A Format-Aware Reducer for Scriptable Rewriting of PDF Files

Prashant Anantharaman
Dartmouth College
Hanover, NH

Steven Cheung
SRI International
CA, USA

Nicholas Boorman
SRI International
CA, USA

Michael E. Locasto
Narf Industries
NJ, USA

2022 IEEE Security and Privacy Workshops (SPW) | 978-1-6654-9643-8/22/$31.00 ©2022 IEEE | DOI: 10.1109/SPW54247.2022.9833885

***Abstract*—Sanitizing untrusted input is a significant unsolved problem in defensive cybersecurity and input handling. Even if we assume that a safe, provably correct parser exists to validate the input syntax, processing logic may still require the application of certain transformations of the parser output. For example, parsers traditionally store the parsed objects in a generic tree structure; hence the processing logic needed to modify this structure can be significant. Also, popular parsing tools do not include the functionality to serialize (or unparse) the internal structure back to bytes.**

This paper argues for the need for a format-aware tool to modify structured files. In particular, we propose adding a reducer to the Parsley PDF checker. The Parsley Reducer—a tool to apply transformations on input dynamically—would allow developers to design and implement rules to transform PDF files. Next, we describe a set of Parsley normalization tools to be used with the Reducer API and showcase their capabilities using several case studies. Finally, we evaluate our normalization approach to demonstrate that (1) the developer effort to design our reducer rules is minimal, (2) tools extract more text from transformed files than the original files, and (3) other popular PDF transformation tools do not apply the corrections we demonstrate.

***Index Terms*—LangSec; Reducer; PDF; Parsley; Serializer; Safe PDF rewriting;**

1. Introduction

There is a need to have an organized way of applying fine-grained rewriting rules to *parsed* input, particularly for data formats. For example, software engineers that deal with a complex file format like PDF need a toolchain to enable them to rewrite an arbitrary, complex PDF file to a notional "PDF/Safe" dialect by removing or rewriting slight malformations or by substituting risky instances of particular format features with an equivalent representation that carries less risk. Engineers who deal in data languages often lack a principled framework for applying transformations to instances of that data in a safe and systematic way [10], [18].

Current data rewriting systems in file and network security are usually based on regular expression search-and-replace, such as URL rewriting schemes when they occur in email. This technique "works" because typical URLs do not have deep, nested structure.[1] This grep-like approach will not work for complex structured input. Moreover, the post-parsed input is likely contained in a tree-like or similar data structure, thereby losing the ease of applying file and line-oriented Unix-style string search-and-replace tools. Such tools and tracers lack format awareness: they do not "see" the meta-language structure on top of the symbols built to scan.

One way of visualizing this problem is to apply a regular expression-aware trigger-action programming pattern. Xpath [12], M4 [22], and Gremlin [33] are standard tools used to achieve this task. However, we would need to build tools to operate on complex graph or tree structures instead of text or XML data in the more generic case.

This paper argues for the need for format-aware tools to transform, sanitize, and serialize data. There are several reasons to design such tools. First, not all data formats are similar. For example, the transformations we must apply to a PDF file differ significantly from DNS network protocol message transformations. File formats often contain offsets—data spread throughout the file, and the locations and size of this data are maintained in a table. Designing an intermediate representation common across various data formats is also challenging.

Second, we need tools that can generate files from an intermediate representation. This operation is called unparsing or serializing [19]. However, with the complex structures in data formats, programmers need to design their serialization tools for a particular format to convert the internal representation to bytes.

Finally, the PDF specification has evolved a lot over the last three decades, resulting in several ad hoc implementations that create PDF files that violate the specifications in subtle ways. We investigated commonly occurring errors in PDF files and found that a substantial number of errors require sophisticated methods to fix these errors. For example, in a dataset of 1 million PDF files, we found that over 3000 of them contained page tree node malformations of some sort—meaning keys mandatory in the page tree nodes

1. Although they can be surprisingly complex, as evidenced in M. Zawelski's Browser Security Handbook, due to the various legal representations of IPv4 and IPv6 addresses along with the permitted encoding schemes for GET and POST parameters.

DOI 10.1109/SPW54247.2022.00019

were missing. As a result, these otherwise benign files do not open on several PDF readers.

Additionally, Muller et al. [26] demonstrated that PDF files contain metadata, personally identifiable information, and redacted data. For example, they found that over 58% of PDF files in their sample included author information. Furthermore, PDF tools also include the creation time and the name and version of the tool used to create the file. An adversary can use this information about the victim's PDF tools to craft exploits.

Although the most common application for data transformation tools is fixing minor malformations in data, there could be other use cases. For example, we can transform valid files to violate format specifications. We can then use these files to test parsers for these formats. We can also use a parser and serializer to redact data transiting network filters. We can achieve this by applying transformations to the parsed data structures and then serializing the parsed structure back to a network packet.

To this end, we present the Parsley normalization tools that address the above challenges using several approaches. First, we introduce the Parsley Intermediate Representation (Parsley IR), a representation to capture the syntax of PDF objects in use. Developers can use this IR to debug PDF files or use it with other normalization tools that are a part of the Parsley toolchain.

Second, we develop the Parsley Reducer API to allow developers to specify various transforms on the PDF structure. A *reducer* is a tool that can dynamically execute transformation functions over data. These transformation functions are defined as a set of reducer rules. Then, a reducer applies these rules to transform data. This paper proposes a reducer API targeted at the PDF format. The API allows a developer to specify selection, filtering, and replacement rules.

Our Reducer API provides a customizable, scriptable rewriting system for the PDF format. This design pattern is similar to the pattern used by Pin [24] and Valgrind [27] and dyninst and DynamoRio [9] for x86 code. These tools use this pattern for other purposes, such as performance profiling, static and dynamic memory and cache analysis, and bug finding. They dynamically inject x86 code to monitor the program's performance. In contrast, we invoke our Reducer API to apply transformations to the PDF structure.

Third, we build a serializer to convert Parsley IR to a transformed PDF file. *Serialization* is the process where an internal representation of data is converted to a set of bytes that can be stored on disk or sent over the network [32], [40]. In the case of a file format, it converts an internal representation to a file format. For example, we can convert a JSON representation of a PDF file to a PDF file. Similarly, in the case of network protocols, a machine beginning to transmit a message will create a structure with all the fields locally. Subsequently, these fields are converted to bytes in a sequence predefined by the serializer. Our serializer for the PDF format does not preserve the object order but retains all the in-use objects while computing the correct offsets and generating syntactically valid PDFs. We automatically remove objects not used.

Our paper makes the following contributions:

- We created a radically new capability we call the Parsley Reducer API, a novel tool to allow developers to programmatically redact or modify portions of PDF files. This concept uses a design pattern similar to Pin and DynamoRio, allowing scriptable dynamic data operations on PDF files. For example, this new capability provides something akin to a graceful dynamic exception handling mechanism that is external to the codebase it is protecting, similar to seminal work in matching exploit signatures in network stacks [38].
- Our research over a large corpus (1 million files) of real-world documents, led to the discovery and characterization of a number of malforms present in PDF files that are amenable to safe rewriting. We use these files to create a set of case studies that successfully demonstrates that this tooling repairs malforms and other conditions of concern in real PDF documents.
- We conducted experiments that explore several dimensions of comparison between existing PDF transformation tools and our Reducer. We discover that these existing PDF (normalization) transformation tools lack any ability to dynamically script or react to malforms present in their input and are unsound (i.e., they introduce errors during their rewriting).
- We provide conceptual contribution: there are many divergent unlabeled dialects of PDF. Our work shows and provides tooling so that one can begin to merge unlabeled dialects into labeled, normalized representations, where the overall organization is provided by the set of transformations defined and executed by the Reducer. In other words, we can begin to formulate dialects that are labeled by the set of transformations applied and/or malforms rectified. The benefit of this is to reduce poor outcomes like parser differentials and other format confusion.

1.1. Definitions

The Parsley normalization tools are a part of the larger Parsley system. The Parsley system first validates PDF files to ensure they are syntactically valid. As a result, we can successfully serialize well-formed files or those that we can transform into well-formed files using our reducer.

Definition 1.1 (Well-formed PDF file)**.** A PDF file is well-formed if the Parsley syntax checker decides that the overall syntax of the file is valid as per the PDF 2.0 specification.

Since we transform a PDF file, we want to ensure that the text in the original PDF is fully preserved. In addition, since we have fixed particular objects by applying transformations, pages the text extraction tools could not previously

access will now be accessible. Hence, we hypothesize that a safe normalization tool must produce the same amount of text or more text than the original file. Additionally, the transformed file must also be syntactically valid.

