Research Report: Strengthening Weak Links in the PDF Trust Chain

Mark Tullsen, William Harris
Galois, Inc.
{tullsen,wrharris}@galois.com

Peter Wyatt
PDF Association
peter.wyatt@pdfa.org

***Abstract*—In many practical and security-critical formats, the interpretation of a document segment as a *Document Object Model (DOM)* graph depends on a concept of reference and complex contextual data that binds references to data objects. Such referential context itself is defined discontinuously, and is often compressed, to satisfy practical constraints on usability and performance. The integrity of these references and their context must be ensured so that an unambiguous DOM graph is established from a basis of trust.**

This paper describes a case study of a critical instance of such a design, namely the construction of PDF *cross-reference data*, in the presence of potentially multiple incremental updates and multiple complex dialects expressing these references. Over the course of our case study, we found that the full definition of cross-reference data in PDF contains several subtleties that are interpreted differently by natural implementations, but which can nevertheless be formalized using monadic parsers with constructs for explicitly capturing and updating input streams.

Producing our definition raised several issues in the PDF standard acknowledged and addressed by the PDF Association and the ISO. In the future, the definition can serve as a foundation for implementing novel format security analyses of DOM-defining formats.

I. Introduction

The task of parsing may be viewed as receiving a document in an unstructured, serialized form, and trustably building its structured representation. For formats defined in conventional data-description languages that correspond closely to well-understood classes of the Chomsky hierarchy (i.e., context-free grammars, with regular expressions as a critical special case), arguments of trustworthiness fall out naturally from the definition of the format itself. This is in part due to the fact that in such formats, the structured representation of a large segment of a message is defined purely in terms of the structure of its segments.

However, these key properties concerning context-freedom critically do not hold for many practical formats, which define a *Document Object Model* (DOM): i.e., the result of parsing a document is a graph between objects, each of which may contain values bound to a large set of fields. In such formats, it is infeasible to provide context-free definitions of well-formedness because data objects that must satisfy critical relations may not occur contiguously: related objects may not form a tree-like hierarchy in the input stream. Such formats typically introduce a notion of *naming* or *reference* by which objects may refer to each other. A critical practical example of this design pattern—and the motivating example of our work—is the document object model of the *Portable Document Format (PDF)* [12]; PDF data objects include an *object identifier*, by which other objects may refer to them.

In such formats, the structures of references and context take on central importance. In practice, their structure is quite complex in order to support practical demands. E.g., PDF's *cross-reference table* enables **(1)** the interpretation of documents to be strongly mutated by appending content, via *incremental updates*; **(2)** the reference structure to be compressed, via standardized but non-trivial algorithms applied to *cross-reference streams*, potentially combined with conventional cross-reference tables in *hybrid reference files*; and **(3)** large documents to be partially processed incrementally, using separated context, via *linearization*. The parsing of this reference and context information occurs before DOM creation and DOM object validation, introducing a reliance on the correctness of all *pre-DOM* processing.

Difficulties in expressing the structure and semantics of references have resulted in critical security vulnerabilities. The induced *ambiguities* cause different parties to assign wildly different semantic interpretations to the same document. Recently discovered attacks that compromise the integrity and usability of digital signatures [18, 34] use maliciously crafted cross-reference tables. Our work has identified additional exploits against digital signatures and PDF file integrity based on our formalisms of PDF file structure and layout [6].

Furthermore, such formats invite document *cavities*—segments of a document that are not reflected in its semantic interpretation—may store content that is completely unobservable to parser clients. Such cavities are a powerful mechanism for creating *polyglot files* (i.e., files that unexpectedly belong to multiple formats), which themselves have been used in recent critical system security exploits [5].

Thus, even the *pre-DOM* parsing and computation is surprisingly complex (as described in Secs. II and III). Fully defining this pre-DOM computation cannot be done with context-free grammars and weaker formalisms. In the conventional setting, a parser returns a semantic value that is then potentially transformed by further computation, which itself be defined in an attribute grammar or parser client logic. The main limitation of such an approach is that computation on semantic values must then itself effectively parse unstructured data after computing partial contextual information; such parsing logic is exactly what should be expressed declaratively in a grammar

and implemented by a generated parser.

This paper explores a more powerful alternative: monadic parsers equipped with actions for explicitly capturing and setting the parser's input, run in a well-founded sequence of staged computations over semantic values. Using parser combinators is not a new idea: they are available in the distributions of modern industrial strength languages [4, 7, 16, 21, 36]. This approach is explored within an industrial strength case study: validating and parsing the reference tables that are needed to create an unambiguous and trustworthy PDF DOM.

In Secs. IV and V, we formalize the pre-DOM processing using monadic parsers, with Haskell as our "computable specification language." There we show executable code that formalizes the subtle parts of pre-DOM parsing. However, the specification is not yet in itself a complete reference implementation as it excludes the primitive parsers as well as excludes some of the more tedious code.

In Sec. VI, we describe our *implementation* in which we, informed by our specification, separately wrote procedural code and integrated this with the primitive parsers and other required components to create a complete (but less manifestly correct) PDF parser.

Our work is unique in that, to our knowledge, it constitutes the first attempt to use a declarative approach to formalize a comprehensive set of features and integrity relationships in PDF pre-DOM processing that define referential context, specifically cross reference tables, incremental updates, and cross reference table compression within cross-reference streams. These are the first stages in the PDF "Trust Chain," and strongly complements all efforts that rely on a trustworthy formalization of reference in order to validate properties of higher-level document abstraction defined in terms of a DOM.

Our pre-DOM formalization is more subtle than what may be often be implemented by inspecting the PDF standard or many extant documents: in the process of producing our definition, we raised several issues with the current PDF standard which have been acknowledged and addressed by the PDF Association and the ISO. However, there is nothing in the format definition that requires the specific language of combinators to be employed: a key goal of our work is to provide this formalized definition as a worked case study, to be vetted and improved upon using definitions in other experimental data definition languages as they are developed.

Organization: Sec. II presents PDF, its complexities, and vulnerabilities; Sec. III discusses the details and surprising complexities of actually *parsing* the PDF format; Secs. IV and V present and analyze our specification of reference in detail; Sec. VI discusses our implementation; Sec. VII reviews related work, and Sec. VIII concludes.

