# Caradoc: a pragmatic approach to PDF parsing and validation

Guillaume Endignoux, Olivier Levillain, Jean-Yves Migeon

# Caradoc: a pragmatic approach to PDF parsing and validation

Guillaume Endignoux
*École Polytechnique*
*École Polytechnique Fédérale de Lausanne*

Olivier Levillain
*Agence Nationale de la Sécurité*
*des Systèmes d'Information*

Jean-Yves Migeon
*Agence Nationale de la Sécurité*
*des Systèmes d'Information*

*Abstract*—PDF has become a *de facto* standard for exchanging electronic documents, for visualization as well as for printing. However, it has also become a common delivery channel for malware, and previous work has highlighted *features* that lead to security issues.

In our work, we focus on the *structure* of the format, independently from specific features. By methodically testing PDF readers against hand-crafted files, we show that the interpretation of PDF files at the *structural level* may cause some form of denial of service, or be ambiguous and lead to rendering inconsistencies among readers. We then propose a pragmatic solution by restricting the syntax to avoid common errors, and propose a formal grammar for it. We explain how data consistency can be validated at a finer-grained level using a dedicated type checker. Finally, we assess this approach on a set of real-world files and show that our proposals are realistic.

## 1. Introduction

PDF – Portable Document Format – dates back to the early 1990's and became the ISO 32000-1 specification in 2008 [10]. It is now a very common way to exchange, display and print documents. PDF is also a widespread channel for malware delivery since, contrary to popular belief, the format is featureful and allows to embed various types of active content (JavaScript, Flash). These features can lead to vulnerabilities, either by themselves, or through flaws in the embedded interpreters, and they have been studied in previous work. Yet, we noticed that the structural layer of PDF files has not received wide consideration, even if the standard is complex and encourages lax behavior in PDF-handling software. The parsing stage is often overlooked and only the subsequent steps, starting with an abstract syntax tree, are studied.

Our work thus focuses on the syntax and the structure of PDF files. First, we explore ambiguities in the specification and, by using crafted files, we test how common software behaves. In the course of this study, we uncovered several novel bugs and found inconsistencies among PDF readers. To overcome these problems, we propose a stricter interpretation of the specification. The proposed rules were implemented in a tool, CARADOC, and we show that they

can be applied to real-world PDF files. In some cases, a normalization stage is initially required.

Contrary to most of previous studies, we endeavor to follow a "whitelist" approach, since our goal is to propose a subset of PDF that we are able to reliably parse and analyze, instead of trying to identify or remove unsafe parts of PDF files (the corresponding "blacklist" approach). Reliable parsing is indeed a necessary step for reliable higher-level analyses. Our contributions in this paper are threefold:

- uncovering bugs and inconsistencies in common PDF readers (bugs were reported and are currently under review by software editors [6], [7]);
- proposing a *restricted* version of the specification, including new validation rules;
- implementing this proposal and discussing the first results (the tool, named CARADOC is available as an open source project on GitHub[1]).

Section 2 presents the file format and some of its pitfalls. Then, we examine known problems and their corresponding solutions in section 3. Section 4 further investigates new issues that we uncovered regarding the low-level structure of PDF files. Next, we describe in section 5 our proposal to restrict PDF using additional rules at syntactic and semantic levels. Section 6 presents CARADOC, our implementation of these rules, as well as the first results that we obtained. Next, we propose some perspectives in section 7. Section 8 compares our approach to some related work.

## 2. PDF: a quick tour

In this section, we introduce the key elements that constitute a PDF file, from the syntax to the document structure.

### 2.1. Syntax of objects

A PDF file is made of objects describing the various parts of the document. For example, a page, an image or an entry in the table of contents are objects.

**Character set.** In its simplest form, a PDF file is written as a text file, mostly made of alphanumeric characters and some special characters.

---

1. https://github.com/ANSSI-FR/caradoc

Like many textual formats, syntactic elements are separated by *whitespace* characters. These characters are interchangeable and do not have a special meaning – except for *newline* characters in some contexts.

PDF files can also contain comments that are semantically equivalent to whitespace. They begin with the '%' character and last until the end of the line.

**Direct objects.** PDF objects are made of a few basic types:

- *null*, represented by the keyword `null`;
- *booleans*, represented by `true` and `false`;
- *integers*, written as sequences of digits, like `123`;
- *reals*, digits separated by a period, like `-4.56`;
- *strings*, between parenthesis, like `(foo)`;
- *names*, introduced by a slash, like `/bar`;
- *arrays*, whose elements are space-separated between brackets, like `[1 2 3]` or `[(foo) /bar]`;
- *dictionaries*, associative tables made of key-value pairs enclosed in double angle brackets, like `<< /key (value) /foo 123 >>`.

**References.** It is also possible to define shared objects as *indirect references*. An *indirect object* is identified by an *object* number and a *generation* number. The latter is used to keep track of updates and starts with 0 for a new object. The following example binds the pair (object No. 15, generation No. 3) to the object *foo*, using keywords `obj` and `endobj`.

```
15 3 obj
```
*foo*
```
endobj
```

This object is accessible elsewhere in the document using the syntax `15 3 R`.

**Streams.** A *stream* is a combination of a dictionary and a byte sequence. It is necessarily an indirect object and has the following format.

```
1 0 obj
```
*<< stream dictionary >>*
```
stream
```
*byte sequence*
```
endstream
```
```
endobj
```

The dictionary contains metadata and allows to use compression *filters*. Using streams, images can come with their dimensions as metadata and be compressed with suitable algorithms.

## 2.2. File structure

**Basic structure.** In its simplest form, a PDF file is made of five elements (see Fig. 1):

- a *header* that identifies the PDF file and its version;
- a *body* that contains indirect objects;
- a *cross-reference table* (a.k.a. *xref table*) that gives the positions of objects in the file;
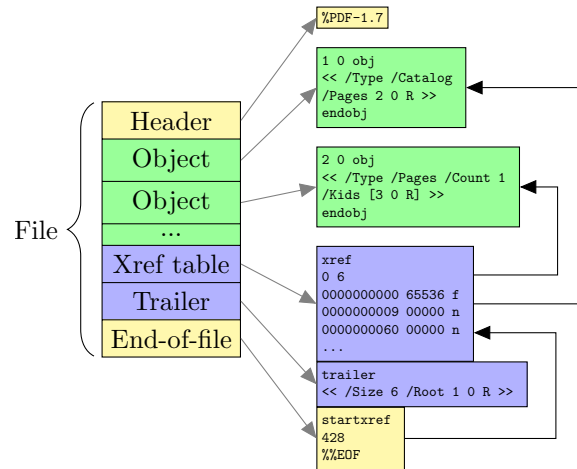


Figure 1. Structure of a simple PDF file.

- a *trailer* that indicates the root of the document;
- an *end-of-file marker* that indicates the position of the xref table.

**Incremental updates.** It is possible to quickly modify a PDF file by appending new content after the end of the file, instead of rewriting it entirely. To this aim, the following elements are added to the basic structure: the objects introduced by the update, a new xref table to reference them and a new trailer. The new table points to the first table, which gives access to the older objects (see Fig. 2). Updates can also mark objects as *free* when they are not used anymore.
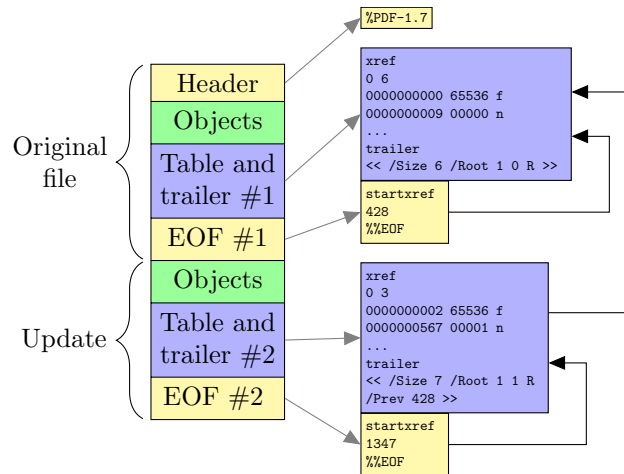


Figure 2. Updated PDF file.