Definition 1.2 (Safe Normalization)**.** The original PDF file $P1$ is transformed to produce a new PDF file $P2$. This transformation is safe if:

- $P1$ and $P2$ are well-formed PDF files, and
- The text extraction output of $P2$ is identical or a superset of the text extraction output of $P1$ across multiple tools.

This paper describes six case studies we constructed to demonstrate the utility of a format-aware normalization tool for the PDF format. The normalization tool uses a reducer and serializer in conjunction to produce transformed PDF files.

Claim 1. The Parsley reducer and serializer produce safe normalizations.

We back this claim up by evaluating our normalization approach against two datasets, each over 6000 PDF files. Furthermore, the reducer rules we evaluated were designed to fix specific malformations commonly seen in PDF files but not commonly fixed by PDF rewriting tools.

Organization. The rest of this paper is organized as follows. Section 2 provides a primer on the PDF format and discusses a Recognizer we use to evaluate our normalization tool. Next, Section 3 discusses the design of the Parsley Normalization tool—detailing the implementation of the Reducer API and Serializer. Section 4 describes various errors and useful transformations we identified in prior work using the Parsley PDF checker and the Recognizer. We evaluate the Parsley normalization tools in Section 5. Furthermore, we discuss other conceptual contributions in Section 6, and other closely related work in Section 7. Finally, Section 8 concludes the paper.

2. Background

2.1. PDF Primer

PDF files, in their simplified form, comprise five basic structures [31].

- **Header**: this specifies the PDF version this file follows.
- **Objects**: a list of objects each conforming to a predefined PDF type.
- **Cross-reference table**: also known as *xref* table, this table contains a list of absolute locations of the objects in the PDF file.
- **Trailer**: this dictionary stores references to the *Root* object and *info* dictionary. The trailer also contains other metadata fields such as the size of the *xref* table.
- **End of file marker**: this marker shows where the *xref* table is stored.

Each object in a PDF file are identified by an *object* number and a *generation* number. For example, let us consider the following object.

```
13 0 obj
Object contents
..
endobj
```

In the above example, the object takes the object number 13 and generation number 0. This object can be referenced from other objects or the trailer via the reference `13 0 R`.

Dictionaries. PDF dictionaries are surrounded by << and >> characters. The keys in these dictionaries must be of the *name* type, whereas the value portion can hold any predefined PDF type. For example, `<</Foo1 (bar) /Foo2 13 0 R>>` is a dictionary with two keys, `Foo1` and `Foo2`, where the corresponding values are of types string and reference respectively. The ISO 32000-2:2020 PDF specification specifies several rules on various dictionaries.

Arrays. These are predefined objects that are surrounded by brackets (`[` and `]`) and contain multiple elements separated by spaces. Arrays can be homogeneous or heterogeneous and can be fixed or variable length. For example, `[(foo) /bar 13 0 R]` is a heterogeneous array of size three where the first object is a string, the second object is a name object, and the third object is a reference.

Streams. These predefined objects contain a dictionary and a compressed stream. The adjoined dictionary holds keys that specify the compression algorithm used and the length of the stream. In most cases, stream objects are *indirectly* referenced, i.e., they are entirely stored in a different object and not embedded in another object. The code snippet below shows an example of a stream object.

```
13 0 obj
<<
stream dictionary objects
>>
stream

endstream
endobbj
```

2.2. Running Example

In the rest of the paper, especially Section 4, we will use the following running example to discuss transformations to PDF files. This example includes two PDF objects that are surrounded by the `obj` and `endobj` tags. The first object in the snippet takes ID 1, whereas the second one takes ID 0. Both the objects have the generation number 0.

Catalog. In the above example, object 1 is a Catalog dictionary. Dictionaries, in PDFs, are surrounded by << and >> symbols. They hold key-value pairs that are separated by whitespaces. In this example, the keys are Names, Outlines, Lang, Type, PageLayout, and PageMode. There are

```
1 0 obj
<</Names <</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang (en-US)
/Type /Catalog
/PageLayout /OneColumn
/PageMode /UseOutlines>>
endobj

2 0 obj
<</Count 13
/Kids [6 0 R 25 0 R 33 0 R
35 0 R 38 0 R 45 0 R 49 0 R
57 0 R 60 0 R 63 0 R 66 0 R
68 0 R 70 0 R]
/Type /Pages>>
endobj
```

several constraints on a Catalog dictionary to ensure cross-compatibility: a file rendered on various PDF readers or text extraction tools must produce nearly identical results.

The Type and Pages keys are the only mandatory keys in a Catalog. Additionally, each key has a type associated with it. For example, the Lang key must have a corresponding string, and the PageLayout and PageMode keys must have a corresponding Name object chosen from a list of options. The Names key has a corresponding dictionary object in this example. Whereas the Outlines and Pages keys must always include indirect references to a dictionary that is stored in another object. The Outlines key refers to object number 5, while the Pages key refers to object number 2.

Page Tree Node. Similarly, Page Tree Nodes also include required and optional keys. The tree's root node must not include a Parent key but include Kids, Type, and Count keys. If any of these keys are absent, the Page Tree Node is invalid, and it can get challenging to traverse the tree. Other non-root nodes in the tree must include a Parent key.

Like in the case of the Catalog dictionary, the specification assigns types for these keys. For example, the Count key must have an associated positive integer, and the Kids key has an array of references to other Page Tree nodes or pages.

2.3. Meriadoc Recognizer

This paper relies on the Meriadoc Recognizer to test a PDF file against a set of PDF parsers [36]. The recognizer instruments PDF parsers such as Mupdf, Mutool [5], Qpdf [8], PDFMiner [34], and Poppler [29], to generate meaningful errors that help debug malformations.

It applies techniques such as parser watchpoints—monitoring parsers in runtime to find out which piece of code leads to errors. These parser watchpoints are format-aware tracers—explicitly designed to handle PDF files. These format-aware tools use a higher-level representation of the parsing events. With this representation, we can provide meaningful, explainable errors as output.

There is a strong coupling between the Meriadoc Recognizer and the Parsley Reducer, as we discuss in Section 3.4. The Recognizer provides feedback to pinpoint where other PDF parsers reject a file Parsley generates. With this insight on the error types, we revise future versions of our serializer to avoid buggy files. We also used this Recognizer to compare a file transformed by other PDF repairing or cleaning tools (such as Caradoc and Mutool clean) in Section 5.4.

3. Parsley Normalization Tools

There are two possible architectures we envision for the normalization tool. First, we can use the reducer as an additional pass to the PDF checker. This way, before files are type-checked, we fix the objects based on the set of reducer rules. Second, the reducer can be used in conjunction with the serializer. In this approach, we focus on making fixes in the intermediate representation and provide a transformed file.

3.1. Background

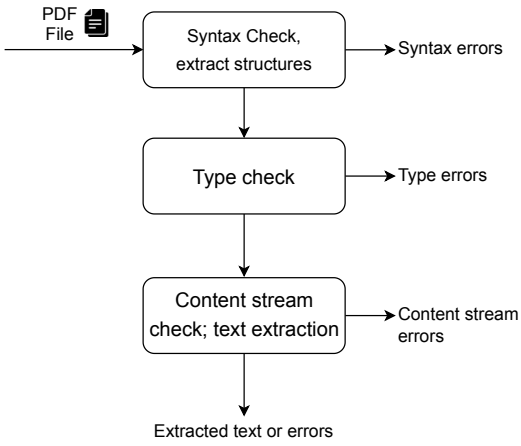

Figure 1. We show the steps followed by the Parsley PDF parser to validate PDF files.

The Parsley PDF parser applies several checks to ensure that a PDF file is well-formed. We built the individual components of the parser to follow the PDF 2.0 specification. Figure 1 shows the steps followed by our PDF parser.

First, we syntactically validate all objects in a PDF file by following the objects from the xref table. This step ensures that the dictionaries, streams, arrays, and other predefined-typed objects are well-formed. Next, the syntax checker extracts an intermediate representation that provides the following structures.

- The object numbers and the corresponding objects in use
- The root (Catalog) and info dictionaries

- PDF file version (extracted from the PDF header and the Catalog dictionary)

In the second step, we type-check the various objects starting with the Catalog dictionary and Info dictionary. We define specifications for each dictionary, array, and stream object to apply the correct types. For example, in arrays, we define the array size and the object type at each location in the array. Similarly, for dictionaries and stream objects, we define the keys that must be present and their corresponding types. Then, iteratively, we type-check all the objects referenced in these dictionaries based on type specifications. Finally, we discuss several reducer rules in the case study based on the errors we identified using our type checker.