II. PDF: Structure, Complexity, Vulnerabilities

A. The PDF Format

PDF is a random-access file format that contains 8-bit binary data, line-based ASCII (7-bit) text data (terminated with various end-of-line sequences), and a fixed format ASCII data format in different sections of a single file. The overall

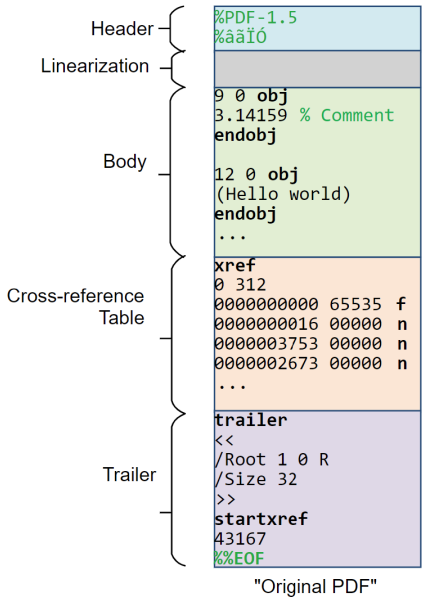

Fig. 1. The structure of a conventional PDF file: labeled sections with selections of representative grammar.

structure of a conventional PDF file without any incremental updates is shown in Fig. 1 and described below, based on the official PDF ISO 32000 standard. PDF 1.5 introduced more compact file structure capabilities known as cross-reference streams and object streams however, since this builds on the following concepts, this modern variation on the PDF file structure will be described later.

The PDF *Header* section contains the file identification marker as an ASCII text comment line `%PDF-` followed by the PDF version as single ASCII digits. An optional second comment line (starting with `%`) containing at least 4 bytes above 127 in value is recommended, to ensure that PDF files are not misidentified as purely 7-bit text files. All file offsets in PDF file are from the `%` sign in `%PDF-x.y`.

The *Linearization* section is an optional optimization section that enables what is commonly known as "fast web view." This section must occur within the first 1024 bytes of a PDF and contains data that enables a Linearized PDF aware parser to quickly display the opening page of a PDF while the rest of a large PDF can download in the background. Linearization data is also invalidated by incremental updates (since an update might change objects on the opening PDF page) and must therefore be checked and ignored. Because not all PDF parsers support Linearization, it is a known form of "parser differential by design," where a *parser differential* is a semantically meaningful difference in the result of two parsers when run on the same document. For the purposes of this paper, we will not consider Linearized PDF further.

The *Body* section is where all indirect PDF objects are defined. Indirect PDF objects are defined as those objects that have "a unique object identifier by which other objects can refer to it (for example, as an element of an array or

as the value of a dictionary entry).” Any PDF object may be defined indirectly: integers, real numbers, strings, arrays, dictionaries, streams, etc. Objects are defined by their object identifier (their object number and generation number pair) followed by the keyword `obj`. The end of every object is defined by the keyword `endobj`. In Fig. 1, object 9 is a real number (in ASCII) followed by a comment (introduced by `%`) and this is followed by object 12 which is a PDF literal string object (enclosed in `(` and `)`). Every indirect object is reached by knowing the file offset to the start of the ASCII integer object number. This offset information for all indirect objects is stored in Cross-reference Tables.

The *Cross-reference Table* section begins with the `xref` keyword. For a PDF file with no incremental updates, the next line will be a cross-reference sub-section text line comprising two integers in ASCII (`0 312` in this example). The first integer is an object identifier and in the case of an original PDF this must be 0. The second integer is the number of objects in the cross-reference subsection.

There are two sets of objects in every PDF document: the in-use list of PDF objects and a free list of PDF objects. Object zero is always the start of the free list as it is not otherwise a valid object number. Each incremental update may also move objects between these two sets. In a conventional PDF, each entry for an object contains a fixed length 20-byte line of text. The first 10 ASCII digits represent the byte offset to the object, followed by a single ASCII SPACE, followed by 5 ASCII digits representing an object generation number. This is then followed by another single ASCII space and the keyword `n` for in-use objects or `f` for free objects. Finally an end-of-line sequence is defined to ensure that the text line entry for each object has a 20-byte fixed length. A parsing subtlety is also that cross-reference sections are the only section in PDF where comments are expressly prohibited.

The *Trailer* section is at the very end of every PDF file. It is defined as the end-of-file comment line `%%EOF` immediately preceded by the `startxref` keyword followed by the file offset (in ASCII as a decimal) to the cross-reference table (i.e. the byte position of the `xref` keyword from the `%` of `%PDF-x.y`). Prior to this (but technically defined as being immediately *after* the cross-reference table) is the trailer dictionary identified by just the `trailer` keyword, rather than the `obj` and `endobj` keywords used in the PDF Body section.

Each incremental update typically appends a Body, Cross-reference Table, and Trailer sections to a PDF file, leaving the entire original PDF unchanged (see Fig. 2). The newly added incremental update trailer dictionary must also contain information referencing the immediately previous cross-reference table by byte offset. The Body section of the incremental update will contain any new or redefined objects. If only fields in the trailer dictionary are updated then a new Body section is not required. The cross-reference table for each incremental update defines changes to objects made by that incremental update. This may include freeing objects by adding them to the free list (the actual indirect PDF objects in the PDF file are not actually deleted), and/or the addition of new objects.

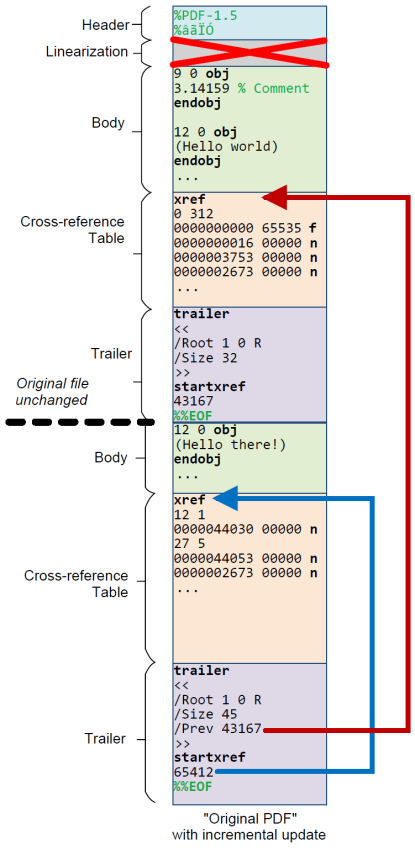

Fig. 2. Conventional PDF file structure with a single incremental update.