**Linearized file.** The basic structure of PDF is problematic in a network context when one wants to display the content of a file during downloading, as critical information is present at the end of the file. PDF 1.2 thus introduced a new structure called *linearized PDF*, which adds reference tables – called *hint tables* – at the *beginning* of the file. Such tables give the positions of objects required to display each page.

**Object streams.** In the basic format, objects are stored one after another, uncompressed. Since they use a textual format, the size of the document is not optimized. For this reason, the notion of *object stream* was introduced in PDF 1.5. It is a stream that contains other objects (Fig. 3). The benefit is that streams can be compressed.
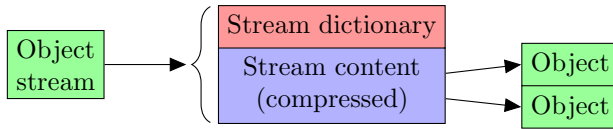


Figure 3. Object stream layout.

## 2.3. Document structure

From a semantic point of view, objects are organized into a directed graph whose edges are *indirect references*. The root of this graph is the *trailer*, that contains several metadata and refers to the *catalog*.

The rest of the document is referenced by this catalog. The most common parts are:

- *pages*, that contain the graphical content to be displayed or printed;
- *outlines*, that form a table of contents for an easier navigation among pages;
- *name dictionaries*, that provide a way to reference objects via *names* instead of *references*;
- information for the viewing software, such as the *page layout* or *page labels*;
- various metadata, that optionally give further information about the document.

**Page tree.** The graphical content of a PDF document is made of pages, organized into a tree. This structure enables inheritance of attributes – e.g. page dimensions – among ranges of pages. Pages are the leaves of this tree, as shown on Fig. 4.
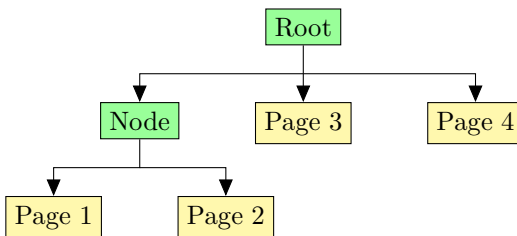


Figure 4. A simple page tree.

**Description of pages.** In order to provide a rich set of features, each page of a PDF document can be associated with a lot of objects. The most common ones are shown on Fig. 5.

The *content stream* contains a sequence of commands describing the graphical content of a page: text, geometric shapes, colors, images, etc.

The page is bound to *resources* – such as fonts and images – that are not directly stored in the content stream. The content stream can refer to them by means of *names*.

Interactive content is described by *annotations*. For example a page may contain a hyperlink: when the user clicks inside a rectangle, they are redirected to a URI. An *annotation* connects the geometric shape to the *destination*, but is dissociated from the text of the link (which is stored in the content stream).
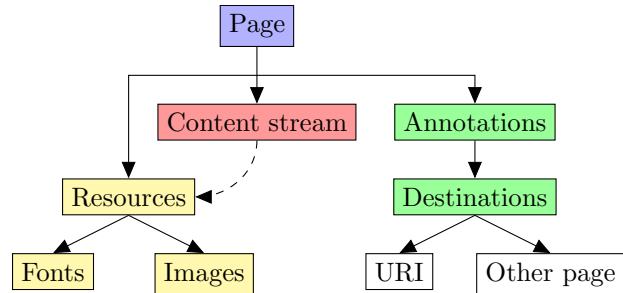


Figure 5. Common objects associated with a page.

## 3. Known problems and available tools

### 3.1. Flaws in security policies

Even if the specification has been available since 1993, PDF has not really been studied from a security point of view until the late 2000's. In 2007, Filiol et al. [9] presented some features that are possibly unsafe in a PDF file: automatic actions upon opening the document, execution of arbitrary commands on the user's machine, submission of forms on a remote URL, execution of JavaScript code, etc. They pointed out flaws in the security policy of ADOBE READER. In fact, some critical actions – such as Internet access – usually require a confirmation from the user by means of a dialog window. Yet, the configuration allowed to bypass this confirmation process, e.g. by adding a domain name to a whitelist. These flaws could in particular be used in *phishing* attacks.

Raynal et al. [28] had a similar approach and gave an in-depth analysis of the security and trust management mechanisms implemented by ADOBE READER. They pointed out that some features – such as JavaScript code – could be declared in privileged mode, which allowed them to run without user action. This privilege is based on a certification system and the authors noted that some of Adobe's software contained the private key used to sign privileged content. Thus, an attacker could extract this key and sign a file containing malicious content.

### 3.2. Polymorphic files

One of the biggest issues of current PDF readers is that they accept flawed files and try to correct the errors by themselves. Wolf [36] noted that ADOBE READER accepts a file containing the header in the first 1024 bytes, instead of

at the very start of the file. The cross-reference table can also be invalid or even missing without causing any warning!

This lax behavior is accounted for compatibility reasons with existing PDF files. However, one can object that this does not promote a strict behavior on the PDF-writing software side. Consequently, competing PDF readers have to accommodate these errors if they want to stay in the race.

Raynal et al. [28] produced *polyglot* files, that are at the same time *valid* PDF files – or more precisely files accepted by PDF readers – and valid files of another format, such as JPEG. The trick is to store the PDF content inside a comment field of the JPEG file, with the PDF header in the first 1024 bytes of the file.

In the same spirit, Albertini published numerous variants of PDF files that are also – among others – valid ZIP, Flash, HTML or Java files [2], [3]. He also gave examples of *schizophrenic* PDF files, that display different content with different PDF readers [1].

This polymorphism can lead to security problems. Jana et al. [11] have shown that malware detectors are often fooled by files having multiple "types", because they only identify one of these types. For example, an attacker could embed a malicious PDF inside a JPEG file. If an antivirus only identifies the JPEG type, it will see an innocuous file and accept it.

The fundamental issue here is that the question "Is this file safe?" is much harder to answer than the question "Is this file, considered as X, safe?" Specifications can help avoiding type collision by requiring a fixed magic number at the beginning of files, but this rule needs to be strictly enforced. An alternative approach is to use a typed filesystem to keep track of file formats, as was proposed by Petullo et al. in Ethos operating system [27].

### 3.3. Analysis of JavaScript content

Since version 1.3, PDF documents can contain JavaScript code. A lot of vulnerabilities were discovered and often exploited by malware in the JavaScript interpreter of ADOBE READER. For example, CVE-2014-0521 showed how to leak sensitive information such as the content of an arbitrary file. Hence, several studies focused on extracting and analyzing JavaScript code in PDF files.

In this context, Laskov et al. presented a tool to detect malicious JavaScript code, called PJSCAN [19]. The extraction process was based on the POPPLER library, a free implementation of PDF used by the majority of PDF readers on Linux. This can lead to problems regarding the reliability of extraction, as we will see in section 4. Their static code analysis was based on statistical methods aimed at recognizing patterns similar to known malware.

Tzermias et al. have developed a dynamic tool called MDSCAN [35]. JavaScript code extraction was only based on the presence of a /JS key in some objects. The code was then emulated and the memory was analyzed to find malicious patterns. The authors pointed out that they used the free JavaScript engine SPIDERMONKEY and that there were idiosyncratic differences with the implementation of ADOBE READER. These implementation disparities can lead to misinterpretation and false negatives.

In his master thesis [30], Schade presented FCSCAN. He pointed out the inherent problems of emulation: idiosyncrasies of a specific interpreter, PDF-specific API extending the JavaScript language, timing differences between emulation and real software... He thus proposed a more generic analysis of function calls to detect uncommon patterns.