In the third step in the Parsley PDF checker, explore content streams—descriptions of the page's contents. These content streams include text, shapes and drawings, and images. We validate content streams to ensure that the text and the metadata are well-formed. We also provide the functionality to extract this text.

In summary, the Parsley PDF checker performs three levels of validation to ensure that files are well-formed. Our second and third passes generate a binary output by checking the internal representation. Therefore, our normalization tools operate on the internal representation generated after the first pass. In definition 1.1, we consider a file well-formed if it passes the first step of validation in Figure 1.

This paper focuses primarily on the Parsley normalization tools that we use in conjunction with the rest of the Parsley PDF checker. Figure 2 demonstrates the overall architecture of the Parsley normalization tools. We designed the Parsley normalization tools (the reducer and serializer used in conjunction) with several goals in mind:

- **Usability: Provide an API to support scriptable editing**
  There are several reasons to edit PDF files. We want to design an API that provides flexibility to identify objects of interest and edit them.
- **Composable Design: No shotgun serializers**
  We must not place the burden of serializing the internal representation on the developer. We envision developers would use our serializer as a black box. Additionally, we also want to avoid the anti-pattern of a shotgun serializer—when a code performs validation or transformations while also serializing portions of the internal representation.
- **Soundness: Produce syntactically valid PDF files**
  Our serializer must produce syntactically valid PDFs that pass the first syntax check the Parsley system makes. However, these files may not pass the other two validation checks we place since PDF files may already contain type or content stream malformations in them. Additionally, developers can also introduce malformations using the reducer API—such as removing mandatory keys from dictionaries or deleting objects.

3.2. Parsley Reducer

The Parsley PDF parser and our normalization tools are implemented in Rust and require the compiler version 1.55.0 or higher.

We require that the developer specify a selector hash map, use one of the filtering functions we have built-in as a part of the reducer API, and then describe the type we must replace the filtered keys with.

3.2.1. Parsley IR. The Parsley Intermediate Representation (IR) is made available as an output of the Parsley syntax checker. This IR comprises of four items: (1) a hash map containing a mapping of PDF object IDs and their corresponding type-annotated objects, (2) object ID of the root object, (3) object ID of the Info dictionary, and (4) version number of the PDF file extracted from the header and the version string in the Catalog dictionary.

This IR does not store information about the exact locations of the various objects in the original file. Since PDF objects can be updated, traditionally these older, unused objects are still stored in a PDF file. However, our IR does not represent these unused objects—only storing the used objects.

```
(1,0): {
 Names: PDFType:Dict(map: {
        Dests: PDFType:Ref((4,0))
    }),
 Outlines: PDFType:Ref((5,0)),
 Pages: PDFType:Ref((2,0)),
 Lang: PDFType:String(val: "en-US"),
 Type: PDFType:Name(Catalog),
 PageLayout: PDFType:Name(OneColumn),
 PageMode: PDFType:Name(UseOutlines),
}

(2,0): {
 Count: PDFType:Integer(13),
 Kids: [PDFType:Ref((6,0)),
    PDFType:Ref((25,0)),...],
 Type: PDFType:Name(Pages)
}
```

Code Snippet 1. Parsley IR example of our running example

Code snippet 1 demonstrates the Parsley IR syntax for our running example. Essentially, it is a hash map of PDF object IDs and generation numbers and the object contents.

3.2.2. Selector. The selectors we use as input to the reducer take in a hash map with the list of keys and the corresponding values for selecting an object. For example, if we need to apply transformations to the Page Tree Node objects where the objects have the type key "Pages," we create a hash map with the key "Type" and value "Pages."

The Selector traverses the Parsley IR hash map to find the appropriate objects. Occasionally, this could also result

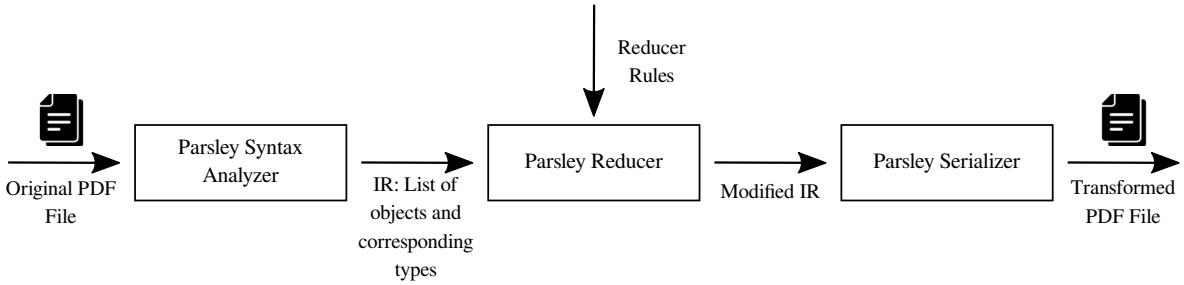

Figure 2. Changing the architecture of Parsley to support PDF rule-based transformations.

in traversing the dictionary structure of a certain object. For example, a selector rule to find all the Page Tree Nodes would need to find the Type key of every object to find if it is a Page Tree Node. The selected objects are then passed on to the filtering rules.

3.2.3. Filtering rules. Once the intermediate representations of selected objects are available, we can find specific structures of interest. Our filtering mechanism can extract specific objects based on dictionary keys or array indices. For example, we may want to filter out objects missing a particular key or inspect the type of the value corresponding to a key. Our reducer API supports several built-in functions to find objects within an outer object.

We list a few of the filtering rules we provide as a part of the reducer API.

- **Delete a key from a dictionary:** This rule is useful when some data needs to be redacted: such as personally identifiable data in the file metadata.
- **Select particular keys from dictionary or array**: This rule is useful when we need to replace specific keys or array locations. The transformed type is defined separately in the transformer.
- **Select version string**: The version string is specified in the file header and the Catalog dictionary. Hence, a modification to the version string needs to reflect in both locations. Hence, we provide a separate filtering rule and a corresponding transformer function to aid in this transformation.
- **Select keys in the Catalog/Info dictionary**: These dictionaries are special and are the root nodes. Hence, we provide a separate way to get access to keys in these nodes instead of traversing through the intermediate representation.

3.2.4. Transformer. We provide new values to the filtered keys or array locations using the transformer functions. The function follows the syntax where we provide a filtered object O and the object to replace O, O'. Then, the transformer applies these rules to each filtered object individually. Finally, we return the modified IR with the correct transformations applied.

3.2.5. Finding well-formed PDFs. Parsley IR is produced by the first step of the Parsley PDF parser—the syntax checker. Hence, to ensure that our Reducer rules can be applied, we must ensure that the file is syntactically well-formed. To find well-formed PDF files in our dataset, we separated the first pass of the PDF parser and ran it over all files in the dataset. Therefore, we only operate on this subset of well-formed files from our datasets in the rest of this paper.

3.3. Parsley Serializer

We built a serializer for the PDF format, starting with the Parsley IR. The files we generate must be syntactically valid PDFs to comply with the soundness property we set for our normalization tools. [2] We implemented our serializer in less than 200 lines of code in Rust.

We follow a recursive structure to serialize each PDF object. First, we traverse the Parsley IR, which contains a hash map of all object IDs to their corresponding objects. Then, we identify the type of each of these objects and then call a recursive function to serialize the internal objects within the object. For example, dictionaries can be nested. Dictionaries may also contain other arrays and stream objects.

Name objects in PDF can contain special characters. To handle these special characters, we included an additional check to ensure that we escaped these special characters. Similarly, String objects may also contain special characters that are escaped using a certain syntax. We escaped all characters in PDF strings to avoid further confusion. However, this approach makes debugging hard, and the modified files significantly diverge from the original files.

For Stream objects, we kept the dictionary and the compressed portions intact—not redacting or decompressing any objects to continue to preserve space. Finally, we remove all comments from the file by not serializing any comment objects. We do this because comments, metadata in the info dictionary, and unused objects collectively contain much unredacted information.

In summary, our serializer produces files compliant with the PDF 2.0 syntax. Furthermore, we recursively implemented the Parsley serializer since PDF objects contain complex types such as streams, dictionaries, and arrays

2. PDF parsers may still reject our modified PDF files for type malformations originally present or introduced as a part of our reducer rules. We only ensure that the syntax is correct.

containing other complex or basic types such as boolean, strings, numbers, and name types. We evaluate our serializer in Section 5 to ensure that the files we generate are syntactically valid by running our generated files against a host of PDF parsers.