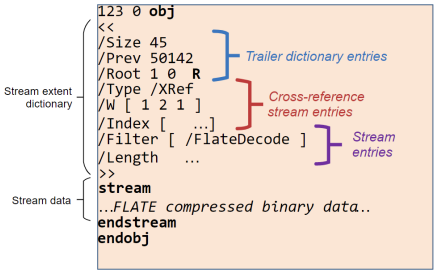

Fig. 3. An example PDF 1.5 cross-reference stream.

As previously mentioned, PDF 1.5 introduced cross-reference streams and object streams because the 10-digit byte offset in conventional cross-reference tables imposed notable limitations on file size. Such files replace the Cross-Reference Table section (including the `xref` keyword) with a PDF stream object containing binary data (see Fig. 3). As with all streams in PDF, cross-reference streams may further reduce file size by compressing files using standard algorithms, such as FLATE. If a cross-reference stream is used, then the trailer dictionary is also no longer used. Instead, the context-defining key/value pairs of the trailer are added to the stream extent dictionary of the cross-reference stream.

PDF 1.5 enabled further optimization via object streams (see Fig. 4). Practical large PDF’s will typically contain many

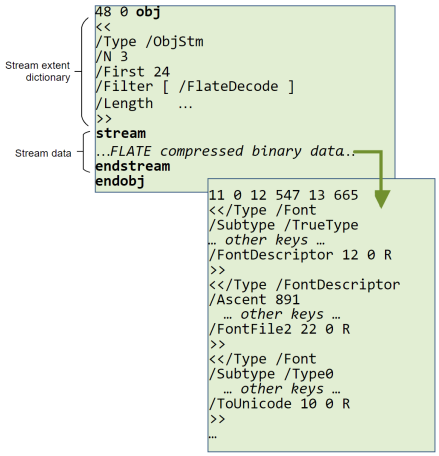

Fig. 4. An example PDF 1.5 object stream and its uncompressed content.

indirect objects; thus the repeated use of the object identifier pair with `obj` and `endobj` keywords can consume a significant amount of space. Object streams are compressible text streams "…in which a sequence of indirect objects may be stored, as an alternative to their being stored at the outermost PDF file level." In place of keywords and object-identifier pairs, object streams use a single object number and a byte offset within the object stream. This ASCII text data can be aggressively reduced using algorithms for stream compression. If object streams are used, then cross-reference streams must also be used. However cross-reference streams may also be used by themselves with the conventional Body section.

As noted above, because both cross-reference streams and object streams are standard PDF streams, they can benefit from stream compression. However this also means that parsers are susceptible to attacks represented by compressed files of a tractable size whose uncompressed data is much larger and intractable to process (i.e., "ZIP bombs"), cycles in object references, and handling of semantic errors (invalid stream extent dictionaries) during pre-DOM processing.

PDF structure is further complicated in the case of "hybrid-reference" files, where both conventional cross-reference tables and cross-reference streams co-exist to support parsers unaware of PDF 1.5. Such files are not addressed in this work.

The number and types of incremental updates that can be added to a file are unlimited. Later updates may "undo" changes from any previous incremental updates by restoring (changing back to in-use) objects previously feed. PDF objects need not be numbered sequentially, with skipped object numbers assumed to be on the free list (although the standard does not state this explicitly).

In effect, incremental updates form a time-line of all changes made to a PDF, with parsing starting at the end of the file with the most recent change back to the original document at the beginning of the file.

B. Root Causes of PDF Complexity

Most data formats can be described by much simpler mechanisms; most language processors (e.g., a Python parser) can be described and parsed by textbook methods (e.g., *lex* and *yacc* are sufficient for most language processors); so what makes PDF processing so much more complex?

As described above, PDF uses random access to byte offsets to then parse lines of text, which may then switch to binary data or fixed-length record parsing. In some, but not all, cases the end of an object can be known a priori, but in the most common case of conventional cross-reference tables, only forward parsing can determine the end of an object. This gives rise to a phenomena we call *cavities* where not every byte in a PDF file is parsed. Such cavities can be used to hold data in an alternate format giving rise to polyglots, or potentially to hold shellcode that might be naïvely loaded into memory and further exploited in vulnerable parsers. In addition, various descriptions in the PDF standard require "parsing backwards" which is an unnatural programming language idiom. Furthermore, some explicit pre-DOM requirements in the PDF standard refer to bytes *before* a given byte offset and that parsers are highly unlikely to check. Our investigations reveal that these requirements are effectively "writer requirements" (unparsers) to support better file reconstruction and recovery in the event of later data corruption. However if these requirements are never checked or enforced then it is possible to use them as attack vectors. The PDF specification also requires multiple sub-languages and computation (such as stream decompression).

If a PDF file accidentally or maliciously fails to adhere to this set of complex PDF file structure rules, or an implementation has bugs, PDF parsers will typically silently attempt to recover by reconstructing the cross-reference table information. This is not defined by the PDF file format standard so ad-hoc algorithms are used. Typical reconstruction algorithms parse from the start of a PDF file, searching for indirect PDF objects (`x y obj ... endobj`) at the start of lines. This can clearly result in ambiguous reconstruction of a PDF DOM and is highly likely to lead to parser differentials.

The PDF DOM created by a set of indirect PDF objects forms a directed graph. Each PDF object is a vertex and indirect references between objects form the directed edges. It is specifically *not* acyclic as PDF formally defines concepts such as parent references to pages and other objects. This is different to most XML-based file formats because XML naturally forces hierarchical relationships via nesting of tags.

The PDF standard defines very few explicit data integrity relationships for pre-DOM parsing. Our work has been to critically review the under-appreciated and under-studied pre-DOM PDF file structure using formal methods to highlight data integrity relationships that can ensure the PDF Trust Chain is robust prior to DOM processing.

C. Vulnerabilities

The majority of prior work researching PDF vulnerabilities start with a pre-existing PDF DOM, ignoring pre-DOM pars-

ing and processing [14, 17, 35]. This work typically looks for obfuscated PDF objects such as JavaScript streams or strings, action objects, URL strings, or file attachment objects that can lurk in various places throughout the PDF DOM. Machine-learning approaches so far have not used feature vectors based on the contextual information in PDF pre-DOM file structure [2, 19]. Prior work in privacy has identified that incremental updates obviously pose a risk for complete redaction and data scrubbing workflows if not processed correctly [1, 37]. Given our recent points about the *PDF Trust Chain* Sec. III-A, it should not surprise us that some PDF attack vectors involve aspects of breaking the *DOM* abstraction. I.e., the root cause occurs pre-DOM.