### 3.4. Statistical methods

In the field of computer security, several studies have explored *machine learning* methods [13]. Some of them focused on PDF files.

Maiorca et al. presented PDF MALWARE SLAYER [23], a tool that extracts several features from a PDF file, such as the frequency of some keywords. This set of features was then compared to a database of benign and malicious documents with a statistical classification method. Smutz et al. [33] had a similar approach, but also used more exotic features such as the total number of pixels in images or the length of the /Author attribute. Their tool PDFRATE is a web service where one can send files to analyze.

However, these methods have several flaws. In fact, they rely on the principle that malicious files are significantly different from benign ones. The authors of these tools conceded that a *mimicry* attack was possible, where an attacker modifies its malicious file so that it resembles a benign one. The authors proposed various methods to counter this attack, for example by modifying the feature set.

Biggio et al. conducted a more in-depth study of a *mimicry* attack from a mathematical point of view, and applied it to PDF files [4]. They noted that it was easy to add content to a PDF file in order to manipulate its features, e.g. with an update (see Fig. 2). Laskov et al. [18] proposed to add content directly after the cross-reference table but before the end-of-file marker. Such content is then ignored by PDF readers, but not by a simple parser such as PDFRATE.

Maiorca et al. presented a method of *reverse mimicry* [22]. Instead of starting with a malicious file, they proposed to start with a benign file and add malicious content to it, which also proved to be an efficient evasion technique.

### 4. Structural problems: a new avenue

In 2014, Bogk et al. attempted to write a formally verified parser with the proof assistant COQ [5]. They pointed out ambiguities in the format, detected with the help of COQ. While PDF allows to write several incremental updates successively, the specification does not clearly set constraints against making a recursive structure with two updates, where each update refers to the other one as the *previous* update (see Fig. 6). In particular, an update is not required to be located before its predecessor in the file.

To forbid such recursive structures, Bogk et al. chose to register the positions of tables that have already been
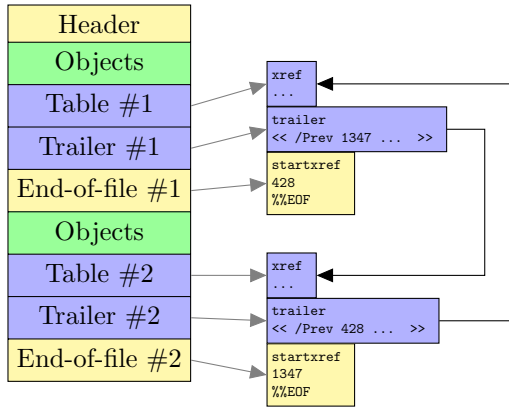
Figure 6. Loop in updates.



Figure 7. The outline hierarchy.



Figure 8. Example of an incorrect structure.

explored, at the cost of a more complex proof – adding 500 lines of code. Besides, their parser only handled a basic subset of the syntax and did not validate compression nor relationships between objects. Writing a feature-complete PDF parser verified in COQ seems to be a daunting task.

As a matter of fact, many sub-structures of a PDF document are organized into directed acyclic graphs. However, as with the cross-reference example, PDF readers rarely enforce the expected properties with all necessary checks. In particular, cyclic structures can trigger infinite recursion on most readers.

With that in mind, we forged several PDF files, containing either malformed elements, or valid elements with an unclear interpretation. Some of these files triggered interesting behavior in PDF readers. We present several examples of problems that we found and a summary of the obtained results.

### 4.1. Trees and graphs

The document *outline* is an optional feature allowing to display a table of contents, usually shown on the side of the reader. Each element contains a link to a position in the document, for example to access chapters. This table of contents is organized as a tree structure that contains several levels of hierarchy. The parent-children relationship is stored by means of a doubly-linked list. The parent contains references to the first and last children, and each child links to the previous and next children, as Fig. 7 shows.

Yet, as with the cross-reference tables, it is possible to create an incorrect hierarchy that is not a tree. We tested ill-formed *outlines* containing various forms of cycles – such as Fig. 8 – against several PDF readers. Most of them looped indefinitely for at least one file.

We believe that the specification is faulty here, because the intended structure is a tree, but neither the data structure nor the implementation notes explicitly enforce it. Also, a simpler structure, for example with a flat array instead of a doubly-linked list, would have been more appropriate and easier to parse in practice.
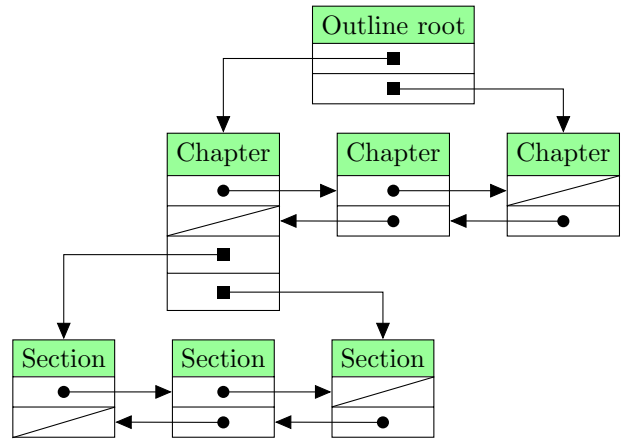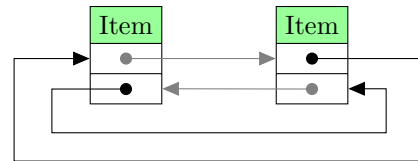
### 4.2. An idol with feet of clay

High-level analyses are based on the proper decoding of the file at syntactic and structural levels. If the parser is not reliable, these analyses become irrelevant. Some mimicry attacks already exploited discrepancies between the library used for the analysis and the one used for rendering. That is why we tested PDF readers on files containing low-level errors. Here are two meaningful results on *dictionaries*, that constitute the backbone of the PDF structure because the majority of complex objects are built upon them.

First, the specification states that all keys shall be unique in a dictionary. For example, the following structure is invalid because the key /Key is present twice.

```
<<
    /Key (value 1)
    /Key (value 2)
>>
```

In practice, none of the tested software reported an error, but instead, all of them associated the key /Key with the second value (value 2). However, other parsers might interpret it differently.

Second, a problem arise from escape sequences, that allow to use any characters in *names*, including special characters. The pattern #xx, where xx are hexadecimal digits, is replaced by the byte with the corresponding code. If the '#' character is not followed by two hexadecimal digits, the escape sequence is invalid and an error should occur[2], but most readers do not report it. Even worse,

---

2. Here again, the standard does not specify a clear behavior to adopt. One could reject the character, the name, the object or even the entire file.

the pattern `/Foo#7/Bar` has two possible interpretations: some readers see one *name* token whereas others see two *names*: `/Foo#7` and `/Bar`. This allowed us to create a file with totally different interpretations among readers.

### 4.3. Generation numbers

As described in section 2.1, indirect objects are identified by a pair which consists of an *object* number and a *generation* number. However, the generation number does not seem to have any semantic meaning and it is often set to zero in practice. Thus, we wanted to investigate the behavior of PDF readers regarding these numbers.

**Incorrect number.** When no object corresponds to the pair of numbers, some readers discard the generation number and look for an object with the same object number.

```
% undefined generation number
7 1 R
% targeted object
7 0 obj ... endobj
```

**Negative number.** Generation numbers must be between 0 and 65535 – inclusive. Thus, we tested negative numbers. Most readers incorrectly recognized the targeted object in the following example.

```
% negative generation number
7 -2 R
% targeted object
7 -2 obj ... endobj
```

**Special value.** The PDF reference [10] states that *free* objects – i.e. objects that have been deleted – must have a generation number of $65535 = 2^{16} - 1$. However, it is not really clear whether this value should be allowed for in-use objects. We tested the following example and some readers effectively identified a deleted object whereas others recognized our target.

```
% special generation number
7 65535 R
% targeted object
7 65535 obj ... endobj
```

### 4.4. A summary of problematic behaviors

In this study, we compared the behavior of several PDF readers: ADOBE READER, EVINCE, FOXIT READER, SUMATRAPDF, as well as the embedded viewers of CHROME and FIREFOX browsers. This list was not intended to be exhaustive and other readers might be affected as well.