3.4. Using Meriadoc Recognizer as a feedback loop

We found several bugs in our serialization approach after running our generated files against the set of parsers in the Meriadoc recognizer toolset. We list some of the findings and then discuss how we fixed these issues.

Incorrect understanding of the Size field in the cross-reference table. The specification says this when describing the Size field in the PDF file trailer dictionary, "this value shall be 1 greater than the highest object number defined in the file." Unfortunately, in the first version of our serializer, we wrongly computed these values as the total number of entries in the Xref table.

Qpdf and Mutool parsers that are a part of the Meriadoc recognizer spotted this malformation and provided warnings specifying that the file may be malformed. We subsequently fixed our serializer to generate these values correctly.

Special characters in the Name and String objects need to be escaped. The Name type in PDF contains strings that start with a slash (`/`). These name types are extensively used in dictionary keys. To use characters that are not alphanumeric, we can represent these values using hexadecimal. For example, the ASCII "+" character would be denoted as the hexadecimal string "`#2B`." Similarly, names can also include whitespace characters escaped using this syntax. The specification uses the following example: `Lime#20Green` must be interpreted as `Lime Green`. Our serializer did not implement this hexadecimal encoding for these special characters producing the string `/Lime Green` in PDF files. This led to several parser errors since the string `Green` here does not conform to any PDF type.

Likewise, PDF string objects use octal or hexadecimal encoding to store special characters. Any parenthesis used in a string must be balanced and require no special treatment. However, any special character must be escaped using an octal syntax `\ddd`—where `ddd` is the octal character code.

In an earlier implementation of the serializer, we wrote the internal string representation to bytes in the PDF objects. Unfortunately, similar to the case with name objects, special characters in strings were serialized as is. This led to several errors due to the incorrect handling of numerous special characters in these PDF objects. We have subsequently patched this issue by representing all string characters in octal encoding.

4. Case Study

Given the age of the PDF formats, several PDF creation tools have deviated from the specification in various ways. These deviations expose parser differentials in PDF readers. In this section, we study several PDF specification violations and data redaction cases and how we added reducer rules to handle these patterns and anti-patterns.

4.1. Finding type check errors in PDF datasets

Table 1. PDF TYPE CHECK ERRORS IN PDFS

PDF Errors	Occurrences in 1 million files
Incorrect Lang key	1k
Page Tree Node error	20k
Incorrect PageMode keys	500
Null references	5k

We ran the Parsley PDF type checker—closely designed for the PDF version 2.0 on 1 million PDF files collected via CommonCrawl and GovDocs digital corpora. We logged errors we encountered on files in this dataset and clustered them based on the type of error. Table 1 provides a summary of all the errors we found and their frequency in our dataset.

As a result of our study, we've designed case studies C1-4 to serve as a way to correct these commonly seen errors in PDFs. Subsequently, C5, C6, and A1 are other use cases such as redacting data or generating files that contain errors for better testing.

4.2. C1: Incorrect `Lang` key syntax

The language key in the Catalog dictionary aids text extraction. Some PDF creation tools use incorrect syntax to describe the language. For example, several files use the following Catalog dictionary syntax.

```
<</Names <</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang /en
/Type /Catalog
/PageLayout /OneColumn
/PageMode /UseOutlines>>
```

This dictionary differs from the running example in only the `/Lang` key. The value `/en` uses the name type in PDF. However, the specification notes that this field must be a string—not specifying what readers must do if this field is incorrect.

The specification does not offer a default value for this key. Instead, it specifies that if this key is absent, the language is considered to be unknown. Therefore, we remove this key from the Catalog dictionary in an attempt to remove this common anti-pattern from PDF files.

Selector	Filtering rule	Transformer
Catalog	Delete: `Lang`	-

The above table shows our reducer rules to remove the `Lang` key since the specification does not provide any default values for this key. Therefore, we specify that the filtering rule we apply is a delete operation and specify the key that must be deleted.

4.3. C2: Missing keys in Page tree nodes

Page tree nodes in PDF files must contain the `Kids`, `Type`, and `Count` keys. However, we found several PDF files in the wild with some of these keys missing. When PDF viewers attempt to walk the page trees in documents with these malformations, they encounter the page tree nodes with either the `Count` or `Kids` key missing—and often fail to render the PDF file. The count key holds an integer value. The kids key stores an array of references to other page tree nodes or pages. For example, following is an example of a malformed Page Tree node—differing in some ways from our running example.

```
<<
/Kids [6 0 R 25 0 R 33 0 R
35 0 R 38 0 R 45 0 R 49 0 R
57 0 R 60 0 R 63 0 R 66 0 R
68 0 R 70 0 R]
/Type /Pages>>
>>
```

In the above example, the Count key is missing. Other variations of this error could be a missing Kids key or both Kids and Count keys missing.

We transform these malformed page tree node dictionaries using our reducer syntax. We select dictionary objects where the `Type` key is set to `Pages`. Then, we check if the mandatory keys are present. We consider three cases:

- Both the keys are missing: we create an empty array for the kids key and set the count key to 0.
- Only the count key is missing: we count the number of pages in the kids array. We then set the correct count value. The count value may also be incorrect in the previous version of the file.
- Only the kids key is missing: we create an entry for the kids key in the dictionary. We then reset the count key to 0.

Selector	**Filtering rule**	**Transformer**
`Type=Pages`	Select if `Kids` missing	Set `Kids` to `[]` and `Count` to 0
`Type=Pages`	Select if `Count` missing	Set `Count` to `Kids.length()`

The above reducer rules account for all three cases we listed. The first rule we list accounts for both cases—irrespective of whether the Count value is present or not.

4.4. C3: Incorrect `PageMode` keys

The Catalog dictionary also contains the `PageMode` key. The PDF 2.0 specification provides a list of allowed values for this key. This field can only take the following values: UseNone, UseOutlines, UseThumbs, FullScreen, UseOC, and UseAttachments. Additionally, the specification also specifies the default value as UseNone.

```
<</Names <</Dests 4 0 R>>
/Outlines 5 0 R /Pages 2 0 R
/Lang (en-US)
/Type /Catalog
/PageLayout /OneColumn
/PageMode /None>>
```

We found hundreds of PDF files with incorrect page mode values not in this list. The code snippet above demonstrates such a value. So we wrote a reducer rule to rewrite the Catalog dictionary with the page mode value reverted to the default value. The serialized PDF does not contain this malformation and can produce reliable viewer output—since different PDF readers may handle invalid values differently.

Selector	**Filtering rule**	**Transformer**
Catalog	Select: `PageMode` if not in list	Replace with: `UseNone`

In our rule, we select the Catalog object and select the `PageMode` key in the Catalog. Since this key is optional, it may not be present—in which case this rule is not applied. We use an allowlist of the values this key in the Catalog can take. If the value for this key is not in the allowlist, we replace it with the default value.

4.5. C4: Removing Null References

Let us consider the second object in the running example in Section 2.2. In this example, The Dests, Outlines, and Pages key all have an indirect reference to another object stored in the same PDF file. However, as we demonstrated in Table 1, there were several cases where these indirectly-referenced objects were missing from the file.

Most PDF readers ignore these keys if they are not mandatory keys. They arrive at this interpretation following clause 7.3.10 in the PDF 2.0 specification. First, any undefined object is considered to be a Null object. Furthermore, in a dictionary, if a key is associated with the Null type, we must treat the dictionary as if this key was absent.

However, this is not easy to implement in a type checker. Although in the running example, three keys are indirect references, any key can hold an indirect reference unless otherwise specified in the specification. For example, the Lang key could be in its object with the following syntax.

```
25 0 obj
(en-US)
endobj
```

To capture clause 7.3.10, we must treat every key as a disjunction between a Null object and the correct type definition. We must also include a flag to ensure that the keys that do not need such a disjunction are skipped.

Instead, we rely on creating reducer rules to remove any Null references in a PDF dictionary. We iterate over every object in the Parsley IR and over every key in the dictionary to find Null references. These objects are then modified to

remove the key with the Null reference. A drawback of this reducer rule is that we may inadvertently remove a mandatory key. For example, suppose the Pages key in the running example holds a reference that does not exist. In that case, our reducer rule may remove this key from the dictionary—creating a type check violation since the key is mandatory.

4.6. C5: Change PDF version

PDF files contain a version string in the header and a field in the Catalog dictionary. However, if both of these version strings are not identical, the higher of the two values takes precedence. If a file's version needs to be updated, either of these two values can be updated to a higher version.

