Static Detection of Malicious JavaScript-Bearing PDF Documents

Pavel Laskov
University of Tübingen
Sand 1, 72076
Tübingen, Germany
pavel.laskov@uni-tuebingen.de

Nedim Šrndić
University of Tübingen
Sand 1, 72076
Tübingen, Germany
nedim.srndic@uni-tuebingen.de

ABSTRACT

Despite the recent security improvements in Adobe's PDF viewer, its underlying code base remains vulnerable to novel exploits. A steady flow of rapidly evolving PDF malware observed in the wild substantiates the need for novel protection instruments beyond the classical signature-based scanners. In this contribution we present a technique for detection of JavaScript-bearing malicious PDF documents based on static analysis of extracted JavaScript code. Compared to previous work, mostly based on dynamic analysis, our method incurs an order of magnitude lower run-time overhead and does not require special instrumentation. Due to its efficiency we were able to evaluate it on an extremely large real-life dataset obtained from the VirusTotal malware upload portal. Our method has proved to be effective against both known and unknown malware and suitable for large-scale batch processing.

Categories and Subject Descriptors

D.4.6 [**Software**]: Operating Systems—*Security and Protection*; I.2.6 [**Computing Methodologies**]: Artificial Intelligence—*Learning*

Keywords

Malware detection, malicious JavaScript, PDF documents, machine learning

1. INTRODUCTION

Since the discovery of the first critical vulnerability in Adobe Reader in 2008[1] the Portable Document Format (PDF) has become one of the main attack vectors used by miscreants. PDF-based attacks were the most frequently used remote exploitation technique in 2009 with a proud share of 49%. Two specific PDF-based vulnerabilities were ranked second and fifth among all vulnerabilities discovered in 2009 [1]. Overall, more than 50 vulnerabilities were discovered in Adobe Reader in 2008–2010, which has led to numerous security-related updates. The severity of security problems has somewhat abated with the introduction of the sandboxing technology—which has not been compromised to date—in Adobe Reader X (version 10). The underlying code base, however, still remains vulnerable, and some critical patches were issued earlier this year [2, 3].

The vulnerabilities of Adobe Reader can be classified into three categories. The earliest—and the largest—class of vulnerabilities arises from bugs in the implementation of the Adobe JavaScript API. This API significantly extends the JavaScript functionality in the specific context of PDF documents. The second class of vulnerabilities is rooted in non-JavaScript features of Adobe Reader but typically requires JavaScript for exploitation, e.g. using heap spraying. Examples of such vulnerabilities are the JBIG2 filter (e.g. CVE-2009-0658) and the heap overflow (e.g. CVE-2009-1862) exploits. Finally, the smallest class of vulnerabilities, e.g. the flawed embedded TrueType font handling (CVE-2010-0195), does not involve JavaScript functionality.

Unlike other modern exploitation techniques such as drive-by-downloads, SQL injection or cross-site scripting, the PDF-based attacks have not received significant attention in the research community so far. Previous work in this field has mostly focused on dynamic analysis techniques. For example, well-known sandboxes JSand [7] and CWSandbox [26] have been adapted to the analysis of malicious PDF documents. Due to their heavy instrumentation and security risks associated with dynamic analysis, the practical applicability of such approaches is limited to malware research systems. For the end-user systems, some early work on the detection of potential exploits in PDF documents [13, 21] has gone largely unnoticed, and in practice the detection of malicious PDF documents still hinges upon signatures provided by security vendors.

In this paper, we explore *static* analysis techniques for detection of JavaScript-based PDF exploits. Our aim is to develop efficient detection methods suitable for deployment on end-user systems as well as in the networking infrastructure, e.g. email gateways and HTTP proxies. We present the tool PJScan[2] that is capable of reliably detecting PDF attacks with operational false positive rates in the promille range. The low computational overhead of PJScan makes it very attractive for large-scale analysis of PDF data.

Conceptually, PJScan is closely related to static analysis techniques for detection of browser-based JavaScript attacks. Similarly to the recent work of Rieck et al. [19], our methodology is based on lexical analysis of JavaScript code and uses machine learning to automatically construct models from available data for subsequent classification of new data. The crucial difference from browser-

[1]collab.CollectEmailInfo (CVE-2007-5659): known, as usual, well ahead of an exploit.

[2]The source code of PJScan and its underlying library libPDFJS can be found at `http://sf.net/p/{pjscan|libpdfjs}`.

based JavaScript attacks is that reliable ground truth information is hardly available for PDF documents. It is especially difficult to identify benign JavaScript-bearing PDF documents. First, as our study will show, such examples are indeed much more rare than malicious ones. Second, while it is relatively easy to verify that web content at a certain URL is benign by using Google Safe Browsing[3], it is much more difficult to extract and analyze JavaScript code in PDF documents. These implications necessitate a conceptual redesign of the detection methods. In PJScan, we have to resort to *anomaly detection* to learn only from malicious examples.

Reliable extraction of JavaScript code from PDF documents is itself a major challenge. Not only is PDF very complex, it is also rich with features that can be used for hiding the presence of JavaScript code. It supports compression of arbitrary objects as well as various encodings for the JavaScript content. Such features are routinely used by attackers to avoid detection by signature-based methods. In our experience, none of the previous tools for static analysis of PDF documents, e.g. PDFID[4], JSUNPACK[5], PDF Dissector[6], were able to provide full coverage of the possible locations of JavaScript code in PDF documents. In the preprocessing component of the PJScan, we have developed an interface to a popular PDF rendering library Poppler[7]. Using this interface, our system is able to handle all potential locations of JavaScript known to us from the PDF Reference.

We have evaluated the effectiveness of PJScan on a large real-world dataset comprising 3 months of data uploaded by users to the malware analysis portal VirusTotal[8]. This is the first study of malicious PDF documents carried out at such scale. Our results confirm that there still exist malicious PDF documents that are not recognized by any antivirus system, although the share of novel malicious PDF documents is no longer significant (we have found 52 such documents among more than 40,000 documents classified by VirusTotal as benign). In our experiments, PJScan has attained average detection rates of 85% for known and 71% for previously unknown PDF attacks with the average operational false positive rate of about 0.37%. Due to the difference in the nature of benign data a direct comparison of PJScan with methods for detection of browser-based JavaScript attacks is not possible. Wepawet[9] was the only previous detection method suitable for PDF-based JavaScript attacks. Much to our surprise, while being perfect in terms of false positives and very good in detection of novel PDF attacks (90%), Wepawet has shown poor performance on *known* PDF attacks, for which it only reached the detection accuracy of 63.6%. As a dynamic analysis tool, Wepawet has been conceived for offline analysis and incurs a significant overhead.