We tested several elements of the PDF syntax and structure, such as those presented in the previous sections. For each element, we crafted ill-formed files and compared the behavior of PDF readers. This included the displayed content, the CPU usage and whether the software crashed.

This study uncovered many problems leading to denial of service or rendering discrepancies. Fig. 9 summarizes the kinds of problems that we found. Details about these bugs have been reported to software editors.

| Description | Count |
|---|---|
| Tested features | 17 |
| At least one reader crashed | 4 |
| At least one reader exhausted all CPU resources | 3 |
| Inconsistencies between readers | 12 |
| Non-conforming behavior w.r.t the standard | 13 |

Figure 9. Test of syntactic and structural errors.

**Discussion.** In our study, we did not find any open source PDF parser relying on lexer and parser generators. Instead, each implementation is written as an ad hoc parser, which is more error-prone and leads to the aforementioned discrepancies. It is worth noting that the PDF specification describes the format in a natural language in English text but does not provide a formal grammar in Backus-Naur Form – or similar. For all these reasons, we decided to propose such a (restricted) grammar, as well as additional rules to build a clear and unambiguous description of what a PDF file should be.

## 5. Towards a PDF/Restricted format

Reliably interpreting a PDF file raises many issues, and an exhaustive validation is a difficult task. To address this problem, we chose to focus on the validation at syntactic and structural levels, in order to provide a solid basis for higher-level verifications.

Because the full PDF format is very complex, we first propose a restriction of the syntax that is easier to validate, and show how files can be rewritten to conform to this restricted format. We then present a type checking algorithm that allows to validate the consistency of each object. Finally, we describe how we can check the overall consistency of more complex structures.

### 5.1. Restriction of the syntax and normalization

**Motivation.** In the previous sections, we discussed how PDF readers try to *correct* invalid files, and how it is possible to create polymorphic files. To avoid these problems, it is essential to guarantee that objects can be extracted unambiguously from PDF files.

Moreover, some features of the language – such as incremental updates or *linearization* – add complexity to PDF parsers but are not essential to the format expressivity.

For these reasons, it seems natural to define a new format to simplify the validation process[3]. Two properties are desirable:

---

3. This format could be called PDF/R for PDF Restricted, in the same spirit as PDF/A for PDF Archive.

- it should be a *restriction* of PDF, so that validated files are directly usable without further processing[4];
- it should be compatible with the most common features of PDF, to have practical use.

**Definition of a grammar.** As is, PDF is error-prone and it is not easy to verify that a parser is correct. The main reason is that the cross-reference table, that indicates the position of objects, is located at the end of the file. This prevents implementations from parsing the file linearly because distinct objects could overlap, holes could be inserted with unknown content between objects, etc.

An elegant approach is to define an unambiguous grammar and use lexer and parser generators, because verifying a grammar is easier than verifying a custom parsing code. Furthermore, using a grammar is language-agnostic and can easily be adapted to various projects.

For these reasons, we propose a grammar that makes it possible to parse files linearly, but only supports a restriction of the PDF language. The goal is to define an $LR(1)$ grammar, to use tools such as MENHIR – a parser generator for OCAML. We could have targeted $LALR(1)$ grammars as well – to use YACC or GNU BISON.

**Grammar rules.** Our intent was to parse the most basic format shown on Fig. 1, i.e. files that are not *linearized* and that contain neither *incremental updates* nor *object streams*. These files can be processed linearly, which means that objects can be extracted without the help of the cross-reference table. We then have to check the consistency between declared positions in the cross-reference table and real positions of objects in the file.

Apart from constraining the overall structure of the file, we decided to add other restrictions. At a syntactic level, we remove *comments*, because they can contain arbitrary content that allows to create polymorphic files (see [1]). Besides, comments are mostly useless – humans seldom read the raw content of a file – and they have no meaning for a software – or at least they should not!

Likewise, we restrict the set of allowed *whitespace* characters – there are six of them in PDF. We did not find errors related to them, but they are all equivalent – except newline characters that are required in some contexts – and it is possible that an implementation forgets one of them or confuses the *NULL* character for an end-of-string marker. That is why we only allow four *whitespace* characters: *space*, *tabulation*, *line feed* and *carriage return*.

**Practical issues.** Defining an $LR(1)$ grammar raised some practical issues. The main reason is how *indirect references* are written. For instance, the `[1 2 R]` array contains *one* element (a reference), whereas the `[1 2]` array contains *two* elements (integers).

In that case, after reading tokens `1` and `2`, we do not know if we are in the middle of a reference or if we have

already read two integers. For this reason, we have to use non-trivial rules to discriminate integers from references. Hopefully, this problem solely occurs in arrays because it is the only place where a sequence of integers is allowed.

**Additional rules.** Aside from the grammar, we add constraints to guarantee the file consistency. For example, the positions in the cross-reference table must point to the corresponding objects in the file. The `/Length` value of streams must match the number of bytes between `stream` and `endstream` keywords.

Regarding *indirect objects*, we force the *generation numbers* to be zero, to avoid the problems presented in section 4.3. Besides, we require all objects to be used – except for object number zero that is never used. To further simplify the structure of the cross-reference table, we require that object numbers form a single continuous range starting from zero.

Last, we restrict the encoded content of *streams*, forbidding them to contain keywords of the PDF language, such as `endstream` or `endobj`. Otherwise, parsers might be fooled if they rely on `endstream` and do not check the length of streams, as Wolf noted [36]. However, this restriction should not be a practical issue because it is always possible to rewrite a stream with a *filter* to get rid of these sequences; for example `/ASCIIHexDecode` produces only hexadecimal characters.

**Normalization.** In practice, our subset of the PDF language is too strict for most PDF files. A solution was to create a *normalization* tool that accepts more files and rewrites them in a simpler form (no incremental update, no comments, etc.). Of course, such a tool is more difficult to verify, but our goal was to trust only the validation tool, which works on the restricted PDF language. A practical use case would be to run the normalization tool in an isolated environment – for example before a file enters a sensitive network – and then to check the file in the classical user environment – for example each time the user opens a file with a PDF reader.

## 5.2. Type checking

**Motivation.** Once we are confident about the overall file structure, we have to check that objects are semantically consistent. For example, we expect the leaves of the page tree to be pages, not images or integers. Unknown problems might arise when types are inconsistent.

Besides, we want to be able to set up a whitelist of allowed features, to avoid malicious content. To ensure this, it is first necessary to know the type of every object.

Finally, we must mention that object types are not obvious. In fact, dictionaries may contain a `/Type` attribute, but this is optional in most cases. Thus, we could not merely iterate over objects to find their types. It was necessary to infer types from the context in which objects are used.

**Algorithm.** A PDF document forms a connected graph, whose edges are indirect references and whose root is the

---

4. This property excludes new formats designed from scratch, as the *Simple Representation* proposed by Rutkowska for Qubes OS [29].

trailer. To infer the types of objects, we do a breadth-first traversal of the graph, starting with the trailer. We propagate type constraints through references until we have identified and checked every object.

During the traversal, each object is in one of three possible states: it is either *unknown*, *inferred* or *checked*. The type of an object is in *inferred* state if constraints from other objects require a given type, but we have not checked its content yet – i.e. this object is in the queue of the breadth-first traversal. At the beginning, the state of the trailer is *inferred*, and other objects have an *unknown* state. The goal is that all objects end up in the *checked* state after the propagation of constraints.

If several type constraints are conflicting or if an object does not comply with its *inferred type*, we stop the algorithm and reject the file.