Supply Chain attacks generalize *Shadow Attacks* [18], wherein an attacker infiltrates a workflow and manipulates a document maliciously, potentially creating a document that is entirely valid. Only later in the workflow, possibly after a digital signature is applied or an official approval step, the malicious data is activated. In the case of digitally-signed documents, this severely undermines confidence and trust in digital signature technology.

Parser differentials constitute a concrete threat to the security of practical systems: attackers leverage differences in output to present different information to different clients (potentially human or other programs) from the same document. Parser differentials arise due to a combination of unintended parser errors, intended parser permissiveness beyond specification, and specification-level ambiguities. Our attempt to formalize PDF's pre-DOM has directly exposed differentials between practical PDF parsers [3].

Polyglots are a potential threat in any system or workflow which relies on different parsers at different stages. If a firewall product determines a file to be of file type X and then only scans for vulnerabilities relevant to file type X, but later software decides the same file is file type Y, an organization is then vulnerable. Polyglots arise from permissive parser implementations and aspects of format specification that allow inclusion of arbitrary data that does not affect processing (e.g. in comments or strings, in freed or unreferenced objects, etc.).

Denial of Service (DoS) is of increasing concern to PDF users, with many PDF services and products utilizing cloud-based processing. In cases such as large-scale automated invoice processing, DoS attacks can also incur additional expenses as well as inconvenience for an organization because exceptions may need to be handled by human operators.

Due to the complexity and subtleties of the PDF standard described above, extant data exhibits many errors in cross-reference tables and related pre-DOM context information. To date, there has not been a comprehensive analysis of PDF file structure faults in extant data caused by defects in PDF writers, due in part to a lack of tooling that comprehensively reports all variations from the standard. However, analysis of open-source PDF parser code bases does highlight tools with varying degrees of lightweight reporting, however all exhibit permissive features related to pre-DOM processing:

- deviation from the required 20-byte fixed cross-reference table entry, with support for 19- and 21-byte variations due to additional white space or incorrect line endings;
- errors in byte offsets to indirect PDF objects and objects in object streams;
- errors in definitions `startxref` byte offsets;
- incorrect trailer dictionary data (such as `Size`), inducing implementation-dependent decisions;
- terminating pre-DOM processing prematurely (i.e. not processing all incremental updates and being susceptible to Shadow Attacks).

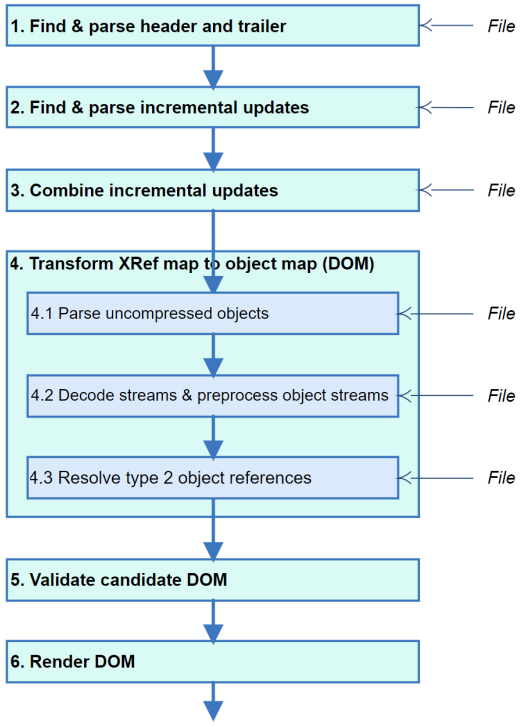

Fig. 5. Stages of PDF Parsing (i.e., the Trust Chain).

III. PDF Parsing

A. Data Dependencies in PDF Parsing (the Trust Chain)

We have touched upon the complexities of parsing PDF, but to appreciate these, one has to understand the dependencies and interactions between the features. The main stages (Fig. 5) include:

- **Stage 1**: Find and parse both the PDF header and the PDF trailer to locate the document trailer dictionary and the start of the *last* incremental update, or the cross reference table of the original document in the case of no updates.
- **Stage 2**: Find and parse each cross reference table or incremental update. Note that this stage uses context from Stage 1 (such as the file offset and trailer information) in order to know *where* and *how* to parse.
- **Stage 3**: Compute the XRef map of the final PDF, by accounting for each set of edits performed by an incremental update. Depending on the laziness of stage 2, this stage may need to further parse the input file.

- **Stage 4**: Transform the XRef map to an object map. This stage is complex, requiring three sub-stages that each parse the input file (see Sec. IV). Upon success, this stage yields a candidate DOM with valid syntax.
- **Stage 5**: Validate that the candidate DOM represents a document graph with valid *semantics*. Only the partially validated DOM, as a mapping from object references to syntactically valid PDF objects, can be the subject of validating of semantic properties such as dictionaries binding expected keys to values of expected types and freedom from unresolvable references.
- **Stage 6**: Render the validated DOM, or selected segments, in the required final representation.

Any error (malicious or otherwise) in earlier stages can affect the output of stages that follow. Note that stages 2, 3, 4.1, 4.2, and 4.3 all depend on inputs from previous stages to determine where and how to parse segments of the PDF input file. Errors that percolate into these stages can cause all forms of havoc (parsing wrong or out-dated data, etc.).

An implementation could potentially merge stages 2, 3, 4.1, 4.2, 4.3, providing a superficial simplicity. However, our eventual argument (see Sec. V-A) is that such an implementation would be overly complex and result in a design for which it is difficult to determine if it correctly implements the standard and to determine that it terminates for all input files.

Although *Validate candidate DOM* (stage 5) is expansive, the properties that it must validate are well defined by the PDF standard. Validation in this stage involves ensuring that each individual PDF object binds all the required keys to appropriate values in the context of its reference in the DOM. E.g., a *thumbnail* object must be an Image XObject and bind the required keys `Height` and `Width` to non-negative integers. A reference expected to refer to a thumbnail that refers instead to a non-thumbnail object (e.g., a font dictionary) is valid in only syntactic—but not semantic—terms. Recent work [33] has resulted in the first specification-derived comprehensive machine-readable model of every object, their attributes and relationships in the PDF DOM. The use of such a model for code generation, DOM validation, or test case generation can significantly reduce the tediousness. *Render DOM* (stage 6) contains its own complexities, including rendering a PDF page as pixels for display or extracting text, depending on context.