1.1 Contributions

In summary, this paper provides the following contributions:

1. **Robust extraction of JavaScript from PDF documents.** We provide a detailed account of the mechanisms for embedding of JavaScript content in PDF documents and present a methodology for reliable extraction of JavaScript code using the open source PDF parser Poppler.

2. **Fully static detection of malicious JavaScript.** We describe a method for discrimination between malicious and benign JavaScript instances based on lexical analysis and anomaly detection. Unlike the previous work, the proposed method does not require manual labeling of data. This is especially important for PDF documents for which it is difficult to verify that a certain document is benign.

3. **High performance.** The key advantage of static analysis is that it allows several orders of magnitude higher processing speed. Our system PJScan has attained the average processing time of less than 50ms per file.

4. **Comprehensive evaluation.** We present the results of a first large-scale evaluation of malicious PDF detection on a real-world dataset comprising more than 65,000 PDF documents. PJScan has detected 85% of known malicious PDF documents compared to *all 42* antivirus scanners deployed by VirusTotal and 71% of previously unknown malicious PDF documents (not detected by any of the VirusTotal's scanners). The promille-range false positive rate of PJScan makes it suitable for practical deployment.

1.2 Paper Organization

The rest of this article is organized as follows. We begin with a brief summary of the main features of PDF and its mechanisms for embedding of JavaScript contents (Section 2). The architecture of PJScan and the methodology used in its specific components is presented in Section 3. In Section 4, we present the data corpus and analyze its statistical features at different representational levels. Our experimental evaluation is presented in Section 5, followed by the discussion of related work in Section 6. Limitations of our methods and potential improvements are discussed in Section 7.

2. PDF AND JAVASCRIPT

Before presenting the technical details of our methods, we briefly summarize the main features of the Portable Document Format and present its syntactic forms used for embedding of JavaScript. A significant portion of the following section contains direct citations from the PDF Reference [15].

2.1 PDF Essentials

A PDF file consists of the following four elements[10]:

- A *header* consisting of the characters `%PDF-` and the version number of the PDF standard used in the file (e.g. 1.1),
- A *body* containing PDF objects with the actual content of the document,
- A *cross-reference table* listing indirectly referenced objects and their location in the file,
- A *trailer*, containing the location of the cross-reference table and some objects in the file body.

The parsing of a PDF file begins with checking the version number and looking at the file trailer for information about the location of the cross-reference table and some special objects in the file body.

[3]Google Safe Browsing API: `http://code.google.com/apis/safebrowsing/`

[4]`http://blog.didierstevens.com/programs/pdf-tools/#pdfid`

[5]`https://code.google.com/p/jsunpack-n/`

[6]`http://www.zynamics.com/dissector.html`

[7]`http://poppler.freedesktop.org/`. Version 0.14.3 was used in our implementation.

[8]VirusTotal, Free Online Malware Scanner, `http://www.virustotal.com/index.html`

[9]Wepawet, `http://wepawet.iseclab.org/index.php`

[10]Many parsers do not strictly follow the PDF Standard. Even the Adobe Reader is notorious for such lack of compliance, e.g. it ignores arbitrary symbols before the header [12] and can dispense with the trailer and cross-references [27].

The PDF standard defines eight basic types of objects:

1. *Boolean* objects take values **true** and **false**.
2. *Integer* and *real* numbers.
3. *Strings* may be stored in two ways:
   - as a sequence of literal characters enclosed in parentheses '(' and ')'.
   - as a sequence of hexadecimal numbers enclosed in angle brackets '<' and '>'.
4. *Names* are sequences of 8-bit characters used as identifiers.
5. *Arrays* are sequences of PDF objects, potentially of different type; arrays can be nested.
6. *Dictionaries* are collections of key-value pairs with keys being names and values being of any PDF object type. Dictionaries are used to describe complex objects such as pages or actions.
7. *Streams* are dictionary objects followed by a sequence of bytes between the words **stream** and **endstream**. Streams can be used to represent large objects, such as images, in a compact way. The content of the byte sequence may be stored in an encoded or compressed form. A special type of streams are *object streams* containing arbitrary PDF objects.
8. The *null* object is denoted by the keyword **null**.

The body of a PDF document is built as a hierarchy of these eight basic types of objects linked together in a semantically meaningful way to describe pages, multimedia, outlines, annotations, etc. A central role in the hierarchy belongs to the *Catalog* dictionary pointed to by the `/Root` entry of the cross-reference table. It serves as the root of a tree-like structure describing the document content.

Objects can be assigned a unique identifier consisting of an object number and a generation number (a sort of a version number). Objects that have a unique identifier can be referenced from other objects using an *indirect reference* written as a sequence of the object number, the generation number and the capital letter 'R'. For example, `23 0 R` refers to an object with the object number 23 and the generation number 0. PDF allows encryption of the contents of strings and streams.

2.2 JavaScript in PDF

PDF provides several mechanisms for inclusion of JavaScript code. These mechanisms are important for the realization of interactive features, such as forms, dynamic content or 3D rendering. Some PDF usage scenarios relying on these features cannot be realized without JavaScript.

The main indicator for JavaScript code is the presence of the keyword `/JS` in some dictionary. The JavaScript source is supplied directly as one of the two possible string types (literal or hexadecimal) or stored in another object pointed to by an indirect reference. In the latter case, it is usually stored in a compressed or encrypted form in a stream attached to that object. Examples of typical syntax for embedding of JavaScript code are shown in Fig. 1.

A simple search for `/JS` patterns in PDF files – as it was realized in some tools for the analysis of PDF documents, e.g. PDFID – does not suffice for identification of JavaScript locations. It can be easily evaded by placing objects containing dictionaries with the keyword `/JS` into object streams. Due to stream compression the keyword `/JS` is not visible in plain text. The simple search may also

```
1 0 obj <<                1 0 obj <<
  /Type /Catalog            /Type /Catalog
  /Pages 2 0 R              /Pages 2 0 R
  /OpenAction <<            /OpenAction <<
    /S /Rendition             /S /JavaScript
    /JS 23 0 R                /JS (alert('Hello World!');)
  >>                        >>
>>                        >>
endobj                    endobj
```

Figure 1: Exemplary syntactic constructs for embedding of JavaScript in PDF documents. Left: code is placed in another object pointed to by an indirect reference (not shown). Right: code is supplied as a literal string.

yield multiple references to identical code if different revisions of the same content are present.