We consider the cases of downgrading and upgrading the version number of a PDF file. For example, we found that the Caradoc PDF parser [16] and PDFCPU [21] do not accept files that are higher than version 1.7. Therefore, to run a PDF file through both these parsers, we must downgrade to version 1.7 from 2.0.

We provide two reducer functions—both take a list of old version numbers and a new version number. We rewrite both the version string in the Catalog and the string in the header in both cases. We did this to ensure there are no parser differentials if specific parsers only read one or the other. We demonstrate our reducer rules below.

Selector	**Filtering rule**	**Transformer**
Version	Select if not `2.0`	Replace with: `2.0`
Catalog	Select `Version` if not `2.0`	Replace with: `2.0`

4.7. C6: Change values in the Info Dictionary

The Info dictionary in a PDF file contains nine optional fields. It contains metadata such as the title, author, creation tool, producer, and creation and modification date. Some PDF creators and conversion tools may add metadata about a user, such as their name and username, and email ID [6], [26].

```
<< /CreationDate (D:20200407135742Z)
/Creator (John Doe; TeX)
/ModDate (D:20200407135742Z)
/PTEX.Fullbanner (This is pdfTeX,
Version 3.14159265-2.6-1.40.20
(TeX Live 2019) kpathsea
version 6.3.1)
/Producer (pdfTeX-1.40.20)
/Trapped /False >>
```

The above code snippet demonstrates the contents of the Info dictionary for a file generated from Latex. We see that it can disclose software versions and creator information in this dictionary.

Hence, we need mechanisms to redact these keys from the Info dictionary to ensure that such information is not exposed when publishing PDF files. Removing these keys from the dictionary does not render a file invalid since all of these keys are invalid.

We identify that the author, creator, producer, and creation date keys can be particularly problematic in exposing details about the origin of a file. Hence, we design rules to redact these crucial keys to ensure better privacy. Unfortunately, since the Info dictionary does not have any mandatory keys, we could not use a selector rule to find it in the IR. Instead, we rely on the trailer of a PDF file to tell us which PDF object contains the Info dictionary. Below, we demonstrate the reducer rules to change the Author and Creator keys in the Info dictionary.

Selector	**Filtering rule**	**Transformer**
Info	Select `Author` and `Creator`	Replace with: `Manul` and `Bobcat`

4.8. A1: Adversarial file generation

We propose another application of the Parsley Reducer API—generating files with deliberate errors to understand if parsers can detect anti-patterns and clear specification violations. For example, we generated a test case to inject an incorrect object ID in the Kids array of every Page Tree Node dictionary in a PDF file.

Since PDF parsers often ignore missing objects, we did not randomly inject a nonexistent object ID. Instead, we append the Catalog or Root dictionary ID to the Kids array. Unfortunately, this action leads to malformed PDFs in three ways.

First, the Kids field in a Page Tree Node must point at other Page Tree Nodes or a Page object—not a Catalog dictionary. Any parser that checks the well-formedness of this tree structure must find this error in the syntax of the tree.

Second, a naive PDF parser that recursively walks the Page Tree structure can enter an infinite loop. As a result, we could trigger a denial of service attack against PDF parsers that do not keep a list of objects already checked.

Finally, a naive text extraction tool can run into an infinite loop in such an infinite tree structure and generate the same text repeatedly. This way, the extracted text would differ from the text extracted from other tools—leading to text-extraction differentials.

Next, we randomly selected 4200 PDF files from our dataset to evaluate our adversarial file generation use case. Finally, we ran each generated file through Qpdf and Mutool clean to see if these tools found our injected malformation.

Table 2. TESTING FILES WE GENERATED WITH ADVERSARIAL RULES AGAINST PDF PARSERS. DATASET INCLUDED 4200 PDF FILES RANDOMLY SELECTED.

Cases	**Mutool**	**Qpdf**
Files Rejected previous accepted	3177	221
Files Timed out, previously ran correctly	18	1

As shown in Table 2, we were able to force qpdf to run over the timeout value (one minute) for one file when the original (unmodified) file runs efficiently within the timeout. The original file ran in 50 seconds, while the modified file did not terminate after running over three minutes in multiple tests. Similarly, we found 18 modified files that caused an infinite loop in the Mutool clean command. Each of these files ran in less than 10 seconds in the unmodified case.

Upon more investigation, we found that this issue was fixed in the latest Mutool release 1.19, following a report in 2020 [30]. However, the Mutool shipping with the package managers on most operating systems is version 1.16. Therefore, all versions of Mutool before 1.19 are vulnerable to a denial of service attack, where a PDF file with a wellformed content stream can cause an infinite loop.

However, we also found that Qpdf does not flag most of the adversarial files we generated as malformed. In contrast, Mutool overwhelmingly finds them malformed with the error "non-page object in page tree (Catalog)." Qpdf provides a warning with the message "expected /Type /Pages, found something else" for the files it rejects.

4.9. Developer effort

Table 3 demonstrates the lines of code needed to use the Parsley normalization tools to apply transformations of varying degrees of difficulty. Using the Reducer API requires some degree of understanding of the PDF specification since we require the developers to specify the transformed type in cases where data is replaced.

We observe that the harder cases of modifying the Page Tree Nodes using cases C1 and A1 require over 100 lines of code to be added. This is because we must account for many combinations of missing keys in Page Tree Nodes, causing an increase in the lines of code. However, the cases of redacting data in objects or replacing them require far fewer lines of code.

Table 3. DEVELOPER EFFORT NEEDED TO WRITE REDUCER RULES

Case	Lines of code added	Complexity
C1	15	Easy
C2	150	Hard
C3	32	Easy
C4	65	Hard
C5	15	Easy
C6	55	Medium
A1	120	Hard

4.10. The need for a reducer

Some of the examples we discussed in our case studies were simple—especially the version editing and fixing page modes. However, C2, C4, and C5 are complex rewriting rules where simple grep- or m4-like expressions would not suffice. For example, to redact keys from PDF dictionary objects, we need to parse the dictionary as a whole and then select the keys for modifications. These dictionaries can also be recursive: dictionaries may contain other dictionaries, streams, or arrays either inline or as a reference.

5. Evaluation

We apply the reducer rules for C1, C2, C3, and C4 as a part of the Parsley reducer since these rules made significant modifications and correct malformed constructs in PDFs. We evaluate the Parsley reducer and serializer to answer the following questions.

- Does the reducer fix bugs and malformations in PDF files?
- How does the Parsley reducer compare with other PDF cleanup tools?
- Do text extraction tools produce the *similar* output for the original and transformed PDF files?

5.1. Dataset

We used two datasets of 10,000 files each. Both these datasets contain files randomly selected from 1 million files collected from GovDocs [15] and CommonCrawl [11].[3] These 20,000 PDF files are representative of a real-world sampling—randomly selected from 1 million files.

We then filtered out the well-formed PDFs from these datasets using the methodology described in Section 3.2.5. As a result, we found that Dataset 1 contained 6753 well-formed files, whereas Dataset 2 comprised 7171 well-formed files. We used the files from both datasets to evaluate our Parsley reducer and serializer methodology.

5.2. Parsley Reducer fixups

This section examines the fixes and changes made by our Parsley reducer. We closely observe files in each of the categories specified in Section 4. Unfortunately, most popular PDF readers cannot render PDF files suffering from C2.

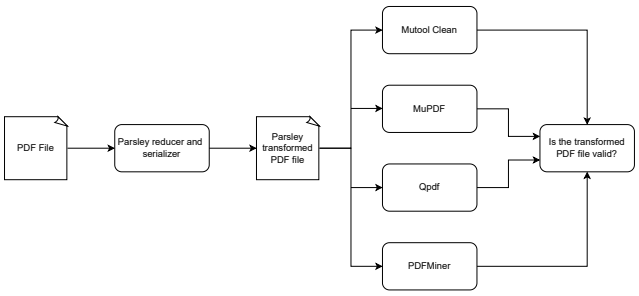

Figure 3. We attempt opening an original PDF file and the transformed file using three PDF tools. We log cases where previously malformed files now open using the various readers.

3. These datasets were collated by Dan Becker of Kudu Dynamics for the DARPA SAFEDOCS project.

Figure 3 outlines our approach to examining the fixes made by the Parsley normalization tools. We rely on the Meriadoc recognizer to compare the parser output of original PDF files and Parsley modified PDF files. After running these PDF files through the Parsley normalization tools, we tried to open the modified files using Qpdf [8], PDFMiner [34], and Mutool [5].