Since we perform a breadth-first traversal of the graph, the complexity of our type checking algorithm is linear in the number of edges.

**Handling transitional ambiguities.** Sometimes, we cannot infer the exact type of an object until we actually check it. For example, the page tree is made of two different types of objects: leaves (pages) and internal nodes (page intervals). A priori, a child of an internal node can be of either type so it cannot be inferred directly when the parent is traversed.

Thus, we infer this object as a *sum type*, a superposition of several types that will be resolved at checking time. It is indeed possible to resolve unambiguously such situations by looking only at the object, without further constraint propagation (otherwise the algorithm would require backtracking), because there always exists a discriminating attribute that directly specifies the type, in the form of a *name*, an *integer* or a *boolean*. For example, the `/Type` field of a page is `/Page`, whereas it is `/Pages` for an internal node.

**PDF types.** The PDF reference defines many types that we have to include in our algorithm. We organized these types into the following building blocks.

- **Basic types**: null, booleans, numbers, names, strings.

- **Enumerated types**: basic objects that have a small set of possible values. For example the `/BitsPerComponent` attribute of an image may only take certain values (1, 2, 4, 8 or 16).

- **Classes**: the majority of complex objects are dictionaries, that contain a set of attributes with predefined keys. For example, a page object contains – among others – the following attributes: `/Type`, `/Parent`, `/Contents`, etc. Some attributes are mandatory and others are optional.

  ```
  <<
      /Type /Page
      /Parent 437 0 R
      /Contents 415 0 R
      /MediaBox [0 0 612 792]
      /Resources 414 0 R
  ```

  ```
      /Annots [413 0 R]
  >>
  ```

- **Homogeneous arrays**: sequences of values of the same type. For example, an internal node in the page tree contains a list of children nodes.

  ```
  <<
      /Type /Pages
      /Kids [1 0 R 2 0 R 3 0 R]
      ...
  >>
  ```

- **Optional arrays**: in some contexts, when an array contains only one element, the array part is optional. For example, the `/Filter` attribute of a stream defines a sequence of chained compression filters, such as:

  ```
  [/ASCIIHexDecode /FlateDecode]
  ```

  But when only one filter is present, the following expressions are equivalent:

  ```
  /FlateDecode
  [/FlateDecode]
  ```

- **Tuples**: arrays of fixed size with a given type for each index. For example, a *destination* describes a position in the document, that can be targeted by internal hyperlinks, and is typically of the form `[page /XYZ x y zoom]`, to target a `page` at position `x`, `y` with a `zoom` factor.

- **Arrays of tuples**: to define an array of tuples, the PDF language often uses a 1-dimensional array, instead of an array of arrays. For instance, a name tree contains pairs of strings and objects, laid out in the following manner.

  ```
  << /Names [
      (section 1) 1 0 R
      (section 2) 2 0 R
      (section 3) 3 0 R
  ] ... >>
  ```

  In this example, `(section 1)` is mapped to object `1 0 R`, `(section 2)` to object `2 0 R`, etc.

- **Homogeneous dictionaries**: dictionaries that map arbitrary names to values of a given type. For instance, the *font resource dictionary* maps names to font objects. These names can be referred to by *content streams*.

  ```
  <<
      /F1 123 0 R
      /F2 456 0 R
  >>
  ```

  In this example, name `/F1` is bound to font object `123 0 R` and name `/F2` to object `456 0 R`.

**Complex dependencies.** Some constraints are difficult to express with our formalism. For example, classes have mandatory and optional attributes. Some mandatory attributes can in fact be inherited from ancestors in a tree structure and are not mandatory at object level.

More precisely, each page must declare its dimensions, but this property can be inherited in the page tree, because pages often have the same dimensions within a document. The dimension attribute is neither mandatory in a single page object nor in a single node object, but each page must have an ancestor with this attribute. We cannot check this kind of property at this stage because our algorithm works locally. However, we could declare the dimension attribute as optional and validate the constraint independently... or the specification could be amended to make the validation of this rule easier.

### 5.3. Higher-level properties

Our parser and type checker provide a basis to verify higher-level properties. This includes the consistency of the graph of objects and the validity of embedded elements (fonts, images, etc.). We already implemented some verifications on the graph and other checks can be integrated in a future work.

**Checking graph properties.** We have seen in section 4.1 that graph structures such as the *page tree* or the *outline hierarchy* raise problems when they are incorrect. To validate these structures, we check that they do not contain any cycle, and we ensure that doubly-linked lists are consistent.

**Checking features.** Once we have confidently identified the types of objects, it is possible to perform checks on specific features. For example, we could integrate tools that analyze JavaScript, such as those presented in section 3.3. We could also check that embedded fonts and images conform to their respective specifications.

## 6. Implementation and results

### 6.1. Implementation

**Implementation choices.** CARADOC was implemented using OCAML. Here are the advantages of this language, regarding our goal to write a robust parser.

- OCAML is a strongly-typed functional language, which allows the developer to catch more errors at compile-time, with relevant debugging information, instead of getting them later at runtime.
- The language is naturally expressive, which helps to write concise code: in particular, *sum types* allow to naturally define abstract syntax trees representing PDF objects. Similarly, *pattern matching* provides an elegant and efficient method to visit these structures.

- As the memory is handled automatically by the garbage collector, several classes of attacks (double frees, buffer overflows) are impossible[5].
- Pattern matching exhaustiveness check is a very helpful feature when dealing with complex structures or algorithms, since the compiler will remind you of unhandled cases.

For the lexical and syntax analysis part, we chose to use OCAMLLEX with MENHIR, respectively a lexer and a parser generators for OCAML, allowing us to simply use the grammar presented in section 5.1.

For all these reasons, we believe not only that our tool is a direct implementation of our language restriction rules, but also that our implementation choices lead to source code that can more easily be verified. It is interesting that other projects have recently used OCAML to achieve complex tasks: PARSIFAL [21], a binary parser generator, and NOT-QUITE-SO-BROKEN-TLS [17], a new TLS stack written by the Mirage project. In both cases, similar arguments have been argued in favor of OCAML as a safe and expressive language.

**Workflow.** We organized our implementation into the following workflow.

First, we implemented a strict parser that complies with our restricted grammar and extracts the objects from the file. We also implemented a relaxed parser that accepts more syntactic structures of PDF, in order to normalize a wide range of files into our restricted format (see Fig. 10).
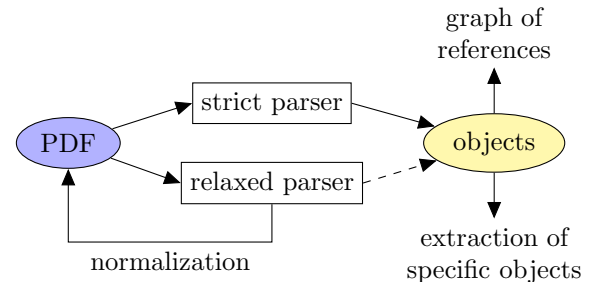


Figure 10. Objects extraction.

Once the objects are extracted, we can perform type and graph checking. Further checks could be added to complete the validation (see Fig. 11).

Apart from that, the user can collect information along the way: retrieve a specific object, extract the graph of all indirect references or get the types of objects.

