnerable API method
(2) A potentially malicious program behavior:
  (a) string length exceeding a predefined limit
  (b) object (array) size exceeding a predefined limit
(3) An *unknown behavior*.

We detect the use of vulnerable APIs (1) by building a PDF-JS model (see Section 4.4), which, through semantic modeling, selectively whitelists those API methods known *not* to be vulnerable. As a result, any call to a non-whitelisted method is detected as malicious. To detect (2), we limit values (strings and objects) on the abstract heap to a predefined size. Based on our evaluation, exceedingly large values are always used to perform heap spraying, in order to exploit memory corruption in the language runtime or PDF viewer. Finally (3), because *SAFE-PDF* is conservative, it reports any code as malware that exhibits *unknown behavior*, which makes it impossible to prove the absence of calls to vulnerable API methods or malicious program behavior. For example, calls to imprecise function objects (due to aliasing or function lookup using an unknown input value, e.g., a $\top_{String}$ value) cause unknown behavior.

```
function PDF_Event() {
   this.type = ⊤String
   this.name = ⊤String
   this.change = ⊤String
   this.changeEx = ⊤String
   this.commitKey = ⊤Number
   this.fieldFull = ⊤Bool
   this.keyDown = ⊤Bool
   this.modifier = ⊤Bool
   this.rc = ⊤Bool
   this.selEnd = ⊤Number
   this.selStart = ⊤Number
   this.shift = ⊤Bool
   this.source = PDF_DOM_NODE
   this.target = PDF_DOM_NODE
   this.targetName = PDF_DOM_NODE.name
   this.value = ⊤String
   this.willCommit = ⊤Bool
}
var PDF_DOM_NODE = {
    name: ⊤String
    setFocus: function() { return ⊤Bool },
    value: ⊤String
}
```

Listing 3: Mock PDF event object used for analysis

4.4 The PDF-JS model

Our analysis depends on a model that emulates the PDF environment during abstract interpretation. Our implementation of the PDF-JS model relies on a set of on API references [12, 13] to properly model parameter and return values. Because *SAFE-PDF* is conservative, it always interprets calls to un-modelled functions as malicious. As a result, given valid JavaScript code, reducing the false positive rate of *SAFE-PDF* usually means extending the model. This is a straight-forward and incremental process, and we show in Section 5 that it yields very good results in practice.

Similar to JavaScript in web browsers, JavaScript code in PDF documents is largely event-driven, i.e., either triggered by system or user events. While we observed that most malicious code is executed when the PDF document is opened, JavaScript code may be placed in event handlers, where it is executed only on certain user actions (e.g., clicking a form field). Because we cannot exclude the possibility that an event handler might contain malicious code, *SAFE-PDF* must consider every event handler as an entry point. Moreover, because event handlers can have side effects that alter the computation of other event handlers, *SAFE-PDF* loops over all handlers and triggers them until a fixpoint is reached, indicating

that it computed a suitable over-approximation for the whole program. Because event handlers receive an `event` object as argument and operate on its properties, our PDF-JS model defines a mock PDF event object that is passed to event handlers at analysis time. Listing 3 shows the mock PDF event object of our PDF-JS model. The right-hand side values $\top_{String}$, $\top_{Bool}$, $\top_{Number}$ represent abstract values. They stand for *any* string, *any* Boolean, and *any* number respectively. The `PDF_DOM_NODE` abstracts nodes in the PDF Document Object Model (DOM) to which events can be attached.

5 EXPERIMENTAL EVALUATION

To assess the effectiveness and robustness of our technique, we implemented the *SAFE-PDF* tool, and investigated the following research questions:

RQ1: How does *SAFE-PDF*'s false positive rate, recall and accuracy compare to state-of-the-art malware detection tools?

RQ2: How resilient is *SAFE-PDF* to parser confusion and adversarial (e.g. mimicry) attacks?

RQ3: How interpretable are *SAFE-PDF*'s results?

5.1 Experiment setup

SAFE-PDF is based on version 1.0 of the *SAFE* abstract interpretation framework for ECMAScript [38]. We complement the original *SAFE* framework with: (1) malware-specific analyses, (2) our semantic models for the PDF JavaScript and XFA environments, and (3) further modeling of the interactions between XML and JavaScript in XFA forms.