Table 4. COMPARING THE OUTPUT OF VARIOUS PARSERS ON ORIGINAL PDF FILES AND THE PARSLEY GENERATED FILES.

Evaluation tool	Fixups after transformation		Errors after transformation	
	Dataset 1	Dataset 2	Dataset 1	Dataset 2
MuPDF	48	31	0	2
Mutool clean	726	96	0	0
Qpdf	5	16	1	2
PDFMiner	217	401	11	25

Table 4 shows the results of our evaluation. We found that when we ran Mutool clean on these files, many files in both datasets were fixed. Similarly, PDFMiner also fixes a number of files in our datasets with malformations—especially ones following the pattern we described in Case Study C2.

However, we also observe that PDFMiner throws errors on some files that were earlier valid. PDFMiner errors were ASCII decoding or parser errors exposed in the parser. None of the other parsers threw errors on the files PDFMiner failed to parse. We will explore these parser errors in PDFMiner in more detail and suggest fixes in future work.

5.3. Does the serializer work correctly?

In this paper, we set out with the goal of achieving *safe normalizations*. To test our claim, we compare the text extraction output for the original PDF file and the transformed PDF file to ensure that our Parsley serializer works correctly. We use three PDF to text tools: Parsley's text extractor, PDFMiner's pdf2txt [34], and Ghostscript [4]. Figure 4 demonstrates our overall approach of comparing text output.

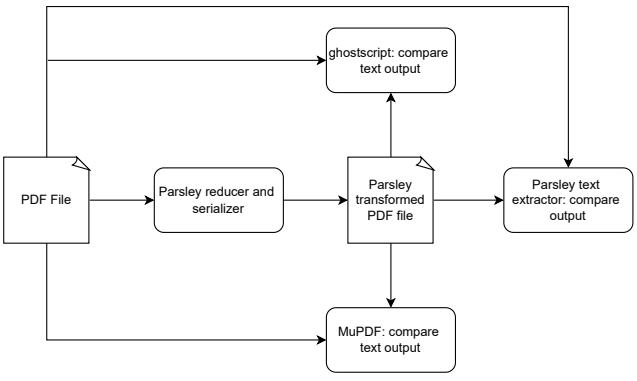

Figure 4. We compare the text extraction output from the original PDF file and the transformed PDF file to ensure we did not damage the PDF files during the transformation.

Mismatches in extracted text. There can be two reasons the extracted text did not match the original text. First, our fixes can provide access to new objects in the PDF file. The text extraction tool may have encountered errors, skipping these objects earlier. Second, the modified file could now contain errors we introduced. Our Parsley serializer *could* introduce certain bugs in the PDF objects.

Hence, we differentiate between these two mismatches by first comparing the extracted text from the original and modified PDF files. If there was a mismatch, we see if we extracted more text or lesser text than the original file. If we extracted more, we would flag this as a fixup. Similarly, we flag cases with lesser text extracted as errors we introduced.

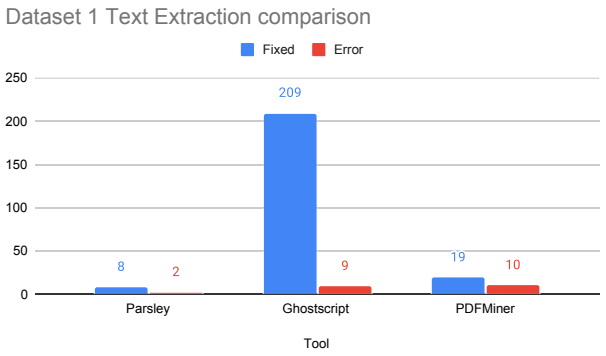

Figure 5. Text Extraction results from Dataset 1 (6753 files). This chart only displays mismatches between running a text extraction tool on the original file and running the same tool on the Parsley-modified file.

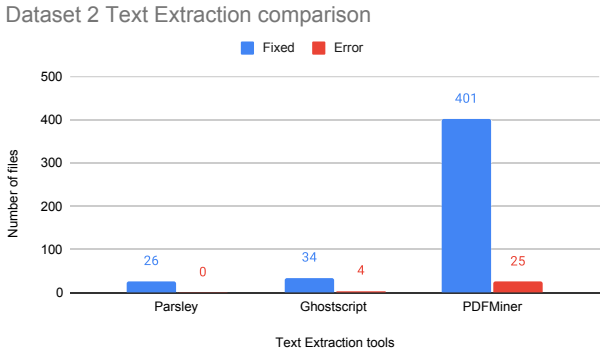

Figure 6. Text Extraction results from Dataset 2 (7171 files). This chart only displays mismatches between running a text extraction tool on the original file and running the same tool on the Parsley-modified file.

Figure 5 and Figure 6 showcase the results of our text extraction comparison experiments. Since these datasets contain very dissimilar PDFs, we see a significant difference in the results. In Dataset 1, we observe that Ghostscript produces more text in over 200 files while throwing errors in 9 files. In contrast, in Dataset 2, we see that PDFMiner generates more text in over 400 modified files than the original files.

These text extraction tools use different algorithms to traverse the page tree nodes and extract text. Therefore, we employed three separate tools to observe different fixes and malformations. We demonstrate that files we've modified generate either the same amount of text or more text in almost all cases—producing safe transformations.

Why do text extraction tools generate more text. We applied case study reducer rules C1 to C4 to evaluate our normalization approach. As we discussed earlier, these rules fix malformations we found in PDF files. For example, C2 finds page tree nodes that are malformed and missing mandatory keys. However, the behavior of text extraction tools on these malformed page tree nodes is not defined. Fixing these malformed page tree nodes allows the text extraction tools to proceed further down a page tree and render text from more pages in the tree.

5.4. Comparing Parsley Reducer with `caradoc clean` and `mutool clean`

Other than using the Parsley reducer, we also ran each file in our Dataset 2 through `caradoc clean` and `mutool clean`. We then validated the transformed PDFs generated by all the three transformation tools through the Meriadoc recognizer. We designed this experiment to check whether Caradoc and Mutool can fix our identified issues.

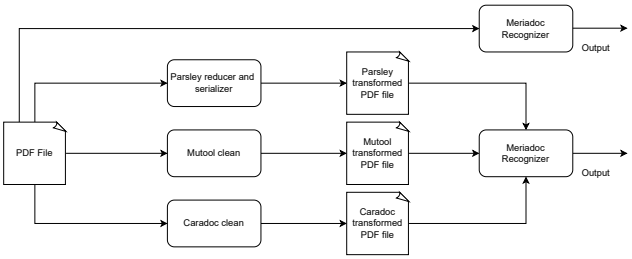

Figure 7. Comparing the transformed files of the Parsley Reducer to files generated by Caradoc clean and Mutool clean.

Figure 7 shows our experimental approach. After transforming the PDF files using three different tools—one of which we built—we run the transformed files through the Meriadoc recognizer with the Qpdf and Mutool parsers.

Table 5. COMPARISON OF CLEANUP TRANSFORMATIONS APPLIED BY PARSLEY, CARADOC, AND MUTOOL AGAINST QPDF AND MUTOOL ON DATASET 2.

Serializer	Fixups after transformation	Errors after transformation
Caradoc	24	0
Mutool	24	32
Parsley	47	4

Table 5 presents the results of our comparison. We find that Caradoc and Mutool both fix files in our dataset. We also find that our Parsley reducer fixes more files that were previously rejected by Qpdf or Mutool than Mutool clean and Caradoc. We investigated the bugs introduced by Mutool clean (32 across Qpdf and Mutool). We found that several of these files generated by Mutool clean were empty, containing no data.

Additionally, we found that a particular type of file X_1 was initially rejected by Mutool but accepted by Qpdf. Mutool complains about several issues, ranging from a broken Xref table, content stream syntax errors, and a corrupt JPEG data segment. Here is a snippet of the error log from running Mutool on X_1.

```
$ mutool clean -s -d x_1
error: cannot recognize xref format
warning: trying to repair broken xref
warning: repairing PDF document
warning: PDF stream Length incorrect
error: syntax error in content stream
error: unknown keyword: 'e'
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: syntax error in content stream
error: unknown keyword: 'r7529.751'
error: syntax error in content stream
error: syntax error in content stream
error: zlib error: invalid block type
warning: read error; treating as end of
file
error: syntax error in content stream
Corrupt JPEG data: premature end
of data segment
```

Mutool does, however, produce a modified PDF file X_1'. Mutool and Qpdf are both also unable to open the file, saying it requires a password—when Qpdf was able to process the original file—and several PDF readers could display the original file.

```
$ mutool clean -s -d x_1_prime
error: cannot authenticate password:
x_1_prime
```

Similarly, we also investigated the four files that Parsley produced that were malformed. We found that these files contained referenced objects but were not present in the file. Mutool rejected these files on these grounds.