This paper does not further discuss stages 5 and 6, focusing instead on stages 1 to 4, referred to as *pre-DOM* parsing or computation. We chose to focus on Stages 1 to 4 as they have been given little emphasis and are the source of multiple vulnerabilities. Errors in these stages can compromise the correctness of the complete format definition, even if later stages are defined as intended when viewed in isolation.

We have been using "PDF Trust Chain" to refer to the sequence of dependent parsers in Fig. 5, although the term *Trust Chain* (or "Chain of Trust") is overloaded with similar meanings in different contexts: e.g., *digital certificates*: a sequence of certificates signing certificates, starting with a root certificate; *supply chain*: a product is no more reliable or secure than its outsourced components; *trusted boot*: unless the boot-loader is correct and non-malicious, there can be no possibility of the operating system being the same; *software stacks*: upper layers are dependent upon lower layers (such as system libraries) and vulnerabilities at the lower layers affect all higher layers. The common idea is that we have layers (or components) that rely on lower layers (sub-components) for their validity. In other words, **a flaw in one layer of the trust chain causes every higher layer to be untrustworthy!**

The intentions behind the term *Trust Chain* are **(1)** to highlight the many dependent stages in PDF processing; **(2)** to highlight the importance of ensuring the pre-DOM parsing, data integrity relationships and computation (the base of our Trust Chain) is correct and secure; **(3)** to remind that the integrity of the DOM cannot be verified independently of verifying all earlier stages; and **(4)** to illustrate that while PDF holds various unique features, it follows a general pattern that may be used to understand formats broadly.

B. Parsing PDF: The details

We described the physical structure of a PDF file in Sec. II, but the processing sequence may not be apparent. The following paragraphs describe the processing necessary to correctly parse a PDF file containing incremental updates.

PDF parsing begins by locating the PDF Header, as it is not uncommon for PDF files to have preamble bytes (such as an HTTP header, HTML or XML). The offset to the start of the PDF file (PDF offset zero) within a physical file can then be determined from the `%` sign in `%PDF-x.y`.

Processing continues by seeking to the end-of-file and locating the *last* `startxref` keyword, a numeric value is then parsed (giving us a byte offset to the last cross-reference table in the PDF). It should be verified that `%%EOF` follows the numeric value. Note that this requires parsing *backwards* which is unnatural for data definition languages as well as many programming languages; this is a proven source of parser differentials. The offset to the cross-reference table must be adjusted for any preamble bytes prior to the PDF Header.

The parsing sub-language depends on the form of the incremental update, with conventional cross-reference tables being simpler and largely independent of other processing. Cross-reference streams however are more complex as they are often compressed and thus require the pre-DOM parser to "trust" the stream extent dictionary data.

The parser must then locate either the `xref` keyword for conventional PDF cross-reference tables, or a cross-reference stream, identified by tokens of the form `x y obj` . A parser must then determine if the PDF object is a semantically valid cross-reference stream by further parsing the stream extent dictionary and validating the necessary key/value pairs and also recognizing the `stream` keyword after the dictionary end token `>>` and the `endstream` keyword (see Fig. 3).

In the case of conventional PDF cross-reference tables, after the cross reference table will be the trailer dictionary, identified by the `trailer` keyword. Note that this algorithm is at odds with the file structure as defined in the PDF standard: the Trailer section is formally defined to contain the trailer

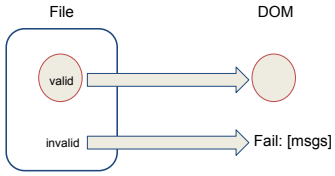

Fig. 6. Validator

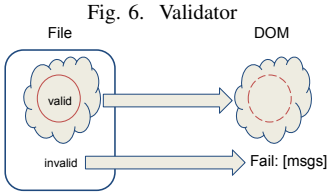

Fig. 7. Parser

dictionary and `startxref` keyword, yet the parsing algorithm requires locating the first trailer *after* the end of the cross-reference table (versus the last trailer dictionary above the `startxref` keyword). Alternatively for PDF 1.5 and later files with cross-reference streams, the trailer dictionary data will be in the stream extent dictionary of the cross reference stream.

Any previous cross-reference data is identified by the value of the `/Prev` entry in either the trailer dictionary or the stream extent dictionary of a cross-reference stream. The value of the `/Prev` key is the byte offset to the immediately preceding cross-reference data which, again, can either be a conventional cross-reference table and to the start of the `xref` keyword, or to a cross-reference stream. This process repeats, working from the most recent incremental update back through time to the original PDF document.

In each incremental update, the trailer dictionary is required to duplicate all keys from the previous trailer and update accordingly. Of particular note is the `/Size` entry, which must be one greater than the largest object number allocated in the PDF file. Objects with numbers greater than `/Size` are defined to be the special PDF `null` object.

Data in each cross reference table must be parsed to identify the byte offset to the start of each PDF object, whether this be a file offset to an indirect object in a Body section, or a relative object position within an object stream (and where the object position is transformed to a byte offset within the object stream from the first line of text in the object stream). Note also that with conventional PDF cross-reference tables there is no definition for the byte offset to the end of an object, however for cross-reference streams this can be pre-determined.

C. Specifying a Parser, not Specifying a Validator

A *key* idea in the DARPA-funded "SafeDocs" program is to understand and capture extant—used in the wild—PDF parsers. Creating an implementation that perfectly matches the PDF specification is not particularly useful and would only parse a small fraction of extant PDF files.

See Fig. 6 where we show a validator (or a format *recognizer*), its function is to produce a DOM when we have a valid PDF, it must fail otherwise. Compare this to figure Fig. 7 where we show a parser which has a **completely different requirement**: to efficiently construct the correct DOM when given an valid PDF. A *parser* is willing, if not happy, to accept files beyond the pale of PDF. In the *parser* diagram, we use a cloud to suggest that for various parser tools, each is going to accept a different subset due to

- redundancies in the format allow for different choices,
- the tools do not interpret the Standard uniformly,
- tools may traverse and evaluate implicit data structures differently, and
- tools differ in how they allow for "minor" errors or do error recovery.