In order to reliably extract JavaScript code, the documents must be processed at the semantic level, i.e. considering potential uses of JavaScript in the context of other objects in a document. In general, the use of JavaScript code in PDF documents is bound to the so-called *action dictionaries*. Such dictionaries may be tagged by a keyword/value pair `/Type/Action`, but unfortunately such explicit qualification is optional and cannot be relied upon. A mandatory feature of all action dictionaries is the keyword `/S` which may take on 18 different name values. Two of such values, `/JavaScript` and `/Rendition`, are important for the search for JavaScript code. The former must, and the latter may have a keyword `/JS` [15], as shown in Fig. 1. The content associated with the keyword `/JS` must use the PDFDocEncoding (as defined in [15]) or the UTF-16BE (big-endian) Unicode encoding. In the rest of this article we denote JavaScript source code located in or referred to by one *JavaScript* or *Rendition* action dictionary as a *JavaScript entity*.

JavaScript or *Rendition* action dictionaries can be found at the following locations of the PDF object hierarchy:

- The Catalog dictionary's `/AA` entry may define an additional action specified by a JavaScript action dictionary.
- The Catalog dictionary's `/OpenAction` entry may define an action to be run when the document is opened.
- The document's name tree may contain an entry 'JavaScript' that maps name strings to document-level JavaScript action dictionaries executed when the document is opened.
- The document's *Outline* hierarchy, referenced by means of the 'Outlines' entry of the Catalog dictionary, may contain references to JavaScript action dictionaries.
- Pages, file attachments and forms may also contain references to JavaScript action dictionaries.

Besides being directly embedded in a PDF file, JavaScript code may also reside in a different file on a local machine or even be retrieved from a remote location using the directives `/URI` or `/GoTo`. JavaScript also supports dynamic code execution using the `eval()` function or its equivalent, `setTimeOut()`. Such mechanisms are difficult for static analysis; however, they are launched from an existing *entry point* code inside a document.

3. SYSTEM ARCHITECTURE

The architecture of our PDF scanner PJScan is shown in Fig. 2. Conceptually, our system consists of the feature extraction and the learning components. The feature extraction component searches

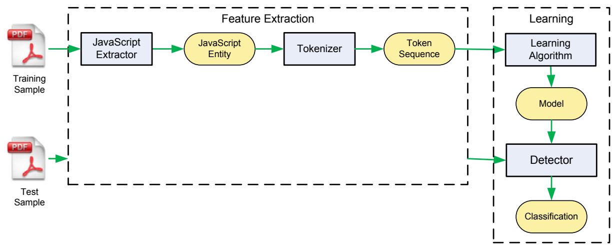


Figure 2: Architecture of PJScan

for JavaScript code embedded in a document and performs lexical analysis on it. The resulting token sequence is used as an input to the learning component. The learning component is first trained on examples of *malicious* documents. It produces a model for the JavaScript content in a malicious PDF document. Classification of new documents is performed using this model. A detector measures the deviation of a new document from a learned model and compares it against a predefined threshold (the threshold can also be automatically determined at the training stage). Documents that are close to a learned model are classified as malicious and otherwise as benign. The functionality of specific components depicted in Fig. 2 is described below.

3.1 Extraction of JavaScript Content

The main challenge of the extraction of JavaScript content lies in the decoding of object streams and the handling of the encoding used for JavaScript content. Furthermore, a parser must be robust against potential incompatibilities with the PDF Standard. For this reason, contrary to the approach taken in [24], we have decided against the parsing of PDF files "by hand" and tailored a popular open source PDF parser Poppler to the needs of our analysis.

Our JavaScript extractor begins with opening the PDF file and initializing Poppler and its internal data structures. Next, the Catalog dictionary is retrieved which serves as the starting point in the search for action dictionaries. All candidate locations listed in Section 2.2 are checked, and the found action dictionaries are queried for their type. If the type is *Rendition* or *JavaScript* and a dictionary contains the `/JS` key, the value of this key (or the referenced object in case of an indirect reference) is retrieved. The JavaScript entity is then decompressed and decoded if necessary.

The peculiarity of our approach is that we fully process only those objects in which *JavaScript* and *Rendition* action dictionaries can potentially occur. This strongly reduces the computational effort for extraction of JavaScript content and is crucial for batch processing of large datasets. Files that do not contain any JavaScript are not processed beyond the extraction stage.

3.2 Lexical Analysis

Two factors motivate the use of lexical analysis for the detection of malicious JavaScript code. First, we believe that at the text level, accurate discrimination between malicious and benign programs is not possible. Second, malicious JavaScript code is usually – sometimes insanely – obfuscated. We have also observed obfuscation in benign JavaScript entities extracted from PDF documents. Hence we have decided to use an intermediate representation – the set of lexical tokens – to capture the salient properties of code in subsequent analysis.

The lexical analysis can be efficiently carried out by the state-of-the-art open source JavaScript interpreter SpiderMonkey[11] developed by the Mozilla Foundation. To use it as a token extractor, we have patched SpiderMonkey to stop short of byte-code generation. Our extractor queries SpiderMonkey for tokens until an end-of-file or an error is encountered. Tokens representing various syntactic elements of the JavaScript language, e.g. identifiers, operators, etc., are represented as symbolic names with integer values ranging from -1 (`TOK_ERR`) to 85.

Some semantics of the code is lost during lexical analysis. For example, all identifiers get assigned the same token `TOK_NAME` (regardless of their names), calls to different functions with identical signatures are translated into the same token sequences, and so on. As a result, JavaScript entities that are distinct at the source code level may be non-distinct at the token sequence level.

The following example illustrates the tokenization process. The malicious JavaScript entity

```
bvb('var lBvXSUfYYL7RK = ev' + 'al;'); // a real example
lBvXSUfYYL7RK('var uzWPsX8 = this.info' +
   z("%2e%46%61%6b") + 'erss;');
```

is transformed into the following sequence of tokens, shown in their order from top to bottom:

Value	Symbolic name	Description
29	`TOK_NAME`	identifier
27	`TOK_LP`	left parenthesis
31	`TOK_STRING`	string constant
15	`TOK_PLUS`	plus
31	`TOK_STRING`	string constant
28	`TOK_RP`	right parenthesis
2	`TOK_SEMI`	semicolon
29	`TOK_NAME`	identifier
27	`TOK_LP`	left parenthesis
31	`TOK_STRING`	string constant
15	`TOK_PLUS`	plus
29	`TOK_NAME`	identifier
27	`TOK_LP`	left parenthesis
31	`TOK_STRING`	string constant
28	`TOK_RP`	right parenthesis
15	`TOK_PLUS`	plus
31	`TOK_STRING`	string constant
28	`TOK_RP`	right parenthesis
2	`TOK_SEMI`	semicolon
0	`TOK_EOF`	end of file

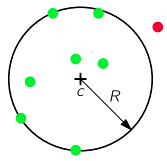
Besides the tokens recognized by SpiderMonkey, we have defined extra tokens that are indicative of malicious JavaScript entities. The newly-introduced tokens are listed in the following table. The impact of these tokens on the classification performance of PJScan is evaluated in Section 5.4.