5.5. Meriadoc recognizer feedback loop

As we discuss in Section 3.4, we used the Meriadoc Recognizer to find errors in our serializer and then fixed them. Table 6 shows that the number of errors introduced by the Parsley serializer according to both Qpdf and Mutool reduced in the second run, while the number of fixed files marginally increased. This table considers a file to be fixed if the original file was rejected and the modified version was considered valid.

Table 6. DEMONSTRATING THE FEEDBACK LOOP FROM MERIADOC TO PARSLEY ON DATASET 2

Run	Qpdf		MuPDF	
	Fixed	Errors	Fixed	Errors
1	24	15	33	101
2	26	1	35	2

5.6. Summary

In summary, we evaluated our Parsley reducer approach to ensure the following.

- First, we evaluated files generated by Parsley to ensure we do not introduce many malformations in PDF files as judged by a selection of PDF parsers.
- Next, we ran multiple text-extraction tools on our generated PDF files. We found that in a vast majority of cases, we generated more text or the same amount of text as the original file. We have an implementation that is capable of performing *safe normalization*.
- Finally, we compared the fixing capabilities of the Parsley reducer approach to popular tools such as Caradoc and Mutool clean.

6. Discussion

6.1. Reducing Parser Differentials

Since more PDFs fail to include the "Creator" and "Producer" tags in the Info dictionary, it is not easy to pinpoint the software used to create a particular file. Therefore, tools such as the PDF observatory have included a classifier to find the software used to create a given PDF based on specific structural properties of PDF files [13]. The Parsley PDF checker is strict and can pinpoint fine-grained malformations such as missing keys in dictionaries, wrong type implementations, and missing indirect references. Hence, we believe we can use the PDF observatory in conjunction with the Parsley PDF checker to design Reducer rules to remove entire classes of malformations.

In other words, different PDF tools produce different types of malformations. Therefore, we can transform the unlabeled dialects of the PDF specifications that these PDF tools implement and normalize them into a compliant dialect of the PDF format. By transforming several different malformations introduced by a variety of tools into a single, normalized form that is compliant with the PDF specification, we are ensuring that we encounter fewer parser differentials. However, Reducer rules can be used to formalize the de facto specification implemented by a PDF tool and to formalize *grammar drifts*.

6.2. Reducing Developer Effort

Our Parsley Reducer API follows a mechanism similar to a graceful "exception handling" mechanism where we transform data using customized handlers. Large classes of document malforms can be represented using these reducer rules and would not require updates to large codebases. Our methodology standardizes the methods of transforming patterns in documents.

In the future, we wish to explore generating these reducer rules dynamically. We need to use a PDF type checker to find out where specification deviations occur. Such a type checker must not stop the type checking process once it encounters an error—instead continuing to process other objects in the file. Such a process would allow the ability to find all deviations in a file rather than just the first.

We would then need to understand how to fix a particular error. For example, if an object is missing a Type key, but holds every other key, essentially allowing us to predict what the dictionary type is, then we must be able to patch the dictionary with the Type key. However, if a dictionary holds a key that must hold a value from a set of values, such as PDF versions, reverting to a default value would be a direct fix.

6.3. Errors that Caradoc and Mutool fix

We closely inspect the transformations Mutool clean and Caradoc apply in their tools. We find that the fixes by Mutool clean in our dataset fit broadly into three categories. First, we find that Mutool clean fixes truncated files without a cross-reference table or trailer. It calculates the correct offsets and inserts a valid cross-reference table and a trailer.

Second, Mutool fixes files with incorrect string encodings, causing errors. This malformation is similar to the ones we introduced in an earlier version of the Parsley serializer that we subsequently fixed (Section 3.4). The Mutool clean command adds the correct octal encoding to special characters that were previously unescaped.

Finally, we observe that Mutool fixes several off-by-one errors in stream encoding and compression. Given the number of compression, image, and font libraries, such off-by-one errors causing parser differentials are fairly common in PDF files.

We found that Caradoc fixed files in which Mutool could not find objects. Caradoc expands compressed object streams to place all the objects from these compressed streams in the PDF file. We find that several errors caused by compression errors are fixed by decompressing object streams.

6.4. Comparing visual output

In our evaluation, we compared the text output of the original file and the modified files to ensure that we did not break the file in a way text extraction tools can no longer extract any text. Another possible evaluation approach is to compare the visual output of the PDF files [17], [23], [39]. There have been several attempts to compare the visual rendering of PDF files using machine learning or a pixel-by-pixel comparison. In future work, we will explore such a comparison as an additional soundness measure.

7. Related Work

This section relates our Parsley normalization approach and Meriadoc with other approaches. Caradoc [16] and ICARUS [13] are the closest tools to our approach, and they set out with similar goals. We ensure that we can support an API to allow developers to script the rewriting process.

7.1. Caradoc

Caradoc [16] is a PDF parser freely available on GitHub. Caradoc implements three passes to check PDF files. For lexical analysis and parser generation, Caradoc uses MENHIR and OCAMLLEX. They implement a strict type checker and a content stream checker for the other two passes. This way, the Parsley PDF parser follows a similar overall architecture to that of Caradoc.

Caradoc includes a *normalizer* to patch broken PDF files. In Section 5.4, we tested the efficiency of this normalizer by running files through the normalizer in an additional pass before running the PDF file through various parsers. Additionally, we also discuss the errors that the Caradoc tools fix in the generated files in Section 6.3.

Caradoc also checks a PDF file for cycles. They implement indirect references in files as a graph and look for cycles in these structures. A PDF file with cycles can lead to infinite recursion or nontermination if not handled carefully.

This paper goes beyond the normalization work Caradoc does by reducing developer effort by providing a structured format with an API to apply transformations. We also demonstrate several use cases such as adversarial file creation that Caradoc does not support.

7.2. Other PDF fixing tools

Most of the prior work is focused on detecting malicious PDF files using various techniques such as machine learning [14], [28] and structural features [35], [37]. Cowger et al. presented ICARUS, a tool to extract the de-facto specification of the PDF format from a corpus [13]. They also propose converting a PDF file to a safe subset. They require users to define bi-directional lenses to convert a file from one version of the specification to another. This ICARUS approach to using lenses is the closest related work to our Reducer API. Instead, we take the approach of allowing developers to transform objects in a file with more flexibility. We also created a new serializer from the Parsley IR to a PDF file to only serialize objects in use—and to avoid polyglot files [7], [25].

7.3. PDF comparison tools

There are multiple tools available to compare the output of various PDF parsers on one PDF file. These tools are not much different from VirusTotal—a website that runs a host of antivirus scanning engines to make a decision [20]. Allison et al. built a file observatory to analyze the properties and syntax of various PDF files [1], [2]. For example, they find misspellings of various fields in PDF files by using the Levenshtein edit distance. They were also able to identify the tools used to create a PDF based on the syntax it follows.

Cowger et al. presented the PDF observatory—a tool to run a set of parsers against each file in a corpus [13]. For each file, they then iterate over `stdout` and `stderr` to decide whether the parser rejected a file or accepted it. The Meriadoc recognizer follows a similar approach to comparing parser output.

Ambrose et al. introduce the idea of topological differential testing, a mechanism to decipher the behavior of a set of programs on a corpus of inputs [3]. They use this mechanism to learn the de-facto specification of the input format implemented by these programs. Meriadoc uses a similar consensus approach to finding patterns but does not extract a de-facto specification. The PDF observatory and topological differential testing could be used in place of Meriadoc since all three projects share similar goals and approaches.

8. Conclusion

This paper presents a novel approach to a principled, scriptable rewriting of PDF objects. First, we demonstrated our normalization tool on a set of case studies we designed—informed by our research on a large corpus of PDF files. Then, we evaluated the normalization tools against two datasets of PDF files and demonstrated that text extraction tools generate more text from the modified files than the original files.

Much of the future work remains. As we discussed in Section 3.3, our serializer generates a normalized PDF file that contains all the in-use objects in the original file. However, we remove linearization and incremental updates, forcing the modified PDF to diverge significantly from the original file. We will explore how to provide users with flags to support these features in future work.

We will explore another direction of creating a tight-nit feedback loop between the format-aware tracing tools of Meriadoc and our Reducer API. With such a tool, we can dynamically generate these reducer rules with little developer input to fix commonly seen malformations in PDFs.