Our goal for a *parser specification* is to encompass any reasonable and correct "cloud". This goal is somewhat subjective, but generally this will imply that we attempt to capture the laziness of various tools that only parse or validate values upon demand.

However, at the same time, it would be very desirable to write a single piece of code from which we could extract both a *parser* as well as a *validator*. To resolve this conflict, we have **(1)** specified an implementation to be *lazy* whenever possible; in other words, whenever possible it defers the reading, parsing, and validation of data; and **(2)** extended this specification with separate `validate` predicates that, when executed, would extend our implementation to form a complete validator.

One of our objectives is to test implementations with respect to our parser specification; we would test by validating that when an implementation produces a DOM, that DOM is equivalent to the DOM produced by the specification. This is why we want our specification to be lazier than most implementations.

IV. Specifying pre-DOM PDF Parsing

A. Haskell as Specification Language

We have formally specified the PDF pre-DOM in the Haskell programming language [15]. A detailed presentation of Haskell's features is beyond the scope of this paper, so we note only that it is a statically typed, pure, functional programming language with lazy semantics. Haskell is well-suited among *general-purpose* languages for declaratively expressing computation. Although Haskell may not be ideal for communicating with non-programmers, having a definition that is unambiguous and executable has shown itself to be a useful artifact.

The specification formalizes pre-DOM processing, stages 1 to 4 (Fig. 5); it is defined in terms of several primitive parsers (see Appendix A for a comprehensive list, with types) whose definitions are not included. Such primitives are arguably better specified in formalisms such as the publicly available *DaeDaLus Data Description Language* [9].

The specification supports PDF's latest version (2.0), including compressed objects and cross-reference streams. It (safely) ignores Linearization data, and in hybrid-reference PDFs it ignores the conventional cross-reference tables designed for

```
pPDFDom :: P DOM                                                          1
pPDFDom =                                                                 2
    do                                                                    3
    -- Stage 1:                                                           4
    (seek,xrefOffset,version) ← findHeaderAndTrailer                      5
    -- Stage 2:                                                           6
    updates ← parseAllIncUpdates seek xrefOffset                          7
              :: P [(XRefRaw, TrailerDict)]                               8
    -- Stage 3: combine all the updates to get a single xref table       9
    xrefs ← combineUpdates seek updates                                   10
            :: P (Map ObjId (Offset :+: Type2Ref))                        11
    -- Stage 4:                                                           12
    dom ← transformXRefMapToObjectMap seek xrefs                          13
          :: P DOM                                                        14
    -- final checks and validations:                                      15
    finalValidations dom version updates                                  16
    return dom                                                            17
```

Fig. 8. A formalization of pre-DOM stages 1–4 in Haskell.

pre PDF 1.5 readers. It processes signatures (as incremental updates), but it does not support validation of signatures.

B. Core definitions

The function `pPDFDom` (Fig. 8) implements stages 1 through 4 (illustrated in Fig. 5), its type (line 1) indicates that it is a *monadic* parser `P` that returns a value of type `DOM`. In general, monads are a rich class of parameterized types that describe a surprising variety of data and computations. For the purposes of this paper, it suffices to view a monad as a construct that allows us to sequence effectful *actions* in a purely functional manner; effects can be global variables (or mutable state), exceptions, I/O, and etc.

Haskell provides a syntactic form for concisely sequencing actions, structured as the keyword `do` followed by bindings of the form `x ←A`, which denotes that the action `A` is performed and its resulting value is bound to variable `x` to be used by subsequent actions; for examples, see Fig. 8.

The specific parser monad `P` abstracts effects on a parser's state. Its state includes **(1)** one read-only variable, which stores the PDF file being read, and **(2)** one mutable variable `readLocation`, which stores the offset in the file at which the next byte is read. `readLocation` is accessed by primitive parsers (which have type `P`) and is updated by sequencing the parsing action

```
seekPrimitive :: Offset → P ()
seekPrimitive off = <update readLocation with 'off'>
```

`P` parsers implicitly track parsing errors, which are thrown when an input document is not in the defined format. I.e., each parsing action of the form `x ←A` can be viewed as attempting to parse according to action `A`, binding a result to variable `x` only on success and failing otherwise.

To aid understandability, expressions are occasionally annotated with types (see Fig. 8 lines 8, 11, and 14).

C. Stage 1: find and parse the header and trailer

Stage 1 is defined as follows:

```
findHeaderAndTrailer :: P (SEEK, Offset, Version)
findHeaderAndTrailer =
    do
    -- find "%PDF-x.y" near start of file,
    -- searching the first 1024 bytes:
    (version, headerOffset) ← findPDFHeader 1024

    -- search backwards from EOF for 'startxref',
    -- gives up after 5000 bytes:
    xrefOff ← findStartxrefThenParseToEOF 5000

    let seek n = seekPrimitive (headerOffset+n)
    return (seek, xrefOff, version)

type SEEK = Offset → P () -- type of seek
```

The magic number 1024 is not in the ISO standard, but comes from an earlier Adobe implementation note and many years of legacy. Implementations going beyond this could result in a parser differential. The magic number of 5000 stems from an acknowledgment that implementations often just seek to n bytes from the end of the file (e.g, 5000) and search forwards from there: caveat, they need to return the last found `startxref`, not the first found.

File offsets in a PDF are not defined in relation to the beginning of the file, as might be expected, but instead to the beginning of the *PDF header*; the `seek` provides a clean abstraction that defines this aspect. But we will need to pass `seek` to all the actions that need to "seek" in the PDF file (note this being done in Fig. 8).

Alternatively, we might have added a function to the monad to set the offset to where we should `seekPrimitive`

```
setBeginningOfFileOffset :: Int → P ()  -- seekPrimitive always uses.
```

although this shortens the code, it does not allow us to enforce that the offset is only set one time. The purity of our current approach does make the data dependencies more explicit.

Because the two function calls (lines 5 and 8) have no data dependencies, they can be performed in any order. The spec has no global variables beyond `readLocation` and neither function depends on it.