Value	Symbolic name	Description
101	`TOK_STR_10`	a string literal of length < 10
102	`TOK_STR_100`	a string literal of length < 100
103	`TOK_STR_1000`	a string literal of length < 1,000
104	`TOK_STR_10000`	a string literal of length < 10,000
105	`TOK_STR_UNBOUND`	a string literal of length > 10,000
120	`TOK_UNESCAPE`	a call to `unescape()`
121	`TOK_SETTIMEOUT`	a call to `setTimeOut()`[12]
122	`TOK_FROMCHARCODE`	a call to `fromCharCode()`[13]
123	`TOK_EVAL`	a call to `eval()`

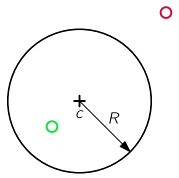
[11]`http://www.mozilla.org/js/spidermonkey/`

[12]In PDF, the function `setTimeOut()` of the `app` object can be used as a replacement for `eval()` to execute arbitrary JavaScript code after the specified timeout.

[13]`fromCharCode()` is a static method of the *String* object that converts Unicode values to characters. In malicious documents, it is used to decode encoded strings for execution using `eval()`.



(a) Learning stage: the center c and the radius R of the sphere are determined.



(b) Classification stage: new data are accepted (green) or rejected (red).

Figure 3: OCSVM operation

3.3 Learning and Classification

In the last step in our processing chain, the learning component of PJScan determines whether a PDF file is benign or malicious. Prior to deployment, it must be trained on a representative set of malicious PDF files. The training results in a model of malicious JavaScript entities in a PDF document. At the deployment stage, classification of new PDF documents is carried out with the help of the learned model. After the feature extraction steps described in Sections 3.1 and 3.2 are completed, the set of tokens from a new document is tested for proximity to the model.

In our system, we use the One-Class Support Vector Machine (OCSVM) [23] as the learning method[14]. Its main advantage is that it only needs examples of one class to build a model. This is necessary since examples of benign PDF documents with JavaScript content are quite rare, and it takes a lot of manual effort to verify that they are benign. On the other hand, examples of malicious PDF documents abound on malware collection systems, and their maliciousness can be ascertained with high confidence if they are detected by antivirus systems.

The learning stage of OCSVM (cf. Fig. 3(a)) amounts to finding the center c and the radius R of a high-dimensional hypersphere such that the total percentage of all data points lying outside of the hypersphere is at most ν. A hypersphere may be extended to arbitrary surfaces by a non-linear transformation to a special feature space equipped with the so-called "kernel function". The kernel function type and the training rejection rate ν are the only parameters to be specified for training of OCSVM. The learned model comprises the center of the sphere c and the radius R.

The classification stage of OCSVM involves the calculation of the distance between the data point to be classified and the center of the hypersphere. If the distance is greater than R (the data point lies outside of the hypersphere), then it is considered an anomaly and is treated as benign. The radius thus serves as a threshold that is automatically determined at the training stage. The classification stage of OCSVM is illustrated in Fig. 3(b).

OCSVM cannot be directly applied to token sequences emitted by PJScan's feature extraction component. The reason for this is that OCSVM expects the data points to be numeric values lying in a high-dimensional space equipped with typical mathematical operations such as addition, multiplication with a constant and an inner product. Sequential data does not form such a space: it is not immediately clear how to add or multiply two strings. A solution to this problem involves a well-established technique of embedding sequences in metric spaces [20]. By counting the occurrences of substrings in data points and assigning the resulting numeric values to coordinate axes, the required mathematical properties can be enforced.

[14]The popular open source SVM implementation LibSVM [6], version 2.86, patched to support one-class SVM, was used in our experiments.

The embedding of sequences provides an elegant way for handling multiple JavaScript entities in the same file. To obtain an aggregated representation of all JavaScript entities, it suffices to add them using the addition operation provided by the embedding. To avoid the dependence on sequence length, the values in individual dimensions are binarized (by setting any positive values to 1) and normalized so that the Euclidean norm of the resulting vectors is equal to 1.

4. DATA COLLECTION AND ANALYSIS

The success of any learning-based approach crucially depends on the quality of data available for training. Likewise, the viability of a learned model can only be demonstrated on up-to-date real data. The evaluation of our method rests on an extensive dataset collected from the research interface to the popular malicious software portal VirusTotal. VirusTotal is a web service that enables ordinary users to upload suspicious files for a scan by 42 antivirus engines. Our dataset comprises 65,942 PDF documents with the total size of nearly 59GB. This data has revealed some interesting features, and is worth looking at in some detail.

We downloaded three batches of data on November 3, 2010, January 19, 2011 and February 17, 2011 each containing all PDF files available on VirusTotal at a given time. The data is kept only for 30 days, and there is surprisingly little overlap between subsequent months[15]. We have observed at most 200 identical files across different snapshots. We have split our corpora into two parts, the "detected" sub-corpus in which documents were flagged as malicious by at least one scanner, and the "undetected" sub-corpus containing supposedly benign data.

It is instructive to look at the statistical properties of our data presented in Table 1. One can notice interesting effects in the collected data. The average file size in the "detected" corpora (0.106MB) is about 13 times smaller than in the "undetected" ones (1.390MB). This shows that malicious PDF files do not contain a lot of meaningful content, which is confirmed by a manual investigation of some of these files. The percentage of files with JavaScript in the "detected" corpora (59.5%) is about 25 times higher than in the "undetected" corpora (2.4%). This is a strong indicator that JavaScript plays a crucial role in PDF-related exploits.

Considering only the files containing JavaScript one can see that the average number of JavaScript entities per file in the "detected" datasets (7.2) is around 33 times *smaller* than in the "undetected" datasets (241.1). This observation seems counter-intuitive but it turns out that "undetected" data usually contains hundreds of very simple JavaScript entities like `this.zoom=100;this.pagenum=39`. Similarly, distinctness of JavaScript entities at the code level is 3.2 times higher in "detected" corpora than in "undetected" (16.9% vs. 5.2%). These findings suggest that non-malicious usage of JavaScript in PDF documents essentially boils down to boring and redundant code!

Similar effects take place at the token level (the second row from the bottom in Table 1). One can observe a further decrease of distinctness (6,419 vs. 35,990, or 17.8%) due to lexical analysis. The "detected" sub-corpora are 7.5 times more distinct than the "undetected" sub-corpora. Finally, the last row in Table 1 reveals that many files contain identical sets of token sequences, which can be explained by common code reuse in both types of files.