Experiments were conducted on a set of 14 306 benign and 9410 malicious PDF documents. Malicious samples were collected from VirusShare [9], a free online repository of malware samples, Contagio [2], and VirusTotal [10]. The benign benchmark set contains non-malicious PDF documents from Contagio, VirusTotal, PDF attachments from a public email dataset [11], as well as samples collected from the web (from Google queries targeting PDF documents, e.g. `filetype:PDF`), test cases for *PDF.js*, the PDF-rendering engine of Mozilla [7], test cases for *PDFium*, the PDF-rendering engine of Chrome [6], and interactive documents from the pdfPictures [8] website. The samples from VirusTotal include the hard-to-extract set of documents from [17], which we refer to in Section 4.1. All PDF documents in the benign dataset were confirmed as benign using VirusTotal's online scanning service. Table 3 lists the different benchmark sets, the number of files they contain, and whether they contain malicious or benign samples. Separately, in Section 5.3 only, we use a PDF dataset crafted to evade malware detection, provided by the authors of the Chameleon framework [24].

To extract JavaScript code from PDF documents, we extended version 2015.1.4 of the *Clean Content* SDK [5], as described in Section 4.1. In the rare cases where a PDF document causes *Clean Content* to fail with an extraction error, we pre-process the document with a modified version of *PDFBox* [1] in an attempt to fix structural issues. Also, we syntactically remove one particular nonsensical but benign code fragment (`jQuery.post(Drupal. settings.basePath + 'jstats.php', {...})`), which calls the non-existent `jQuery.post` method, from the extracted JavaScript code. Instances of this call were introduced into our web-sourced dataset by an obviously broken web-based PDF creator.

Table 3: PDF sample benchmarks used in this study

Benchmark	Source	#Files
Malicious	Contagio	7110
	VirusShare	1206
	VirusTotal	1094
Benign	Contagio	388
	EDRMv2	4434
	VirusTotal	2137
	Web	7347

Table 4: Numbers of true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN) per tool

Tool	TP	TN	FP	FN
Slayer	9338	13877	428	72
Hidost	9379	14086	219	31
SAFE-PDF	9403	13920	386	7

Table 5: Comparison between *Slayer*, *Hidost*, and *SAFE-PDF*

Tool	FP Rate	Recall	Accuracy
Slayer	2.992%	99.23%	97.89%
Hidost	1.531%	99.67%	98.95%
SAFE-PDF	2.698%	99.93%	98.34%

Analysis overhead For our experimental evaluation, we set a 30-second timeout for the abstract interpretation step, after which *SAFE-PDF* rejects an input as malware. We run six instances of *SAFE-PDF* in parallel, sharing eight cores of a Xeon E5-2.60GHz with 32GB RAM. On average, *SAFE-PDF* takes less than 4 seconds to perform full analysis (i.e., extraction and abstract interpretation) of a single PDF document, while less than 1% of samples cause a timeout (see Table 7).

5.2 RQ1: Comparison to state-of-the-art tools

To determine how *SAFE-PDF* compares to the state-of-the-art, we compared the detection rate of *SAFE-PDF* against two other publicly available PDF malware detection tools: *PDF Malware Slayer* [43], and *Hidost* [58].

PDF Malware Slayer first identifies keywords that are characteristic of benign and malicious documents from sets of benign and malicious PDFs. It then trains a *Random Forests* classifier on feature vectors obtained by computing the frequency of characteristic keywords in each document. To measure the false positive rate, recall, and accuracy of *PDF Malware Slayer*, we perform a 10-fold cross-validation experiment with default parameters and report averaged results.

Hidost also uses a *Random Forests* classifier to identify malicious PDFs. It mainly differs from *PDF Malware Slayer* in the way it extracts feature vectors from PDFs. *Hidost* builds feature vectors by extracting structural paths from PDF documents, where structural

paths capture the embedding of PDF components. Because *Hidost* was initially trained and tested on a very large dataset, comprising more than 400,000 documents, it considers only structural paths present in at least 1000 documents by default. To accommodate our smaller dataset, we reduced this threshold to 200. Because *Hidost* also uses a random classifier, we perform a 10-fold cross-validation experiment and report averaged results.

Table 4 summarizes the reports of *PDF Malware Slayer*, *Hidost* and *SAFE-PDF* on our benchmarks. All of the evaluated tools failed to analyze some PDF documents, either through silent failure (e.g., not analyzing the document) or by exiting with an error. Note that *PDF Malware Slayer* and *Hidost* rely on different PDF extraction techniques and are both affected by the limitations described in Table 2 (see [17] for details). Because *SAFE-PDF* is conservative, we treat any extraction failure as an indication that the document is malicious. To perform a fair comparison, we treat errors in other tools as malicious reports too. In total, *PDF Malware Slayer* had 631 errors, *Hidost* had 377 and *SAFE-PDF* had the lowest number, 72 errors. For *SAFE-PDF*, all errors come from heavily broken PDF documents that cause our code extractor to fail.