### 6.2. Tests on real files

**Set of files.** In order to assess our tools with real-world files, we gathered a set of PDF files from the Internet. To find these files, we used a search engine, specifying that we wanted PDF files. Each query consisted of a single word

---

5. However, to be honest, this alone is far from perfect, since we may avoid arbitrary execution but not a fatal exception.
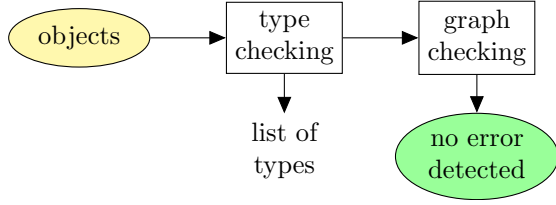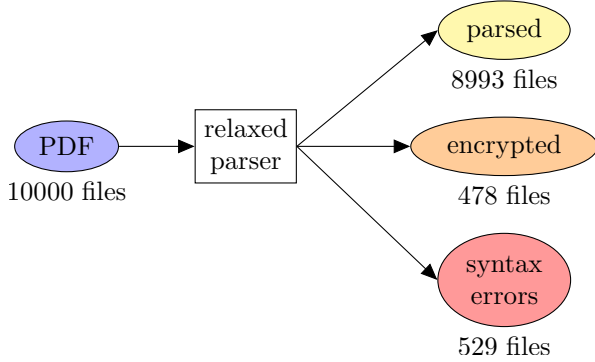
Figure 11. Validation process.



Figure 12. Set of real-world files.

extracted randomly from an English dictionary. In total, we gathered 10000 files.

This method may have some biases: correct files have more chances to be indexed by the search engine, documents are likely to be written in English because of the queries that we make and the search engine may prefer some kind of high-quality content. However, we observed that the resulting set comprised a great variety of features and errors at every level of the validation process.

Some of the files contained encrypted objects and others had syntax errors that were not recovered by our relaxed parser. In total, we could parse 8993 files (see Fig. 12).

All possible versions of PDF – from 1.0 to 1.7 – were present in the set (Fig. 13). The majority of the files contained between 50 and 300 objects (Fig. 14).

**Direct validation.** We first tested the validation chain directly. Only 1465 files were accepted by our strict syntactic

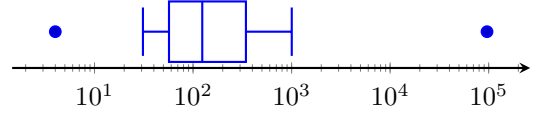| Version | Number of files |
|---------|-----------------|
| 1.0 | 0.1% |
| 1.1 | 0.5% |
| 1.2 | 4.1% |
| 1.3 | 19.7% |
| 1.4 | 32.2% |
| 1.5 | 23.6% |
| 1.6 | 15.6% |
| 1.7 | 4.2% |

Figure 13. PDF Version.



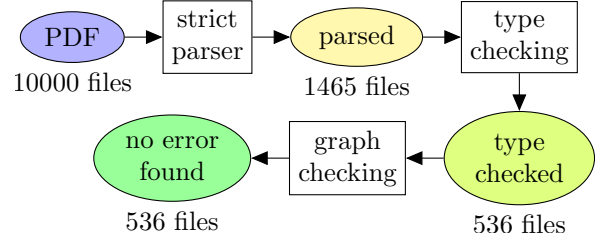Figure 14. Number of objects per file (8993 files).



Figure 15. Direct validation.

rules, of which 536 were successfully type-checked. No error was found in the graph structure for them (Fig. 15).

**Limitations of the strict parser.** In fact, the majority of the files made use of *incremental updates*, that were not allowed by our restrictions. Some files even contained several successive updates (Fig. 16).

Apart from that, many files used *object streams* and/or contained *free objects* (Fig. 17).

Finally, from the files that did not contain these forbidden structures, a significant number did not conform to our strict syntactic rules.

### 6.3. Normalization

**Relaxed parser.** In order to handle more files, we implemented a normalization tool that rewrites files into the restricted format when possible. This tool was based on our relaxed parser. Contrary to the strict parser – that processes

| Number of updates | Number of files |
|-------------------|-----------------|
| 0 | 36.2% |
| 1 | 43.1% |
| 2 | 18.3% |
| 3 | 1.3% |
| 4 | 0.4% |
| $\geq 5$ | 0.8% |

Figure 16. Number of incremental updates per file.

| Problematic structure | Number of files |
|-----------------------|-----------------|
| Incremental updates | 64% |
| Object streams | 35% |
| Free objects | 29% |
| Encryption | 5% |
| At least one of these structures | 76% |

Figure 17. Structures not allowed by the strict parser.

the file linearly from the beginning to the end – the relaxed parser uses the xref table(s) to obtain the positions of objects and extract them. Hence, it was able to decode files containing *incremental updates* and *object streams*. It did not recognize *linearized* files as such, but these files could be decoded by means of standard xref tables, because *linearization* only adds metadata to the classic structure.

Our normalization tool then removed all objects that were not accessible from the *trailer* and renumbered the objects in a continuous range starting from zero. In total, we could rewrite 8993 files out of 10000.

**Ad hoc extensions.** In practice, some files did not pass the relaxed parser, either because they contained encrypted objects or because of syntax errors with respect to the standard PDF specification.

For example, we found files that contained ill-formed cross-reference tables. In fact, the cross-reference table allows to define *in-use* objects but also *free* objects. This is useful to remove content by means of incremental updates. The invalid files that we found incorrectly declared *in-use* objects at offset zero instead of using the appropriate syntax for *free* objects. Hence, our first version of the relaxed parser could not parse them – since it did not recognize an object at offset zero. However, since this bug was common, we decided to add an ad hoc option to accept this mistake and be able to normalize the files in question.

We implemented similar options for common bugs that we noticed, but did not intent to correct every possible mistake. Thus, some files could not be normalized. The 8993 normalized files were obtained with these options.

**Consistency.** A required property of any normalization step is to be non-destructive: a normalized file must be equivalent to the original one from a semantic point of view. For this purpose, we checked manually – i.e. on a restricted set of files – that the normalized files could effectively be opened in PDF readers and that the graphical result was the same. Since our normalization simply rewrote the objects in a cleaner manner but did not modify the content of these objects, we have good confidence that the process is effectively non-destructive.

**Benefits.** Moreover, we found cases where the normalized file was better than the original. For example, PDF allows to integrate forms that the user can fill in and save into a new file. This new file often makes use of an *incremental update* to append the new content of the forms – and invalidate the previous content. However, we noticed that some forms filled-in with some reader could sometimes not be loaded by other readers. Yet, after normalization by CARADOC, these files were readable by all readers. Clearly, normalization made these files more portable.

This example also shows that *incremental updates* are not well supported by PDF readers in some cases, and that it made sense to disallow them in our restricted format.

## 6.4. Type checking

The whole PDF language contains a large number of types, presented in more than 700 pages in the specification. Hence, we did not intent to be feature-complete, but to integrate the most common types. This subset can easily be extended in the future.

**Choices of types.** To define the types to include first, we worked with simple PDF files produced by LaTeX, such as an article and a BEAMER presentation. We added types incrementally until these files were fully type-checked. Then, we also used the set of real-world files to add some widespread types that were not present in our LaTeX files.

In the end, our set comprises 165 types, including 108 classes. It contains the following types:

- overall structure of the document: the catalog, the page tree, name trees and number trees;
- graphical content of pages and resources: colors, images, fonts;
- interactive content: the outline hierarchy, simple annotations, common actions (such as destination to a URI), transitions (for slideshows);
- various metadata: viewer preferences (default view of a document), page labels (to number pages in a specific range).

We did not implement the following elements yet, but they could be integrated in a future work:

- advanced interactive content: JavaScript code, multimedia;
- the *logical structure* of a document (that allows to identify elements such as chapters, sections or figures).

**Results.** We present here the results of the type-checker on the normalized files.

Although our set of types was incomplete, it gave promising results on our data set. In the majority of the files, we could *infer* the types of more than 90% of the objects (Fig. 18).
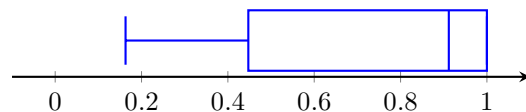


Figure 18. Proportion of objects of inferred type per file (7597 files).