The full definition of `findStartxrefThenParseToEOF` is omitted, to simplify the presentation. In abstract terms, it:

1) Finds the "EOF marker" `%%EOF` (near the end of the physical file);
2) Parses "backwards" to find the last `startxref` keyword followed by an end-of-line sequence;
3) Parses the integer value encoded as a sequence of ASCII bytes (this represents the byte offset in the PDF file, which as noted above is adjusted for any preamble prior to the header).

D. Stage 2: find and parse incremental updates

Stage 1 only finds the start of the trailer; the trailer dictionary itself, along with the complete cross-reference table, are defined in stage 2 as follows:

```
type Update = (XRefRaw, TrailerDict)                                      1
                                                                          2
pIncUpdate_Traditional :: SEEK → P Update                                 3
pIncUpdate_Traditional seek =                                             4
    do                                                                    5
    (xrefRaw, xrefEndOff) ← pXrefRaw :: P (XRefRaw,Offset)                6
    validate $                                                            7
      verifyXrefRaw xrefRaw                                               8
```

```
    -- this ensures no duplicate objectIds
  seek xrefEndOff
    -- This seek is needed because pXrefRaw doesn't need to read
    -- the entries of each XRef subsection, so let's leave the
    -- current file read location after the xref table.

  pSimpleWhiteSpace -- no comments allowed between XRef table and ..
  keyword "trailer"
  trailerDict  ← pDictionary
  trailerDict' ← dictToTrailerDict trailerDict
                 -- ensures dictionary has proper keys
  return (xrefRaw, trailerDict')
```

`validate` (line 7) is a special function used to demarcate semantic checks (line 8) that are not necessary to create the DOM but which could detect invalid or inconsistent PDFs. The check could be used in a "validate" mode, but would not necessarily be performed by all implementations.

The resulting subsections of the `XRef` table are unstructured, each of type `XRefRaw`, rather than a sequence of fully parsed `XRef` entries; this aspect of the definition is revisited and fully motivated in Sec. IV-E. The partially structured representation still retains a set of `ObjId`'s and for each, the offset of its of its `XRef` entry. Constructing the representation critically relies on a precise requirement established in the standard: an `XRef` entry must have exactly 20 bytes.

The `XRef` table and trailer themselves are simply instances of incremental updates, which are parsed as follows:

```
parseAllIncUpdates :: SEEK → Offset → P [Update]
parseAllIncUpdates seek offset =
  parseAllIncUpdates' IntSet.empty seek offset

parseAllIncUpdates' :: IntSet.IntSet → SEEK → Offset → P [Update]
parseAllIncUpdates' prevSet seek offset =
    if offset `IntSet.member` prevSet then
      error "recursive_incremental_updates"
    else
      do
      seek offset
      (xref,trailerDict) ← pIncUpdate seek
      case trailerDict_getPrev trailerDict of   -- lookup 'Prev' key
        Nothing      → -- no Prev key, we're done:
                       return [(xref,trailerDict)]
        Just offset' → -- Prev key found, find updates starting at
                       -- offset':
                       do
                       us ← parseAllIncUpdates'
                              (IntSet.insert offset prevSet)
                              seek
                              offset'
                       return ((xref,trailerDict):us)
```

Much of the above definition is dedicated to ensuring that potential loops in the `Prev` offsets do not cause the definition to lose well-foundedness (or in operational terms, that the natural corresponding parser does not loop infinitely). The set `prevSet` tracks the offsets of every update processed. The definition **(1)** `seek`s to the `offset` and parses the update, then **(2)** checks for a `Prev` key. If no key is present, then parsing is complete; otherwise, parsing continues with an extended set of offsets.

`pIncUpdate` supports `XRef` tables as defined in versions of PDF up to 1.5 and represented using cross-reference streams:

```
pIncUpdate :: SEEK → P (XRefRaw,TrailerDict)
pIncUpdate seek =
      pIncUpdate_Traditional seek
  .|. pIncUpdate_XrefStream seek
      -- I.e., parse one or the other,
      -- syntactically, they are mutually exclusive.

pIncUpdate_XrefStream :: SEEK → P (XRefRaw,TrailerDict)
pIncUpdate_XrefStream seek = notImplementedYet
```

The function `pIncUpdate_XrefStream` is more complex than `pIncUpdate_Traditional` but the results are the same: it finds the subsections without parsing the `XRef` entries.

Upon success, stage 2 produces values from which a client can directly determine all `ObjId`'s in the PDF and produce the trailer dictionaries and incremental updates. The only PDF values that have been parsed are trailer dictionaries, but many documents can be rejected, including those with duplicated `ObjId`'s in the `XRef` tables.

E. Stage 3: combine incremental updates

In stage 3, updates are processed from last in file to first:

```
combineUpdates :: SEEK
               → [(XRefRaw, TrailerDict)]
               → P (Map ObjId (Offset :+: Type2Ref))
combineUpdates seek updates =
    do
    let (xref,dict):us = updates -- safe because: null(updates)==False

    -- parse all xref entries in last (near EOF) update:
    xrefEntries ← mapM (thawXRefEntry seek xref) (getObjIds xref)

    -- initial Map:
    let tbl0 :: Map ObjId XRefEntry
        tbl0 = M.fromList xrefEntries

    -- merge each update into tbl0:
    tbl1  ← foldlM (mergeUpdate seek) tbl0 us
    return (removeFrees tbl1)

type XRefEntry = Free :+: (Offset :+: Type2Ref)
```

Each incremental update can add new objects, mark existing objects in use as free, or update objects. We've observed implementations processing the updates from last to first and well as first to last. Either could work, but we choose to process from last to first because it can be a lazier (and more efficient) approach that could do less parsing of `xref` entries.

The bulk of the work is done in `mergeUpdate`, it is effectively computing the union of two maps with a couple twists: (1) the second map `(xref,dict)` contains unparsed `XRef` entries, so we will need to parse these, but (2) we only parse the `XRef` entries that are needed, if we already have an entry, previous updates do not need to be looked at.

```
mergeUpdate :: SEEK
            →  (Map ObjId XRefEntry)
            → (XRefRaw, TrailerDict)
            → P (Map ObjId XRefEntry)
mergeUpdate seek map0 (xref,dict) =
    do
    let objIds       = getObjIds xref
        neededObjIds = objIds \\ M.keys map0
                       -- set subtraction

    -- only parse (thaw) needed XRef entries:
    newEntries ← forM neededObjIds
                      (thawXRefEntry seek xref)
    return
      (M.union map0 (M.fromList (newEntries :: [(ObjId,XRefEntry)] )))
```

We are currently ignoring one complication: we are not constraining `ObjId`'s using the values bound to trailer dictionaries' `Size` key as required in the standard: "Specifically, any object in a cross-reference section whose number is greater than the bound value shall be ignored and treated as missing." And thus that object identifier shall be considered to be the PDF object `null`.