Going back to our initial research question, the metrics presented in Table 5 show that *SAFE-PDF* achieves a comparable false positive rate, recall and accuracy with state-of-the-art PDF malware detectors.

5.3 RQ2: Resilience to evasion attacks

To evaluate the resilience of *SAFE-PDF* to evasion, we evaluated *SAFE-PDF* on malicious PDF documents that were specifically designed by the authors of [17] to evade detection by performing parser confusion and reverse mimicry attacks [42]. Table 6 shows how many evasive variants were detected by *Slayer*, *Hidost*, and *SAFE-PDF*. Interestingly, while parser confusion attacks were designed to primarily target approaches that rely on JavaScript code extraction, Table 6 shows how tools that rely on structural PDF features are also affected. Indeed, all malicious documents containing R5 handlers evaded *Slayer*, while documents containing R6 security handlers evaded both *Slayer* and *Hidost*. *SAFE-PDF* caught all evasive variants based on parser confusion.

Furthermore, because *SAFE-PDF* statically analyzes program behavior, it is oblivious to reverse mimicry attacks, which are known to be very effective against structure-based approaches [42, 57, 63]. Indeed, as shown in Table 6, *SAFE-PDF* could detect the malicious payload, even in the presence of both reverse mimicry and parser confusion attacks. Manual investigation of the seven false negatives in Table 4 revealed two causes: (1) social engineering that tries to trick the user into executing a malicious attachment of the PDF by displaying instructions in an alert message; (2) inactive (or broken) malware, where JavaScript code contains an obfuscated payload, which is not accessed or invoked anywhere else in the code.

Chameleon evasion study To further assess the robustness of *SAFE-PDF* against evasions, we also evaluated it on 1395 malicious and 81 benign documents from the Chameleon dataset [24]. The Chameleon framework was designed to generate malicious PDF documents with one or more evasions with the goal of measuring their effectiveness against existing academic and commercial tools. The evaluation was conducted in a blind fashion, where we ran *SAFE-PDF* on the untagged Chameleon dataset, and reported our results to the authors of [24] for them to compute the false positive rate and recall.

As can be seen from [24, Figure 3], *SAFE-PDF* achieved the highest recall (100%) among all tools, at the cost of a higher false positive rate (34.57%) on this dataset. Note that while Slayer produced a similar false positive rate (28.77%), its recall is much lower (about 50%). The reason for *SAFE-PDF*'s higher false positive rate, compared to the one reported in Table 5, is that some of the benign documents in the Chameleon dataset exhibit evasive behavior typical of malicious code (e.g. loading and dynamically evaluating code encoded in an image file). For this reason, the benign documents in the Chameleon dataset are not representative of real-world PDF documents and should be considered worst-case false positive rates.

5.4 RQ3: Interpretability of results

As highlighted by Guo et al. [30], lack of transparency reduces the trust in a tool, which is one of the main barriers for wide adoption of deep learning tools by security practitioners. On the one hand, good explanations can help security practitioners understand how malware operates. On the other hand, explanations can also help practitioners troubleshoot classification errors and tune the analysis accordingly.

To evaluate the interpretability of *SAFE-PDF*'s results, we investigated all malicious reports. Benign reports were excluded because a document is classified as benign if and only if it does not exhibit *any* potentially malicious behavior. Table 7 shows the different causes for malicious reports together with their interpretability, count and overall proportion.

The vast majority of malicious reports stem from malicious behavior, and are readily interpretable. *SAFE-PDF* indeed pinpoints the *type* of behavior that was detected (e.g., a vulnerable API call, or allocation of large objects in memory), and either the source location of the call or the allocated object. Unexpected behavior refers to semantically incorrect JavaScript operations (e.g., loading properties from undefined variables, or calling undefined functions) which would cause runtime exceptions and thus represent implementation bugs. In such cases, *SAFE-PDF* pinpoints the source location where the unexpected behavior occurs, providing the security practitioner with valuable insights to understand the report and tune the tool to tolerate certain non-malicious behaviors, if desired. JavaScript parsing errors are also self-explanatory and stem from syntactically broken JavaScript code that cannot be analyzed.

The last two causes require more effort to interpret. First, when abstract interpretation reaches its timeout (30 seconds in our setup), it stops before a fixpoint can be reached. Diagnosing the root cause of a timeout is an involved task that requires a good understanding of abstract interpretation and the framework used for analysis, and is out of scope for the average practitioner. Extraction errors typically stem from severely broken PDFs. We observed that only two of these from our benign dataset can actually be opened with the latest version Acrobat Reader. For this reason, we recommend simply rejecting files that cause extraction errors as malicious. Fortunately, fixpoint and extraction errors are rare (i.e. less than 2%), and do not significantly hinder the usability of our tool.