To enable the quantitative evaluation of detection accuracy in the forthcoming experiments, we have manually labeled the "undetected" part of our data. Among 960 benign files with JavaScript,

[15]In fact, we were originally unaware of the 30 day lifespan and started a periodic collection of snapshots only in January.

	03. Nov. 2010		19. Jan. 2011		17. Feb. 2011		Total
	detected	undet.	detected	undet.	detected	undet.	
Dataset size	873MB	13GB	429MB	13GB	1.5GB	29GB	59GB
Files in the dataset	7,592	7,768	6,465	9,993	11,634	22,490	65,942
Files containing JavaScript	6,626	272	1,127	196	7,526	492	16,239
JavaScript entities	26,372	75,199	33,418	42,265	50,269	113,994	341,517
Distinct JavaScript entities	8,597	5,178	2,376	3,774	9,238	6,827	35,990
Distinct token sequences	1,108	429	815	356	2,947	764	N/A
Distinct files on the token sequence level	538	115	358	95	1,900	237	N/A

Table 1: Statistics of PDF documents collected from VirusTotal

we found 52 PDF documents that we believe to have been falsely classified as benign by all antivirus engines at VirusTotal. No cases were found where PDF files belonging to the same group of distinct files at the token level were assigned different labels.

5. EXPERIMENTAL EVALUATION

The real-world nature and the sheer size of the VirusTotal data make our evaluation especially challenging. First, the distinction between "detected" and "undetected" corpora is somewhat vague, as classifications by antivirus engines cannot be fully trusted. Second, the huge size of the "detected" corpus makes its manual analysis infeasible. On the other hand, the small size of the labeled JavaScript-bearing part of the "undetected" corpus is too small to be used for training purposes.

As the baseline for comparison we consider Wepawet, a web-based service based on JSand [7]. Wepawet performs both static and dynamic analysis of PDF files based on their JavaScript content and can detect malware that it has a signature for (labeled as *malicious*), as well as unknown malware (labeled as *suspicious*) using statistical features. In the evaluation, we treat both categories as detections. Similar to our system, Wepawet generally does not recognize PDF malware that does not use JavaScript. Table 2 shows Wepawet's classification on the "detected" and "undetected" parts of all three corpora at our disposal. In some cases file uploads were rejected by Wepawet, referred to as *fail*, or resulted in internal errors despite multiple submissions, referred to as *error*. We treat such cases (about 1.7% of the total data) as benign.

	03. Nov. 2010		19. Jan. 2011		17. Feb. 2011	
	det.	undet.	det.	undet.	det.	undet.
Fail	12	38	9	25	19	73
Error	15	1	5	0	83	0
Benign	3,860	212	502	167	1,050	397
Suspicious	1,474	11	149	0	257	0
Malicious	1,265	10	462	4	6,117	22

Table 2: Wepawet classification results

5.1 Objectives and Evaluation Criteria

Our experiments address the following questions:

1. How well do PJScan and Wepawet detect known malicious documents? This question may appear meaningless: why bother detecting something that is already detected? In practice, however, it is impossible to deploy all 42 antivirus engines from VirusTotal. For a single method, attaining the detection accuracy close to that of 42 established antivirus products is still a very challenging goal[16]. The corresponding quality measure is the *true positive rate on known attacks* TP_N defined as the ratio of the number of files in the "detected" corpus classified as malicious to the total number of JavaScript-bearing files in that corpus.

2. How well do both methods detect attacks that were missed by all 42 VirusTotal engines? We consider documents in the "undetected" corpus as novel attacks if they are classified as malicious during manual analysis. The *true positive rate on unknown attacks* TP_U is defined as the ratio of the number of files in the "undetected" corpus classified as malicious to the total number of malicious JavaScript-bearing files in that corpus.

3. How many normal documents are classified as malicious by the methods in questions? The *laboratory false positive rate* FP_L is defined as the ratio of the number of files in the "undetected" corpus classified as malicious to the total number of benign JavaScript-bearing files in the "undetected" corpus. The *operational false positive rate* FP_{OP} is the ratio of the number of files in the "undetected" corpus classified as malicious to the total number of benign files in the "undetected" corpus.

The distinction between the laboratory and the operational false positive rates is essential for estimation of the expected impact of false positives in practical deployment.

5.2 Experimental Protocol

Our experiments were carried out using the following procedure. We merged all 3 corpora from different dates keeping only the distinction between "detected" and "undetected" parts. We then randomly split the full "detected" corpus in two non-overlapping parts such that the corresponding sets of token sequences are of the same size. Due to a significant redundancy of token sequences this results in two sets of files that are different in size. One of these half-corpora is used to train PJScan, the other half is used to evaluate TP_K. To decrease the impact of non-determinism via random splitting, we repeat the experiment the second time by swapping the training and the evaluation datasets and averaging the detection accuracy. This process is known as *2-fold cross-validation*.

To determine the detection accuracy on unknown data, we apply the trained model on the full "undetected" corpus. We use the ground truth information to compute TP_U, FP_L and FP_{OP}. The reported results are also averaged over the two partitions of the training data.

[16]Unfortunately we cannot compare any method against the *best* detector at VirusTotal's. The labels in batch data from VirusTotal reflect only the number of detections but not the specific engines that classified a document as malicious.

Since the models used in Wepawet do not depend on our training data (but rather on the data its statistical part was trained on), the results presented for this method reflect the accuracy of scanning a complete respective dataset ("detected" or "undetected").

Some preliminary experimentation was needed to choose the parameters of OCSVM used in our method. We chose the training rejection rate $\nu = 0.15$ and the n-gram length of 4, which seem to provide the best trade-off between the true positive and false positive rates. The full results of our preliminary screening for optimal parameters cannot be presented due to space constraints.

5.3 Experimental Results

The results of a comparative evaluation of PJScan and Wepawet according to the criteria specified in Section 5.1 are presented in Table 3. Two configurations of PJScan were considered: using only native JavaScript tokens and using a set of additional heuristic tokens introduced in Section 3.2. It can be seen that PJScan significantly outperforms Wepawet on the known malicious data but performs less accurately on previously unknown attacks. Most of failed detections were caused by 11 files which are redundant at the token level and contain the following code[17]:

```
app.setTimeOut(this.info.dgu,1)
```

In this example, the attack code resides *not in a JavaScript entity* but in the *Info* dictionary[18]. It can be still accessed by a very short entry-point JavaScript code above as text and gets interpreted as JavaScript by calling the function `setTimeOut()` which is equivalent to `eval()`. With an exception of this kind of attack, the detection rate of PJScan would have also reached the 90% mark.

It is not clear to us why Wepawet has performed relatively poorly on known malicious data. In a related comparative evaluation against Cujo [19] in the context of web-based JavaScript attacks (drive-by-downloads), Wepawet was a clear winner with a detection rate of 99.8% compared to 94.4%. Most likely, the reason for worse performance of Wepawet in our experiments lies in technical problems with the extraction of JavaScript code from PDF documents.