Acknowledgments

We want to thank Linda Briesemeister and Eric Bond for their insightful discussions, help, and support in putting this paper together. We also want to thank Prashanth Mundkur, Sameed Ali, and Natarajan Shankar for their help in building the Parsley PDF checking engine. We thank the anonymous reviewers and Sean Smith for their suggestions—their changes have dramatically improved the final manuscript.

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract Nos. HR001119C0075 and HR001119C0074.

Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA).

References

[1] Tim Allison, Wayne Burke, Valentino Constantinou, Edwin Goh, Chris Mattmann, Anastasija Mensikova, Philip Southam, Ryan Stonebraker, and Virisha Timmaraju. Research report: Building a wide reach corpus for secure parser development. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 318–326, 2020. DOI 10.1109/SPW50608.2020.00066.

[2] Tim Allison, Wayne Burke, Chris Mattmann, Anastasija Mensikova, Philip Southam, and Ryan Stonebraker. Research report: Building a file observatory for secure parser development. In *2021 IEEE Security and Privacy Workshops (SPW)*, pages 121–127, 2021. DOI 10.1109/SPW53761.2021.00025.

[3] Kristopher Ambrose, Steve Huntsman, Michael Robinson, and Matvey Yutin. Topological differential testing. *arXiv preprint arXiv:2003.00976*, 2020.

[4] Artifex Software, Inc. Ghostscript: an interpreter for the postscript language and pdf files. https://www.ghostscript.com/doc/current/Use.htm, 2016.

[5] Artifex Software, Inc. Mupdf: a lightweight pdf, xps, and e-book viewer. https://mupdf.com/docs/manual-mutool-clean.html, 2016.

[6] Tuomas Aura, Thomas A Kuhn, and Michael Roe. Scanning electronic documents for personally identifiable information. In *Proceedings of the 5th ACM workshop on Privacy in electronic society*, pages 41–50, 2006. DOI 10.1145/1179601.1179608.

[7] Adam Barth, Juan Caballero, and Dawn Song. Secure content sniffing for web browsers, or how to stop papers from reviewing themselves. In *30th IEEE Symposium on Security and Privacy*, pages 360–371, 2009. DOI 10.1109/SP.2009.3.

[8] Jay Berkenbilt. Qpdf: A content-preserving pdf transformation system. https://github.com/qpdf/qpdf, April 2008.

[9] Andrew R Bernat and Barton P Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools*, pages 9–16, 2011. DOI 10.1145/2024569.2024572.

[10] Sergey Bratus, Trey Darley, Michael Locasto, Meredith L Patterson, Rebecca bx Shapiro, and Anna Shubina. Beyond planted bugs in" trusting trust": The input-processing frontier. *IEEE Security & Privacy*, 12(1) pages 83–87, 2014. DOI 10.1109/MSP.2014.1.

[11] Christian Buck, Kenneth Heafield, and Bas Van Ooyen. N-gram counts and language models from the common crawl. In *LREC*, volume 2, page 4. Citeseer, 2014.

[12] James Clark, Steve DeRose, et al. Xml path language (xpath), 1999.

[13] Sam Cowger, Yerim Lee, Nichole Schimanski, Mark Tullsen, Walter Woods, Richard Jones, EW Davis, William Harris, Trent Brunson, Carson Harmon, et al. Icarus: Understanding de facto formats by way of feathers and wax. In *2020 IEEE Security and Privacy Workshops (SPW)*, pages 327–334. IEEE, 2020. DOI 10.1109/SPW50608.2020.00067.

[14] Bonan Cuan, Aliénor Damien, Claire Delaplace, and Mathieu Valois. Malware detection in pdf files using machine learning. In *SECRYPT 2018-15th International Conference on Security and Cryptography*, page 8p, 2018.

[15] Digital Corpora. Govdocs1 — (nearly) 1 million freely-redistributable files. http://downloads.digitalcorpora.org/corpora/files/govdocs1/by_type/files.jpeg.tar, 2021.

[16] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: A pragmatic approach to pdf parsing and validation. In *IEEE Security and Privacy Workshops (SPW)*, pages 126–139. Ieee, 2016.

[17] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D Hansen, and Jonathan C Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4) pages 289–309, 2011. DOI 10.1177/1473871611416549.

[18] Lars Hermerschmidt, Stephan Kugelmann, and Bernhard Rumpe. Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars. In *IEEE Security and Privacy Workshops*, pages 134–141. IEEE, 2015. DOI 10.1109/SPW.2015.29.

[19] Lars Hermerschmidt, Stephan Kugelmann, and Bernhard Rumpe. Towards more security in data exchange: Defining unparsers with context-sensitive encoders for context-free grammars. In *2015 IEEE Security and Privacy Workshops*, pages 134–141, 2015. DOI 10.1109/SPW.2015.29.

[20] Hispasec Sistemas. Virustotal. http://www.virustotal.com/, June 2004.

[21] Horst Rutter et al. pdfcpu: a Go PDF processor. https://github.com/pdfcpu/pdfcpu, 2017.

[22] Brian W Kernighan and Dennis M Ritchie. *The M4 macro processor*. Bell Laboratories Murray Hill, NJ, 1977.

[23] Doron Kletter. Effective system and method for visual document comparison using localized two-dimensional visual fingerprints, December 6 2016. US Patent 9,514,103.

[24] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6) pages 190–200, 2005. DOI 10.1145/1064978.1065034.

[25] Jonas Magazinius, Billy K Rios, and Andrei Sabelfeld. Polyglots: crossing origins by crossing formats. In *Proceedings of the ACM SIGSAC conference on Computer & communications security*, pages 753–764, 2013. DOI 10.1145/2508859.2516685.

[26] Jens Müller, Dominik Noss, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Processing dangerous paths. In *Network and Distributed Systems Security Symposium*. NDSS, 2021.

[27] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6) pages 89–100, 2007. DOI 10.1145/1273442.1250746.

[28] Nir Nissim, Aviad Cohen, Robert Moskovitch, Asaf Shabtai, Matan Edri, Oren BarAd, and Yuval Elovici. Keeping pace with the creation of new malicious PDF files using an active-learning based detection framework. *Security Informatics*, 5(1) pages 1–20, 2016. DOI 10.1186/s13388-016-0026-3.

[29] Derek Noonburg. Poppler, a pdf rendering library. https://github.com/freedesktop/poppler.

[30] Steffen Nurpmeso. Mutool clean: Endless loop. https://bugs.ghostscript.com/show_bug.cgi?id=703092, 2020.

[31] PDF Association. ISO 32000-2 (PDF 2.0). Technical report, Technical report, International Organization for Standardization, 2017.

[32] Dirk Riehle, Wolf Siberski, Dirk Bäumer, Daniel Megert, and Heinz Züllighoven. Serializer. 1997.

[33] Marko A Rodriguez. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*, pages 1–10, 2015. DOI 10.1145/2815072.2815073.

[34] Yusuke Shinyama. Pdfminer: a text extraction tool for PDF documents. https://pypi.org/project/pdfminer/, 2020.

[35] Charles Smutz and Angelos Stavrou. Malicious pdf detection using metadata and structural features. In *Proceedings of the 28th annual computer security applications conference*, pages 239–248, 2012. DOI 10.1145/2420950.2420987.

[36] Ryan Speers, Paul Li, Sophia d'Antoine, and Michael Locasto. Analysis methods and tooling for parsers. https://www.riverloopsecurity.com/blog/2020/06/safedocs-pdf-analysis-methods-intro/, June 2020.

[37] Nedim Šrndic and Pavel Laskov. Detection of malicious pdf files based on hierarchical document structure. In *Proceedings of the 20th Annual Network & Distributed System Security Symposium*, pages 1–16. Citeseer, 2013.

[38] Helen J. Wang, Chuanxiong Guo, Daniel R. Simon, and Alf Zugenmaier. Shield: Vulnerability-driven network filters for preventing known vulnerability exploits. *SIGCOMM Comput. Commun. Rev.*, 34(4) page 193–204, aug 2004. DOI 10.1145/1030194.1015489.

[39] John W Webster III. Method and apparatus for visually comparing files in a data processing system, August 25 1992. US Patent 5,142,619.

[40] Adam Wolnikowski, Stephen Ibanez, Jonathan Stone, Changhoon Kim, Rajit Manohar, and Robert Soulé. Zerializer: Towards zero-copy serialization. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, pages 206–212, 2021. DOI 10.1145/3458336.3465283.