Table 6: Parser confusion and reverse mimicry attacks

Obfuscation	Slayer	Hidost	SAFE-PDF
None	✓	✓	✓
Flate compression, obj streams	✓	✓	✓
Flate compression, R5 security handler	✗	✓	✓
Flate compression, R5 security handler, obj streams	✗	✓	✓
Flate compression, R6 security handler	✗	✗	✓
Flate compression, R6 security handler, obj streams	✗	✗	✓
Flate compression, R6 security handler, obj streams, comment in trailer	✗	✗	✓
JS encoded as UTF-16BE in hex string	✓	✓	✓
JS encoded as UTF-16BE in hex string, flate compression, obj streams	✓	✓	✓
JS encoded as UTF-16BE in hex string, flate compression, R5 security handler, obj streams, comment in trailer	✗	✓	✓
Reverse mimicry attack + parser confusion	✗	✗	✓

Table 7: Causes of malware reports by *SAFE-PDF*

Interpretability	Report Cause	Count	Percentage
High	Malicious behavior	8655	88.92%
	Unexpected behavior	709	7.28%
	JS parsing error	213	2.19%
Low	Fixpoint not reached	84	0.86%
	Extraction error	72	0.74%

Overall, *SAFE-PDF* produces highly interpretable results in 98.39% of all cases (see Table 7), which can help security practitioners understand the inner workings of PDF malware, and tune *SAFE-PDF* to meet their requirements.

6 RELATED WORK

In this section, we give a brief overview of existing techniques for PDF malware detection (a detailed survey and taxonomy can be found in [45]) and explain how they compare to our analysis approach. We also discuss semantic malware detection and JavaScript static analysis, which are related fields of research.

6.1 Static PDF Malware Detection

The first group of approaches proposed in academic literature that we consider as related work analyzes a PDF document as a whole and does not analyze any embedded JavaScript code. These techniques are categorized as *Metadata Analysis* in [45]. Three properties of PDF documents are being used to derive a *fingerprint*: document structure, metadata fields, and document content. The related approaches [43, 46, 47, 54, 58] rely on a set of known malicious PDF documents as training data to identify documents with a similar fingerprint as malware.

Caradoc [29] is an exception to the above. Endignoux et al. focus on weaknesses in the PDF standard related to document structure. These can be exploited to attack the parser implementation of a PDF viewer, e.g., to achieve a denial-of-service attack.

Comparison to our approach When aiming to identify malicious PDF documents that exploit vulnerabilities in the JavaScript runtime of a PDF viewer, our approach is more powerful, since it does not depend on a training set and is not susceptible to adversarial mimicry attacks.

6.2 Static PDF-JavaScript Malware Detection

Similar to *SAFE-PDF*, the related work in this section performs a static analysis of JavaScript code embedded in PDF documents. However, unlike us, but similar to the approaches in Section 6.1, they identify malicious JavaScript based on its similarity to known malicious samples.

PJScan [37] and Vatamanu's approach [60] both perform lexical analysis of the extracted JavaScript code, and use machine learning techniques to classify the code as malicious or non-malicious. In [34], the authors describe the use *NiCad*, an existing tool for detecting code clones, for the same purpose. Lux0R [20] uses references to methods of the PDF JavaScript API as machine learning features.

Comparison to our approach The approaches above are always less powerful than our static analysis, because they restrict themselves to lexical features of JavaScript code and do not take its semantics into account. All approaches rely on the similarity of (possibly obfuscated) JavaScript code to known malicious code, and might be defeated by novel obfuscation patterns and adversarial mimicry attacks.

6.3 Dynamic PDF Malware Detection

All approaches in this section rely on the execution of PDF-embedded JavaScript code, either in its native or synthetic runtime environment, for analysis of its behavior.

MDScan [59] and *PDF Scrutinizer* [53] execute the extracted JavaScript code in a synthetic environment, and aim to detect the presence of malicious payload (so called *shell code*) in the execution state (as part of strings and variables). *PDF Scrutinizer* applies further heuristics to identify execution patterns typical of malicious code. *PlatPal* [62] is similar, but uses behavioral discrepancies during execution on different platforms to identify a PDF as malicious. Other uses of dynamic analysis proposed in literature do not compare directly to our approach because they are not suitable

as stand-alone analysis tools. For example, *MPScan* [40] and *FC-Scan* [52] propose to integrate with the PDF viewer software, while *ShellOS* [56] and *CWXDetector* [61] detect malicious code at the system level (on-the-fly) during runtime.