Detection method	TP_K	TP_U	FP_L	FP_{OP}
PJScan (native tokens only)	84.80	71.15	16.35	0.3694
PJScan (with extra tokens)	85.17	71.15	17.35	0.3918
Wepawet	63.60	90.38	0.0	0.0

Table 3: Detection performance overview

A relative disadvantage of PJScan is the high false-positive rate. Measured against only the JavaScript-bearing benign documents it reaches the painful 16-17%; however, due to the rare presence of JavaScript code in benign documents, its operational false-positive rate remains acceptable and corresponds, for our data, to 1.7 false alarms per day.

One can also see that heuristic tokens do not improve the performance of PJScan and even lead to a slight degradation of the false-positive rate. The causes for this effect as well as for the false positives are elucidated in the following section.

5.4 Significant Features

As noted by Sommer and Paxson [22], a security practitioner would always be interested to know what a learning method has actually learned. The model created by the OCSVM (the center c of the sphere) produces a numeric ranking of essential features encountered in malicious JavaScript code. Since no benign data is used for training, this ranking does not reflect the differences between two classes but rather describes only one class known to it. Examples of the 5 most important and the 5 least important features in one of the models learned by PJScan (created for one half of the data) are shown in Table 4.

Although these features do not look particularly malicious, the top 5 features clearly correspond to typical lexical patterns of programming languages: member function dereferencing (Feature 1), string variable assignment (Feature 2), function calls (Features 3 and 4) and variable declarations (Feature 5). On the other end of the spectrum are the features that are obviously very atypical for programming languages.

The scoring of a new data point in the detection phase involves the identification of an overlapping subset of features between this data point and the learned model. The smaller the "weighted overlap" between the new point and the center (i.e. the sum of the weights in the model corresponding to the common features), the larger the distance from the center. This property is confirmed by the examples of accepted and rejected points presented below.

For the accepted points (Table 5, one true positive and one false positive), the main contributions are made by the top features of the trained model. Such points are virtually indistinguishable in our model, and this explains a high "laboratory" false positive rate observed in our experiments. It turns out, however, that *very few benign examples* share the "normal" programming language features captured by the learned model. For the two examples of rejected points (Table 6, one true negative and one false negative) the top features have much lower ranks in the learned models. The majority of benign examples have a small "weighted overlap" with the model and hence are rejected.

The investigation of significant features in our models suggests that the key property that enables effective discrimination between malicious and benign code in PDF documents is the fact that benign usage of JavaScript is very rudimentary from the programming point of view. Anecdotally, the benign example with the highest rejection score corresponds to the code `print(true)`.

5.5 Throughput

The throughput of PJScan was tested on a commodity PC with a quad-core Intel Core i7 860 CPU, 8 GB of RAM and a 7,200rpm SATA hard disk drive. Eight processes were run concurrently for performance measurement.

Each phase of PJScan was run on a respective data partition (training on one half of "detected" corpus, evaluation on the other half and on the full "undetected" corpus). Unlike the accuracy measurement, we learned and classified using *all* files ignoring their redundancy. Learning with thousands of files instead of a few hundred distinct token sequences reduces performance, but due to the fast learning and classification algorithms the difference is negligible. Processing times for all stages of our method are shown in Table 7. In total, parsing of 65,942 PDF files, tokenization of 341,517 JavaScript entities, learning on 15,279 "detected" files with JavaScript and classification of 960 "undetected" files with JavaScript took 1,547 seconds (about 25 minutes). All measurements are expressed in wall clock time[19].

[17] All examples differ in the name for the member of the `this.info` dictionary (in this case, `dgu`).

[18] An Info dictionary is used to store meta-data about the PDF file, such as author name, the software used to create it, etc.

[19] Wall clock time measures real time that elapses between the beginning and the end of a task. It includes CPU time, I/O time and any overhead such as the time process spends waiting for execution. It is a good indicator of real performance but is affected by system load.

Top 5			Bottom 5		
Rank	Weight	Feature	Rank	Weight	Feature
1	0.05285	`NAME . NAME (`	4051	2.285e-05	`) NAME ( THIS`
2	0.05106	`NAME ASSIGN STR ;`	4052	2.285e-05	`+- NAME !== NAME`
3	0.05092	`NAME ( NAME )`	4053	2.285e-05	`] ) - NAME`
4	0.04574	`( NAME ) ;`	4054	2.285e-05	`NAME ] ) -`
5	0.04314	`; VAR NAME ASSIGN`	4055	1.865e-05	`TRUE } ; IF`

Table 4: Features of the center point

True positive top 5			False positive top 5		
Rank	Weight	Feature	Rank	Weight	Feature
1	0.00456	`NAME . NAME (`	1	0.00554	`NAME . NAME (`
2	0.00441	`NAME ASSIGN STR ;`	2	0.00535	`NAME ASSIGN STR ;`
3	0.00439	`NAME ( NAME )`	5	0.00452	`; VAR NAME ASSIGN`
4	0.00395	`( NAME ) ;`	6	0.00413	`; NAME ( NAME`
5	0.00372	`; VAR NAME ASSIGN`	7	0.00386	`NAME ( STR )`

Table 5: Features of TPs and FPs

True negative top 5			False negative top 5		
Rank	Weight	Feature	Rank	Weight	Feature
7	0.00390	`NAME ( STR )`	1	0.01593	`NAME . NAME (`
8	0.00390	`VAR NAME ASSIGN NAME`	98	0.00490	`. NAME . NAME`
10	0.00359	`) ; NAME ASSIGN`	141	0.00394	`( THIS . NAME`
14	0.00338	`THIS . NAME (`	154	0.00372	`THIS . NAME .`
15	0.00338	`ASSIGN NAME . NAME`	355	0.00177	`NAME ( THIS .`

Table 6: Features of TNs and FNs

One can see that JavaScript extraction is the most time-consuming part of PJScan. It has been observed that this operation takes only 2,041 seconds using a single process, with an average processing time of 31 milliseconds per file. The overall CPU usage was very low (up to 40%, with I/O waiting of up to 30%), while at the same time disk utilization remained above 95% during the extraction phase. One can conclude that disk throughput represents the main performance bottleneck for this application. Using a faster storage device or reading files through a fast network is likely to improve the performance of PJScan.

The throughput calculation for different stages of PJScan is presented in Table 8. It shows that the throughput varies strongly between the "detected" and "undetected" files since they vary in file size and the number and size of JavaScript entities. The average throughput of 303.5Mbps is suitable for batch processing tasks even for organizations that have a very high volume of PDF traffic. The average processing time per file is 23 milliseconds. To the best of our knowledge, no other software package achieves lower processing times.