In most cases, we could only partially check the file, which means that the types of some objects were not *inferred*, but no errors were found in the *checked* objects. This partial inference was possible because we used a placeholder `any` type for types that we did not implement yet. Consequently, objects with this `any` type were not further inspected – and objects referenced by them were not traversed. Similarly, we sometimes included only the most common attributes in some types, which means that

we could find unknown attributes – such as metadata – in the wild. Type checking failed on 1391 files, either because our type subset was too restrictive or because the files had indeed incorrect types (Fig. 19).
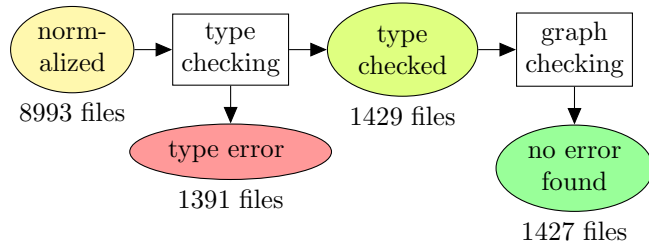


Figure 19. Validation after normalization.

**Discussion.** We found several reasons for type errors to occur in real-world files that are not malicious.

First, type constraints seemed to be too restrictive in some cases. For example, objects that represent fonts include a wide range of attributes, but there are 14 pre-defined fonts for which these attributes are standardized. To ensure compatibility between the two kinds of fonts, the specification states that the additional attributes shall either be all present or all absent. However, we found files where only some of these attributes were present. These files are not compliant, but this is probably a mistake rather than an intentional error, and PDF readers cope with such errors in practice.

Second, we found files with spelling mistakes. For example, some object contained the name `/Blackls1` (with a lowercase 'L') whereas the specification requires `/BlackIs1` (with an uppercase 'i'). In fact, the specification is itself distributed as a PDF file and such names are written in a sans-serif font, with makes it difficult to distinguish the two letters, hence the mistake. Another example was the misspelling `/XObjcect` instead of `/XObject`.

We hope that our tool can provide feedback to PDF producers by pointing out these mistakes, and thus improve them. Also, a new normalization stage could be developped to clean up these inconsistencies.

**Versions of the format.** Some type constraints of the specification are related to the PDF version that is used, because new features were added and others were deprecated over time. We did not perform these checks, because they add a lot of complexity to the analysis and current PDF readers usually implement the features without strictly checking that they are allowed in the announced version.

### 6.5. Graph checking

Of interest, 2 files contained errors at graph level. These errors were related to their outline structure.

In one case, all nodes of the outline tree referred to the root as their `/Parent` attribute, instead of their respective parents. In the other case, the linked-list was only singly linked, i.e. nodes specified a `/Next` link but the `/Prev` link counterpart was missing.

| Filter | Proportion of the files | Supported by CARADOC |
|---|---|---|
| /FlateDecode | 98.2% | yes |
| /DCTDecode | 49.5% | no |
| /CCITTFaxDecode | 14.2% | no |
| /ASCII85Decode | 5.4% | no |
| /LZWDecode | 3.5% | no |
| /JBIG2Decode | 2.7% | no |
| /JPXDecode | 2.6% | no |
| /ASCIIHexDecode | 1.0% | no |
| /RunLengthDecode | 0.1% | no |

Figure 20. Stream filters used in practice (8993 files).

### 7. Future work

Although our approach provides a reliable basis at the structural level, there remains a few limitations for a comprehensive validation of PDF files. The format is inherently complex, because it allows to include various forms of elaborated objects. Let alone JavaScript code, that by itself turns PDF into a Turing-complete format, one can include images, videos, fonts, etc. Some of them are essential to a document format and should be validated as a priority: vector graphics, fonts and images.

### 7.1. Streams

Streams can be encoded with a wide range of filters (Fig. 20). Currently, we do not check most of them because the inherent complexity of compression formats goes beyond the scope of our work.

Thanks to a library, we support the `/FlateDecode` filter, that corresponds to the ZLIB algorithm, because it is often used in *object streams*, that we needed to decode in the normalization process.

However, badly encoded streams with uncommon filters – such as `/JBIG2Decode` – have been used in practice by malware creators to further obfuscate malicious payloads [12], [31]. Thus, checking the proper encoding of streams is essential for the validation process of PDF files.

### 7.2. Content streams

The graphical content of each page is encoded as a sequence of commands wrapped inside a *content stream*. These commands encode geometrical primitives and follow a specific grammar that should be validated as well. Since our type checker already provides the list of objects that are *content streams*, a new validation module for them would fit well within our workflow.

### 7.3. Higher-level features

To fully validate PDF files, we also need to check the consistency of embedded images and fonts. Recent vulnera-

bilities have been uncovered in the parsing of fonts, notably on WINDOWS for which this is done in kernel space [16], [24]. PDF files with embedded fonts are another channel to deliver malicious content.

Likewise, existing analyses on JavaScript content could benefit from our reliable parser and type checker. In fact, JavaScript code can be embedded in many locations: in the catalog, in pages, in an embedded XML file, etc. Contrary to previous approaches that only target a blacklist of locations that may not be exhaustive, a type error will be reported in our framework if we forget to implement one of these types, which hinders the risks of evasion.

### 7.4. Prospects

Our proposition helps improving PDF-manipulating software at two levels.

First, at *generation* level, our formal grammar can be leveraged to ensure that produced documents can be unambiguously parsed. Generators could rely on it to increase the portability of their tools, because errors can be spotted in advance, directly at the development step.

Second, at *parsing* level, CARADOC provides a reliable parsing of documents that can serve many purposes, not restricted to the security goal. Since any unsupported feature can be reported as an error, tools that use it are safer and more reliable. Objects can also be extracted from a PDF file and provided to third-party tools for further investigation. This allows to validate a wider range of content without integrating them directly within CARADOC.

## 8. Related work

### 8.1. PDF validation

Some existing tools were written to validate PDF files. The JHOVE framework [14] is a Java application intended to validate conformance of files to various formats, among which JPEG, GIF or XML. Validation of PDF files is also supported through the PDF-HUL module [15]. Files are checked for well-formedness at a syntactic level and for validity of some criteria at document level, including some tree structures. However, the performed checks are not comprehensive and follow a blacklist approach: non-supported structures are considered valid by default. For example, we were able to create an ill-formed file that caused a crash on some readers but was nevertheless reported as "well-formed and valid" by PDF-HUL. On the contrary, we believe that our approach provides stronger guarantees, because at type checking level, we flag non-supported types as suspicious by default. Our syntax restrictions are also easier to verify.

Apart from that, many tools exist to validate conformance of files to the PDF/A standard. This subformat of PDF is intended for long-term archival of documents and requires additional properties, such as mandatory embedding of fonts inside the file. For example, the VERAPDF project[6]

6. http://verapdf.org/

is expected to provide an open-source PDF/A validator in a near future. However, the PDF/A standard mostly enforces properties at *feature* level and these validators do not really take the *syntax* nor the *structure* of the file into account.

### 8.2. PDF analysis

Other tools were written to inspect, modify and manipulate PDF files, such as ORIGAMI [28], PEEPDF [8], PDFTK [26], PDFMINER [32], PDFBOX [20] or PDF TOOLS [34]. They are useful to handle valid files, but they chose to implement custom parsing code and to attempt to cope with syntax errors. As a matter of fact, most of these parsers mix input data checks with processing logic that do not belong to the validation process, which is a well-known anti-pattern leading to security vulnerabilities [25]. In these so-called *shotgun parsers*, the data and parser computations are intricated and become very context-sensitive, which leads to a drastic increase of reachable states inside the program, including unwanted or dangerous ones.

As we showed earlier in this paper, such parsers could be tricked by targeted syntactic errors due to their internal discrepancies. Objects could be misinterpreted or worse, silently ignored, which is conducive to evading the validation step where object internals are checked. With limited guarantee at syntax level, these tools are therefore unsuitable for validation and analyses built on top of them are not reliable from a security point of view.