Comparison to our approach Dynamic analysis techniques are prone to miss feasible program behavior, because actual execution depends on inputs and the execution environment. In the context of malware analysis, the malware author can actively target a specific environment, thus preventing the detection of malicious behavior in the analysis environment. For example, the de-obfuscation of exploit code might be triggered only in a specific target environment. This undermines the dynamic analysis-based detection techniques described above. Furthermore, unlike our static analysis-based approach, none of the existing dynamic analyses tries to exhaustively explore all possible behavior of the JavaScript code.

6.4 Semantic-Based Malware Detection

Semantic approaches use techniques from program analysis and formal methods to lift malware detection from syntactic features to the level of program semantics. For example, semantic malware detectors use theorem proving [18] or model checking [35] to match a program, based on its semantic properties (e.g., instruction sequences), against a template derived from actual malware. In general, these approaches are more powerful than signature matching, but still prone to evasion by obfuscation. Preda et al. [50] introduce a theoretical framework for semantic malware detection using abstract interpretation. It assumes the availability of *perfect oracles*, which return perfect information related to a program's semantic properties or behavior (e.g., its exact control flow), and shows that whether a detector can overcome a particular obfuscation, depends on the chosen abstract semantics, i.e., the right level of abstraction. More recently, [48] uses statistical analysis of program behavior recorded during dynamic analysis to identify malware. This avoids the difficulty of static reasoning, but introduces the possibility of dynamic analysis missing malicious behavior.

Comparison to our approach Our work can be seen as an instance of malware detection based on Preda's *interesting actions* [50, Section 5]. However, we use a policy to whitelist acceptable behavior and thus do not have to rely on actual malware as templates for malicious behavior. Furthermore, our conservative strategy enables us to have a practical malware detector in the absence of "perfect oracles".

6.5 Web JavaScript Malware Detection

The primary vector for JavaScript-based malware remains web browsers. Prominent work in this area includes Nozzle [51], Zozzle [22], and Rozzle [36]. Nozzle is a dynamic, in-browser approach that uses heap sampling to detect heap-spraying attacks. Similar to other dynamic approaches, Nozzle requires an instrumented environment, a browser in this case, and induces a performance overhead. Zozzle is a mostly static approach that reduces the performance overhead of Nozzle. It uses a Naïve Bayes classifier, trained on syntactic features of the JavaScript code, to classify programs as benign or malicious. Zozzle relies on the browser's JavaScript interpreter to de-obfuscate the code before performing feature extraction and classification. Because both Nozzle and Zozzle rely on an instrumented browser environment, both approaches are susceptible to miss malware that exhibits malicious behavior on other environments only. Rozzle addresses this limitation through the use of symbolic execution to emulate different runtime environments in a single instrumented browser. Instead of symbolic execution, *JSForce* [32] modifies the runtime environment during execution, to execute as much of a program as possible (i.e., *forced execution*), although without maintaining the original program's semantics.

Comparison to our approach Through the use of abstract interpretation, our work detects heap spraying (like Nozzle), performs de-obfuscation (like Zozzle), and simulates different runtime environments (like Rozzle and JSForce) *entirely* statically. Furthermore, *SAFE-PDF* can statically reason about the runtime behavior of the code, making it more powerful than syntax-based approaches.

6.6 JavaScript Static Analysis

The dynamic nature of JavaScript and its lack of static guarantees make it a difficult target for static analysis. This shortcoming is magnified by the inherent complexity of the most common use of JavaScript, as client-side web application code running inside an equally complex browser environment. At this point in time, state-of-the-art tools based on dataflow analysis [28] or precise abstract interpretation [33, 38] can successfully analyze libraries and small applications, but do not scale to real-world JavaScript code in general.

We are nonetheless successful in using the very same techniques to analyze JavaScript in the context of malware detection. This is due to two reasons: (1) JavaScript code embedded in PDF documents is not as complex as code written for the web; (2) malware detection warrants a conservative strategy where unknown behaviors are interpreted as malicious.

7 CONCLUSION

We presented a novel approach for detecting malicious JavaScript embedded in PDF documents that uses abstract interpretation—a static program analysis technique—at its core. Using the results of abstract interpretation in a conservative manner, our malware detection is designed for and achieves very high recall. Furthermore, with an average runtime of less than 4 seconds per document, we showed how traditionally "heavy-weight" abstract interpretation tools can be used in practice, given the right abstraction (e.g. the PDF-JS model). By addressing all known limitations of existing PDF JavaScript code extractors, we showed that PDF malware detectors that analyze the embedded JavaScript code can be used in practice. We also showed how *SAFE-PDF* resists obfuscation, parser confusion, and mimicry evasion attacks that subvert existing malware detector tools. Finally, through comprehensive experimental evaluation, we have shown that our approach achieves almost perfect recall, and a comparable false positive rate to state-of-the-art tools while being able to produce highly interpretable explanations.