Time	Extractor	Tokenizer	Learner	Classifier
Total	1,356s	180s	10.19	0.014s
Average	0.0205s	0.0032s	N/A	0.000015s
Std. dev.	0.0015s	0.0392s	N/A	0.000009s
Percentage	87.65%	11.63%	0.66%	0.0009%

Table 7: Processing time for different stages of PJScan in batch execution mode

	Detected	Undetected	All files
Total time	339s	1208s	1547s
Average file size	0.106MB	1.39MB	0.89MB
Files per second	75.8	33.3	42.6
Data throughput	64.3Mbps	370.5Mbps	303.5Mbps
Seconds per file	0.013s	0.030s	0.023s

Table 8: Throughput characteristics of PJScan

6. RELATED WORK

As it was already mentioned in the introduction, detection of malware in PDF documents has not been extensively studied in the research literature before. The early approaches to identification of malware in PDF documents [13, 21] were based on *n*-gram analysis of *raw document content*. The scope of experimental evaluation in this work was rather limited. It included self-generated malicious PDF documents as well as a relatively small number of examples (less than 300) from the outdated VXHeavens malware repository. Due to the wide-spread use of evasion techniques in modern PDF malware, especially object compression and code-level obfuscation, we believe that the analysis of raw content of PDF documents is no longer adequate. Hence the approach taken in PJScan is fundamentally different from the above mentioned work in that our methods spend a lot of effort in discovering and utilizing the appropriate lexical features of PDF.

The recent work on analysis of PDF documents has emerged from existing tools for static and dynamic analysis. Besides the malware analysis portal Wepawet considered in our experiments, some other tools use a combination of static and dynamic analysis tools. MalOffice [10] uses static and dynamic techniques as well as some heuristics. Its static analysis is based on the utility pdftk[20], and the dynamic analysis builds on CWSandbox [26]. Detections are made by combining scores from various heuristics and policies attached to the analysis tools. Another combination of static and dynamic analysis was used in MDscan recently proposed in [24]. From the architectural point of view, MDscan is similar to our approach. It also uses static analysis to extract JavaScript content (using a self-made parser) and a heuristic approach for the extraction of JavaScript code. The extracted code is interpreted using SpiderMonkey. Detection is carried out at the dynamic stage by using the shellcode detection tool Nemu [16]. The method was evaluated on a set of 197 malicious PDF documents artificially generated using the Metasploit framework and 2000 benign documents. Compared to MDscan, we only use SpiderMonkey for token extraction and perform detection statically, which brings a performance improvement of two orders of magnitude.

A significant body of prior work has addressed the detection of malicious JavaScript in web content, especially in the context of drive-by-downloads. One cannot directly compare the accuracy of such methods with PJScan due to the fact that the data corpora used for the experimental evaluation of respective methods are very different. We will hence focus on methodical comparison of our approach with such methods.

Similar to PDF malware, the methods for detection of malicious JavaScript in web content can be classified into static, dynamic and hybrid. Purely dynamic methods deploy various techniques for monitoring the run-time execution of processes accessing web content, e.g.: full-fledged host virtualization [25], client virtualization [14], instrumentation of a JavaScript engine [11] or heap monitoring [18]. Dynamic methods have high detection accuracy and are hardly prone to false positives. Due to their performance overhead they are usually limited to "post-mortem" analysis.

Hybrid methods aim to minimize run-time overhead while retaining high detection accuracy. Several such methods have methodical affinity with PJScan. JSand [7] uses instrumented versions of the HtmlUnit[21], a Java-based browser simulator, and the Mozilla's Rhino[22] interpreter to extract heuristic features while monitoring the execution of JavaScript code. These features are used to train an anomaly detection system by running JSand on benign web pages. Cujo [19] is another interesting combination of static and dynamic methods. Its static part is similar to PJScan (with the exception of anomaly detection instead of two-class classification in its learning component); its dynamic component extracts symbolic features from a light-weight sandbox ADSandbox [9] and deploys similar *n*-gram analysis and learning techniques as the static part. A "mostly static" detection system Zozzle [8] attempts to avoid dynamic analysis but still needs it to unravel source-code obfuscation before using statistical feature extraction and supervised learning for the classification part. Compared to these hybrid methods, PJScan uses "reverse" anomaly detection—since only malicious data is widely available for PDF documents—and completely dispenses with run-time analysis. Another hybrid method has been proposed by Provos et al. [17]; however, the lack of a technical presentation in this reference prevents us from a detailed comparison.

The only method that can be classified as fully static is Prophiler [5] which deploys techniques similar to JSand except that its features are extracted from a JavaScript engine at the parsing stage *without running the code*. (A similar idea is used in our method but one step earlier, by stopping SpiderMonkey after the lexical analysis.) However, Prophiler has a high false positive rate and is intended to be used as a filter for a subsequent dynamic analysis.

[20]http://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/.

[21]http://htmlunit.sourceforge.net/

[22]http://www.mozilla.org/rhino/

7. DISCUSSION AND LIMITATIONS

The reported experimental results confirm the practical feasibility of the static, learning-based approach for detection of malicious JavaScript-bearing PDF documents. The preprocessing component of PJScan can be very helpful for a security administrator to manually extract and analyze JavaScript code in PDF documents. The main benefit of the learning component of PJScan is the ability to *extract knowledge from large-scale malware corpora*. PJScan enables one to derive light-weight models from heuristic knowledge of several dozen antivirus engines and tens of gigabytes of collected data. Such models can be deployed with no manual interaction and negligible performance overhead (<50ms per file). The operational false-positive rate of less than 0.4% is admissible in practice; even for a highly visible site like VirusTotal with a strong bias for suspicious data, this corresponds to an average rate of 1.7 false alarms per day (148 out of ca. 40,000 benign documents over 90 days).

The high "laboratory" false-positive rate of PJScan (i.e. the rate measured only for those benign files that contain JavaScript) indicates that our current learning setup may indeed have difficulty with accurate discrimination between malicious and benign JavaScript content. This observation is also indirectly supported by our analysis of the learned features. Learning from two classes, as it has been done in the related work on web-based JavaScript content, e.g. [19, 5, 8], may be the right way to avoid this limitation. However, benign JavaScript-bearing PDF data is currently not available in sufficient quantity to evaluate this scenario for PDF documents.