To overcome these limitations, adding more validation logic to their parsing code would make the analysis of all the computation paths untenable, with detrimental results due to the increased complexity. On the contrary, our approach offers a generic way to transform the grammar into a parser that can be used for validation of files but also for extraction purposes. Thus, CARADOC could be used as a building block for more complex analyses.

## 9. Conclusion

In this article, we have studied the low-level structure of PDF files, to analyze how the parsing step might go wrong when interpreting a document. We first crafted invalid or ambiguous files to test existing libraries and viewers, which led us to uncover several bugs – currently under review – in these implementations. Then, we proposed a set of rules that define a restriction of the original PDF specification, to reduce ambiguities at the structural level. Finally, we implemented CARADOC, a validation tool to check our rules in a real-world context.

The first obtained results are promising, since our tools could validate a significant part of our corpus. While we already identified areas for improvement, we also found several cases where rejection was the correct verdict. Moreover, we are convinced that providing a robust parser is the required first step to allow for higher-level analyses as those already existing.

Contrary to previous studies, that usually started with a parsed abstract syntax tree, we do not consider the parsing

step as a simple and straightforward process, but as a possible source of ambiguities and discrepancies among implementations (which usually accommodate with malformed files for compatibility's sake). For all these reasons, parsing a PDF in a reliable and generally acceptable way is nearly impossible. Our proposal essentially identifies possible improvements to fill in the gap between the current state of the standard and a simple and reproducible parsing process.

During our study, we found a lot of inconsistent or vague requirements, leading us to add restrictions; we also encountered requirements that were very hard to validate whereas an adaptation of the rules would be trivial to check. We thus believe that our work (our test cases, the formal grammar and rules, and our implementation) could be a relevant input for a specification update, be it the new PDF 2.0 standard under definition, or a new *restricted* PDF dialect, which could be called PDF/R, similarly to PDF/A for archiving purpose.

Finally, our approach can be generalized to other formats to propose tools and/or specification updates to improve the state of software security. This work naturally falls within the LangSec – language-theoretic security – philosophy, which prones to ban ad hoc input handling and instead use well-known and well-studied formal – yet simple – methods, for example decidable grammars or type-checking algorithms. We strongly believe that developing this state of mind is necessary to counter the growing complexity of our systems and help us secure them in the long term.

# References

[1] A. Albertini. Polyglottes binaires et implications. In *SSTIC*, 2013.

[2] A. Albertini. This PDF is a JPEG; or, this proof of concept is a picture of cats. *PoC or GTFO 0x03*, 2014.

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

[4] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases*, pages 387–402. Springer, 2013.

[5] A. Bogk and M. Schopl. The Pitfalls of Protocol Design: Attempting to Write a Formally Verified PDF Parser. In *Security and Privacy Workshops (SPW), 2014 IEEE*, pages 198–203. IEEE, 2014.

[6] G. Endignoux. Adobe Reader PSIRT tickets 3939, 3940, 3941, 3942.

[7] G. Endignoux. Poppler bug report 91414. https://bugs.freedesktop.org/show_bug.cgi?id=91414.

[8] J. M. Esparza. PeePDF. BlackHat Europe, 2012.

[9] E. Filiol, A. Blonce, and L. Frayssignes. PDF security analysis and malware threats. *Journal in computer virology*, 3(2):75–86, 2007.

[10] ISO. Document management—Portable document format—Part 1: PDF 1.7. ISO 32000–1:2008, International Organization for Standardization, Geneva, Switzerland, 2008. http://www.adobe.com/content/dam/Adobe/en/devnet/acrobat/pdfs/PDF32000_2008.pdf.

[11] S. Jana and V. Shmatikov. Abusing file processing in malware detectors for fun and profit. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 80–94. IEEE, 2012.

[12] D. Johnson. Antivirus Developers Dropped the Ball: PDF Is Not a Surprise. http://talkingpdf.org/antivirus-developers-dropped-the-ball/.

[13] A. D. Joseph, P. Laskov, F. Roli, J. D. Tygar, and B. Nelson. Machine Learning Methods for Computer Security. *Machine Learning Methods for Computer Security, Dagstuhl Manifestos*, 3(1):1–30, 2012.

[14] JSTOR and Harvard. JHOVE - JSTOR/Harvard Object Validation Environment. http://jhove.openpreservation.org/.

[15] JSTOR and Harvard. JHOVE - PDF-hul Module. http://jhove.openpreservation.org/pdf-hul.html.

[16] M. Jurczyk. Vulnerability in Microsoft Font Driver Could Allow Remote Code Execution. Microsoft Security Bulletin MS15-078.

[17] D. Kaloper-Meršinjak, H. Mehnert, A. Madhavapeddy, and P. Sewell. Not-Quite-So-Broken TLS: Lessons in Re-Engineering a Security Protocol Specification and Implementation. In *24th USENIX Security Symposium*, 2015.

[18] P. Laskov et al. Practical evasion of a learning-based classifier: A case study. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 197–211. IEEE, 2014.

[19] P. Laskov and N. Šrndić. Static detection of malicious JavaScript-bearing PDF documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, pages 373–382. ACM, 2011.

[20] A. Lehmkühler. Apache PDFBox - Working with pdfs for Dummies. *ApacheCon*, 2010.

[21] O. Levillain. Parsifal: A Pragmatic Solution to the Binary Parsing Problems. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 191–197, 2014.

[22] D. Maiorca, I. Corona, and G. Giacinto. Looking at the bag is not enough to find the bomb: an evasion of structural methods for malicious PDF files detection. In *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, pages 119–130. ACM, 2013.

[23] D. Maiorca, G. Giacinto, and I. Corona. A pattern recognition system for malicious PDF files detection. In *Machine Learning and Data Mining in Pattern Recognition*, pages 510–524. Springer, 2012.

[24] M. Mimoso. Of TrueType Font Vulnerabilities and the Windows Kernel. https://threatpost.com/of-truetype-font-vulnerabilities-and-the-windows-kernel/101263.

[25] M. L. Patterson, S. Bratus, and D. T. Hirsch. Shotgun parsers in the cross-hairs. In *BruCON 2012*, 2012.

[26] PDF Labs. PDFtk - the PDF toolkit. https://www.pdflabs.com/tools/pdftk-the-pdf-toolkit/.

[27] W. M. Petullo, W. Fei, J. A. Solworth, and P. Gavlin. Ethos' Deeply Integrated Distributed Types. In *35. IEEE Security and Privacy Workshops, SPW 2014, San Jose, CA, USA*, pages 167–180, 2014.

[28] F. Raynal, G. Delugré, and D. Aumaitre. Malicious origami in PDF. *Journal in computer virology*, 6(4):289–315, 2010.

[29] J. Rutkowska. Converting untrusted PDFs into trusted ones: The Qubes Way. http://theinvisiblethings.blogspot.fr/2013/02/converting-untrusted-pdfs-into-trusted.html, 2013.

[30] C. L. Schade. FCScan: A new lightweight and effective approach for detecting malicious content in electronic documents. University of Twente, 2013.

[31] J. Sejtko. Another nasty trick in malicious PDF. https://blog.avast.com/2011/04/22/another-nasty-trick-in-malicious-pdf/.

[32] Y. Shinyama. PDFMiner. https://euske.github.io/pdfminer/.

[33] C. Smutz and A. Stavrou. Malicious PDF detection using metadata and structural features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, pages 239–248. ACM, 2012.

[34] D. Stevens. PDF tools. http://blog.didierstevens.com/programs/pdf-tools/.

[35] Z. Tzermias, G. Sykiotakis, M. Polychronakis, and E. P. Markatos. Combining static and dynamic analysis for the detection of malicious documents. In *Proceedings of the Fourth European Workshop on System Security*, page 4. ACM, 2011.

[36] J. Wolf. OMG WTF PDF. In *27th Chaos Communication Congress (27C3)*, 2010.