ACKNOWLEDGMENTS

The authors would like to thank Phil Boutros and Joe Keslin from the Oracle Outside In team for their support.

REFERENCES

[1] [n. d.]. Apache PDFBox® - A Java PDF Library. https://pdfbox.apache.org/. Accessed 2019-07-08.

[2] [n. d.]. Contagio - Malware Dump. http://contagiodump.blogspot.com.au/2013/03/16800-clean-and-11960-malicious-files.html. Accessed: 2019-07-08.

[3] [n. d.]. Cuckoo Sandbox. https://cuckoosandbox.org/. Accessed: 2019-07-08.

[4] [n. d.]. LiveCycle® Designer ES Scripting Reference. https://help.adobe.com/en_US/livecycle/11.0/DesignerScriptingRef/index.html. Accessed: 2019-07-08.

[5] [n. d.]. Outside In Clean Content. https://www.oracle.com/technetwork/middleware/content-management/oit-all-085236.html. Accessed: 2019-07-08.

[6] [n. d.]. PDFium. https://pdfium.googlesource.com/pdfium/. Accessed: 2019-07-08.

[7] [n. d.]. PDF.js - A general-purpose, web standards-based platform for parsing and rendering PDFs. https://mozilla.github.io/pdf.js/. Accessed: 2019-07-08.

[8] [n. d.]. pdfPictures - Interactive Electronic Rich Media Documents. http://www.pdfpictures.com/. Accessed: 2019-07-08.

[9] [n. d.]. VirusShare.com - Because Sharing is Caring. https://virusshare.com/. Accessed: 2019-07-08.

[10] [n. d.]. VirusTotal. https://www.virustotal.com/. Accessed: 2019-07-08.

[11] [n. d.]. William W. Cohen: Enron Email Dataset. https://www.cs.cmu.edu/~enron/. Accessed: 2019-07-08.

[12] 2006. JavaScript for Acrobat API Reference. http://www.adobe.com/devnet/acrobat/javascript.html. Accessed: 2019-07-08.

[13] 2007. JavaScript for Acrobat 3D Annotations API Reference. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/js_3d_api_reference.pdf. Accessed: 2019-07-08.

[14] 2008. *Document management - Portable document format.* Standard. International Organization for Standardization, Geneva, CH.

[15] 2012. The Rise of PDF Malware. https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/the_rise_of_pdf_malware.pdf. Accessed: 2019-07-08.

[16] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. 2013. Evasion Attacks against Machine Learning at Test Time. *Machine Learning and Knowledge Discovery in Databases* 8190 (2013), 387–402. https://doi.org/10.1007/978-3-642-40994-3_25

[17] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. 2016. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *Proceedings of the Network and Distributed System Security Symposium.* (hard-to-extract file hashes: https://goo.gl/qtbuOC).

[18] Mihai Christodorescu, Somesh Jha, Sanjit A Seshia, Dawn Song, and Randal E Bryant. 2005. Semantics-Aware Malware Detection. (2005). http://repository.cmu.edu/cgi/viewcontent.cgi?article=1031{&}context=ece

[19] Fady Copty, Matan Danos, Orit Edelstein, Cindy Eisner, Dov Murik, and Benjamin Zeltser. 2018. Accurate Malware Detection by Extreme Abstraction. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18).* 101–111.

[20] Igino Corona, Davide Maiorca, Davide Ariu, and Giorgio Giacinto. 2014. Lux0R: Detection of Malicious PDF-embedded JavaScript code through Discriminant Analysis of API References. *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2014), 47–57. https://doi.org/10.1145/2666652.2666657

[21] Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. , 238–252 pages. https://doi.org/10.1145/512950.512973

[22] Charlie Curtsinger, Benjamin Livshits, Benjamin G Zorn, and Christian Seifert. 2011. ZOZZLE: Fast and Precise In-Browser JavaScript Malware Detection.. In *USENIX Security Symposium.* 33–48.

[23] Manuel Egele, Theodoor Scholte, Engin Kirda, and Christopher Kruegel. 2008. A Survey on Automated Dynamic Malware-analysis Techniques and Tools. *ACM Comput. Surv.* 44, 2, Article 6 (2008), 42 pages.