Another limitation of the current version of PJScan is its susceptibility to certain kinds of obfuscation. An exemplary obfuscation technique that is difficult for our method is the use of short JavaScript entry-point code which fetches further code from document locations where JavaScript code cannot be expected (cf. Section 5.3). There are two potential ways to address this limitation. One can use the "mostly static" technique proposed in Zozzle [8] in which compilation requests to a JavaScript engine are intercepted to obtain all code sent for execution. While this technique offers a guaranteed access to unobfuscated code, it may be hampered by just-in-time compilation used in JavaScript engines and eventually produce highly fragmented code. It should be also noted that this idea would be difficult to implement for PDF-based JavaScript code since Adobe provides an extensive PDF-specific API. Another way of dealing with obfuscation is to collect syntactic information from a parser and use compiler optimization to factor out obfuscations.

An attacker may also attempt to use the fact that the models used for detection are derived from data. A taxonomy of attacks against learning algorithms has been recently proposed by Barreno et al. [4]. Following this taxonomy, we remark that causative attacks, i.e. attacks against the training data, do not constitute a serious threat to our approach. We use data from an established malware repository and assume that integrity of this repository cannot be compromised. Even if an attacker submits own data to this repository, he will know how this data is classified by antivirus engines but cannot influence this classification. More realistic are attacks from the exploratory category, i.e. attacks staged at the detection stage. One potential attack strategy is to insert some useless code to make a new JavaScript entity look "anomalous". This attack may indeed be quite potent if an attacker knows the true profile of "normal" malicious data. Since he neither has access to nor can manipulate the training data, we believe that in practice guessing what kind of useless code should be added can be a difficult task.

8. CONCLUSIONS

We have proposed a new static approach to detection of malicious JavaScript-bearing PDF documents. The main advantages of our approach are its high performance and no need for special instrumentation, such as virtual machines or sandboxing. It can attain about 85% of the detection accuracy of *all* antivirus engines at VirusTotal with the performance overhead of less than 50ms per file. It is only marginally affected by text-level obfuscation since the resulting JavaScript code remains very conspicuous at the lexical level. Due to these advantages our method can be used as a standalone application on end-user systems or even be integrated as a filtering tool in email gateways and HTTP proxies.

The computational efficiency of our system PJScan has enabled us to evaluate it on an *unprecedentedly large* real-life data corpus (over 65,000 PDF documents) collected from VirusTotal. This evaluation has confirmed a high detection accuracy of our method for both known and unknown malware. PJScan is more prone to false positives than state-of-the-art dynamic approaches; however, its operational false positive rate still lies in the promille range, which is feasible for practical deployment.

Our future work will address a potential interaction of static and dynamic analysis techniques in order to unravel code-level obfuscation typical for JavaScript attacks. We anticipate that some degree of dynamic analysis can be carried out prior to actual code execution without a significant performance overhead. We also intend to investigate more extensive static analysis techniques, such as syntactic analysis and compiler optimization, to obtain features that better reflect the true semantics of the JavaScript code. Finally, an important open issue remains the detection of malicious PDF documents whose exploitation techniques do not rely on JavaScript.

Acknowledgements

The authors wish to acknowledge finantial support by the Heiseberg Fellowship of the Deutsche Forschungsgemeinschaft (DFG) and by the German Federal Office for Information Security.

9. REFERENCES

[1] Internet security threat report. Symantec, 2010.

[2] APSB11-03. http://www.adobe.com/support/security/-bulletins/apsb11-03.html.

[3] APSB11-08. http://www.adobe.com/support/security/-bulletins/apsb11-08.html.

[4] M. Barreno, B. Nelson, A. Joseph, and J. Tygar. The security of machine learning. *Machine Learning*, 81(2):121–148, 2010.

[5] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: a fast filter for the large-scale detection of malicious web pages. In *International Conference on World Wide Web (WWW)*, pages 197–206, 2011.

[6] C.-C. Chang and C.-J. Lin. *LIBSVM: a library for support vector machines*, 2001. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`.

[7] M. Cova, C. Kruegel, and G. Vigna. Detection and analysis of drive-by-download attacks and malicious JavaScript code. In *International Conference on World Wide Web (WWW)*, pages 281–290, 2010.

[8] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert. ZOZZLE: Fast and precise in-browser javascript malware detection. In *USENIX Security Symposium*, 2011. to appear.

[9] A. Dewald, T. Holz, and F. Freiling. ADSandbox: sandboxing JavaScript to fight malicious websites. In *Symposium on Applied Computing (SAC)*, pages 1859–1864, 2010.

[10] M. Engelberth, C. Willems, and H. T. MalOffice – analysis of various application data files. In *Virus Bulletin International Conference*, 2009.

[11] B. Feinstein and D. Peck. Caffeine Monkey: Automated collection, detection and analysis of malicious JavaScript. In *Black Hat USA*, 2007.

[12] K. Itabashi. Portable document format malware. Symantec white paper, 2011.

[13] W.-J. Li, S. Stolfo, A. Stavrou, E. Androulaki, and A. Keromytis. A study of malcode-bearing documents. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 231–250, 2007.

[14] J. Nazario. PhoneyC: a virtual client honeypot. In *USENIX Workshop on Large-scale Exploits and Emergent Threats (LEET)*, 2009.

[15] PDF Reference. http://www.adobe.com/devnet/pdf/pdf_reference.html, 2008.

[16] M. Polychronakis, K. Anagnostakis, and E. Markatos. Comprehensive shellcode detection using runtime heuristics. In *Annual Computer Security Applications Conference (ACSAC)*, pages 287–296, 2010.

[17] N. Provos, P. Mavrommatis, M. Abu Rajab, and F. Monrose. All your iFRAMEs point to us. In *USENIX Security Symposium*, pages 1–16, 2008.

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

[19] K. Rieck, T. Krüger, and A. Dewald. Cujo: Efficient detection and prevention of drive-by-download attacks. In *Annual Computer Security Applications Conference (ACSAC)*, pages 31–39, 2010.

[20] K. Rieck and P. Laskov. Linear-time computation of similarity measures for sequential data. *Journal of Machine Learning Research*, 9:23–48, 2008.

[21] Z. Shafiq, S. Khayam, and M. Farooq. Embedded malware detection using markov n-grams. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, pages 88–107, 2008.

[22] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *IEEE Symposium on Security and Privacy*, pages 305–316, 2010.

[23] D. Tax and R. Duin. Support vector data description. *Machine Learning*, 54:45–66, 2004.

[24] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *European Workshop on System Security (EuroSec)*, 2011.

[25] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated web patrol with strider honeymonkeys: Finding web sites that exploit browser vulnerabilities. In *Network and Distributed System Security Symposium (NDSS)*, 2006.

[26] C. Willems, T. Holz, and F. Freiling. CWSandbox: Towards automated dynamic binary analysis. *IEEE Security and Privacy*, 5(2):32–39, 2007.

[27] J. Wolf. OMG WTF PDF. Chaos Communication Congress (CCC), Dec. 2010.