[24] Saeed Ehteshamifar, Antonio Barresi, Thomas R. Gross, and Michael Pradel. 2019. Easy to Fool? Testing the Anti-evasion Capabilities of PDF Malware Scanners. *arXiv e-prints*, Article arXiv:1901.05674 (2019). arXiv:1901.05674 http://arxiv.org/abs/1901.05674

[25] Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. 2018. Analysis of classifiers' robustness to adversarial perturbations. *Machine Learning* 107, 3 (Mar 2018), 481–508.

[26] Prahlad Fogla and Wenke Lee. 2006. Evading Network Anomaly Detection Systems: Formal Reasoning and Practical Techniques. In *Proceedings of the 13th ACM Conference on Computer and Communications Security (CCS '06).* 59–68.

[27] Amir Globerson and Sam Roweis. 2006. Nightmare at test time: robust learning by feature deletion. In *Proceedings of the 23rd international conference on Machine learning.* ACM, 353–360.

[28] S. Guarnieri, M. Pistoia, O. Tripp, J. Dolby, S. Teilhet, and R. Berg. 2011. Saving the World Wide Web from Vulnerable JavaScript. In *ISSTA.* 177–187.

[29] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. 2016. Caradoc: A Pragmatic Approach to PDF Parsing and Validation. *2016 IEEE Security and Privacy Workshops (SPW)* (2016), 126–139. https://doi.org/doi.ieeecomputersociety.org/10.1109/SPW.2016.39

[30] Wenbo Guo, Dongliang Mu, Jun Xu, Purui Su, Gang Wang, and Xinyu Xing. 2018. LEMNA: Explaining Deep Learning Based Security Applications. In *Conference on Computer and Communications Security (CCS '18).* ACM, New York, NY, USA, 364–379.

[31] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. 2001. *Introduction to automata theory, languages, and computation* (2nd ed.). Addison Wesely. https://doi.org/10.1145/568438.568455

[32] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2018. JSForce: A forced execution engine for malicious javascript detection. In *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST.* https://doi.org/10.1007/978-3-319-78813-5_37 arXiv:1701.07860

[33] S. H. Jensen, A. Møller, and P. Thiemann. 2009. Type Analysis for JavaScript. In *SAS.* 238–255.

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

[35] Johannes Kinder, Stefan Katzenbeisser, Christian Schallhart, and Helmut Veith. 2005. Detecting Malicious Code by Model Checking. *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA'05)* 3548 (2005), 174–187. https://doi.org/10.1007/11506881_11

[36] Clemens Kolbitsch, Benjamin Livshits, Benjamin Zorn, and Christian Seifert. 2012. ROZZLE: De-cloaking internet malware. In *Security and Privacy (SP), 2012 IEEE Symposium on.* IEEE, 443–457.

[37] Pavel Laskov and Nedim Šrndić. 2011. Static Detection of Malicious JavaScript-bearing PDF Documents. *Proceedings of the 27th Annual Computer Security Applications Conference* (2011), 373–382. https://doi.org/10.1145/2076732.2076785

[38] Hongki Lee, Sooncheol Won, Joonho Jin, Junhee Cho, and Sukyoung Ryu. 2012. SAFE: Formal specification and implementation of a scalable analysis framework for ECMAScript. In *International Workshop on Foundations of Object-Oriented Languages (FOOL).*

[39] Daiping Liu, Haining Wang, and Angelos Stavrou. 2014. Detecting malicious javascript in pdf through document instrumentation. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on.* IEEE, 100–111.

[40] Xun Lu, Jianwei Zhuge, Ruoyu Wang, Yinzhi Cao, and Yan Chen. 2013. De-obfuscation and detection of malicious PDF files with high accuracy. In *Proceedings of the Annual Hawaii International Conference on System Sciences.* 4890–4899. https://doi.org/10.1109/HICSS.2013.166

[41] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. 2018. Towards Robust Detection of Adversarial Infection Vectors: Lessons Learned in PDF Malware. *arXiv e-prints*, Article arXiv:1811.00830 (Nov. 2018). arXiv:cs.CR/1811.00830

[42] Davide Maiorca and Giorgio Giacinto. 2013. Looking at the Bag is not Enough to Find the Bomb : An Evasion of Structural Methods for Malicious PDF Files Detection. *Proceedings of the ASIA CCS'13* (2013), 119–129. https://doi.org/10.1145/2484313.2484327

[43] Davide Maiorca, Giorgio Giacinto, and Igino Corona. 2012. A pattern recognition system for malicious PDF files detection. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 7376 LNAI (2012), 510–524. https://doi.org/10.1007/978-3-642-31537-4_40

[44] Flemming Nielson, Hanne R Nielson, and Chris Hankin. 1999. *Principles of Program Analysis.* Springer-Verlag Berlin Heidelberg New York.

[45] Nir Nissim, Aviad Cohen, Chanan Glezer, and Yuval Elovici. 2015. Detection of malicious PDF files and directions for enhancements: A state-of-the art survey. *Computers and Security* 48 (2015), 246–266. https://doi.org/10.1016/j.cose.2014.10.014

[46] Nir Nissim, Aviad Cohen, Robert Moskovitch, Asaf Shabtai, Matan Edri, Oren BarAd, and Yuval Elovici. 2016. Keeping pace with the creation of new malicious PDF files using an active-learning based detection framework. *Security Informatics* 5, 1 (2016), 1. https://doi.org/10.1186/s13388-016-0026-3

[47] Nir Nissim, Aviad Cohen, Robert Moskovitch, Assaf Shabtai, Mattan Edry, Oren Bar-Ad, and Yuval Elovici. 2014. ALPD: Active learning framework for enhancing the detection of malicious PDF files. In *Proceedings - 2014 IEEE Joint Intelligence and Security Informatics Conference, JISIC 2014.* 91–98. https://doi.org/10.1109/JISIC.2014.23

[48] Sirinda Palahan, Domagoj Babić, Swarat Chaudhuri, and Daniel Kifer. 2013. Extraction of statistically significant malware behaviors. *Proceedings of the 29th Annual Computer Security Applications Conference on - ACSAC '13* (2013), 69–78. https://doi.org/10.1145/2523649.2523659

[49] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. 2016. Distillation as a defense to adversarial perturbations against deep neural networks. In *2016 IEEE Symposium on Security and Privacy (SP).* IEEE, 582–597.

[50] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. 2008. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems* 30, 5 (2008), 1–54. https://doi.org/10.1145/1387673.1387674 arXiv:179

[51] Paruj Ratanaworabhan, V Benjamin Livshits, and Benjamin G Zorn. 2009. NOZZLE: A Defense Against Heap-spraying Code Injection Attacks.. In *USENIX Security Symposium*. 169–186.

[52] Christiaan Leonard Schade. 2013. FCScan: A new lightweight and effective approach for detecting malicious content in electronic documents. (2013).

[53] Florian Schmitt, Jan Gassen, and Elmar Gerhards-Padilla. 2012. PDF Scrutinizer: Detecting JavaScript-based attacks in PDF documents. *2012 10th Annual International Conference on Privacy, Security and Trust, PST 2012* (2012), 104–111. https://doi.org/10.1109/PST.2012.6297926

[54] Charles Smutz and Angelos Stavrou. 2012. Malicious PDF detection using metadata and structural features. *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12* (2012), 239. https://doi.org/10.1145/2420950.2420987

[55] Charles Smutz and Angelos Stavrou. 2016. When a Tree Falls: Using Diversity in Ensemble Classifiers to Identify Evasion in Malware Detectors. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*.

[56] Kevin Z. Snow, Srinivas Krishnan, Fabian Monrose, and Niels Provos. 2011. SHELLOS: enabling fast detection and forensic analysis of code injection attacks. In *Proceedings of the 20th USENIX conference on Security (SEC'11)*. 9. http://dl.acm.org/citation.cfm?id=2028067.2028076

[57] Nedim Šrndić and Pavel Laskov. 2014. Practical evasion of a learning-based classifier: A case study. In *Proceedings - IEEE Symposium on Security and Privacy*. 197–211. https://doi.org/10.1109/SP.2014.20

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

[59] Zacharias Tzermias, Giorgos Sykiotakis, Michalis Polychronakis, and Evangelos P. Markatos. 2011. Combining static and dynamic analysis for the detection of malicious documents. *Proceedings of the Fourth European Workshop on System Security - EUROSEC '11* (2011), 1–6. https://doi.org/10.1145/1972551.1972555

[60] Cristina Vatamanu, Dragoş Gavriluţ, and Răzvan Benchea. 2012. A practical approach on clustering malicious PDF documents. *Journal in Computer Virology* 8, 4 (nov 2012), 151–163. https://doi.org/10.1007/s11416-012-0166-z

[61] Carsten Willems, Felix C. Freiling, and Thorsten Holz. 2012. Using memory management to detect and extract illegitimate code for malware analysis. *Proceedings of the 28th Annual Computer Security Applications Conference on - ACSAC '12* (2012), 179. https://doi.org/10.1145/2420950.2420979

[62] Meng Xu and Taesoo Kim. 2017. PlatPal: Detecting Malicious Documents with Platform Diversity. *USENIX Security '17* (2017). https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-xu-meng.pdf

[63] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically evading classifiers. In *Proceedings of the Network and Distributed System Security Symposium*.