Not only is stage 4's definition complex, but there are multiple, apparently workable variations of `mergeUpdate`, with distinct semantics:

- It is not completely clear from the standard when we apply the "greater than `Size`" bound [31].
- We might not process/check incremental updates if all `ObjId`'s have been defined (this could be determined from `Size`).
- `validate` instances could prohibit double frees of objects, updates of non free objects, updates that do not increment the generation number, frees of non-existent object ID's, and other unconventional uses of generation numbers. Other instances could validate that the offset of new objects in each update are defined in the update's "body region" (i.e., they do not point to previous nor subsequent regions).

Given that some of the properties may be indicators of shadow attacks or PDFs that are otherwise suspicious, Stage 3 is thus critically important as a stage for security validation.

While Stages 2 and 3 could feasibly be defined as a single stage, it would effectively force every compliant processor to parse and validate to a degree that may not be needed by many applications. Our specification was designed instead to include as much validation as possible as instances of `validate`.

F. Stage 4: Transform XRef Map to Object Map (DOM)

The apparent complexity of stage 4 in particular was a primary motivation of our work; it is implemented as the following function:

```
transformXRefMapToObjectMap
  :: SEEK → Map ObjId (Offset :+: Type2Ref) → P DOM
transformXRefMapToObjectMap seek xrefs0 = do
```

The stage contains three distinct substages, each of which directly parse the document's bytes. Stage 4.1 transforms `xrefs0` into `xrefs1`, with types:

```
  xrefs0 :: Map ObjId (Offset              :+: Type2Ref)
  xrefs1 :: Map ObjId (TopLevelDef_UnDecStm :+: Type2Ref)
```

The type `a :+: b` is a synonym for the Haskell sum type `Either a b`, which has constructors `Left` and `Right`. In stage 4.1, traditional `XRef`'s are resolved and the resulting objects are parsed:

```
    -- Stage 4.1: parse all uncompressed objects
    xrefs1 ← mapM
               (mMapLeft (λo→ do {seek o; pTopLevelDef_UnDecStm}))
               xrefs0
```

Further parsing and validation is not possible in stage 4.1 because object streams, and streams in general, cannot yet be decoded: these top level objects cannot always be parsed without resolving `Length` (and similar) keys, which may be bound to indirect references to integer values.

Stage 4.2 further refines the `xref` map, computing the second from the first, with the following types:

```
  xrefs1 :: Map ObjId (TopLevelDef_UnDecStm :+: Type2Ref)
  xrefs2 :: Map ObjId (TopLevelDef          :+: Type2Ref)
```

`xrefs2` is defined as

```
    -- Stage 4.2: decode stream bytes, pre-process ObjStm streams
    xrefs2 ← mapM
               (mMapLeft (extractStreamData xrefs1))
               xrefs1
```

`xrefs1` is a sufficiently complete `DOM` that it can be used to resolve integers needed to decode any currently undecoded streams. If a claimed reference to an integer is not bound in the DOM, then the document is not a valid PDF: all indirect values bound to `Length` keys must be at the document's top level.

At this point, in `xrefs2`, we still have `ObjId`'s that point (via `Type2Ref`) into `ObjStm` streams. Only in stage 4.2 were we able to decode the stream inside of which is the object to be parsed.

Values computed in stage 4.2 are sufficient for computing document cavities. Object streams are pre-processed, but any objects that they contain have not yet been parsed.

Finally, stage 4.3 computes `domCandidate` from `xrefs2`, with the following types:

```
  xrefs2       :: Map ObjId (TopLevelDef :+: Type2Ref)
  domCandidate :: Map ObjId TopLevelDef
```

`domCandidate` is defined following a similar pattern to computation in previous stages. However, unlike previous stages, sum types are no longer used; the result is a map from `ObjId`'s to PDF Values:

```
    -- Stage 4.3: resolve Type 2 references
    domCandidate ← mapM
                     (return `either` derefType2Ref xrefs2)
                     xrefs2
    return domCandidate
```

At this point, every object referenced via the cross-reference table has been correctly parsed. However, objects are not yet read or parsed if they are not referenced, whether they occur in the document's body or within an object stream.

The *Catalog Dictionary* (which has the optional `Version` key) may have been in an Object stream, so in general the intended version of the document cannot be determined until this point; it is arguably unclear if this was intended when defining the PDF standard.

G. Further Validation

We have a few checks and validations that we are unable to do until the DOM is created:

```
finalValidations dom version updates =
    do
    version' ← updateVersionFromCatalogDict dom version
    let etc = unknownKeysInTrailerDict updates

    if version' > (2,0) then
      warn "PDF_file_has_version_greater_than_2.0"
    else
```

```
// xref and dom start with nothing in them:
//  - in reality, their sizes would be dynamically allocated.
XREF_ENTRY xref[Size];
DOM_ENTRY  dom[Size];

// 'dom' is updated dynamically, on demand, via 'deref':
PdfValue deref(ObjId oi) {
  if (evald(dom[oi]))
    return dom[oi];
  else if (infiniteloopdetected())
    quit ();
  else {
    o = lookupOffsetInXref(oi); // updates xref[oi]
    seek(o);
    v = parseObject();
    dom[oi] = v;
    return v;
  }

PdfValue parseObject (){
  ...
  if (some Stream object) {
    ...; len = deref(oi'); ... // need this to finish parsing!
  }
  ...
}

OFFSET lookupOffsetInXref(oi) {
  if xref_evald(xref[oi]) then
    return xref[oi];
  else {
    // follow Prev pointers if not in top xref
    //  - make sure no infinite loop in chasing Prev's
    /* ... */
    xref[oi] = offset;
    return xref[oi];
  }
}
```

Fig. 9. D, A Single Stage, Imperative Parser.

```
  -- version' ≤ (2,0)
  when (not (null etc)) $
    warn "trailer dictionary has unknown keys (per PDF 2.0)"
validate $
  versionAndDomConsistent version' dom

validate $
  trailersConsistentAcrossUpdates updates
```

Due to PDF-1.5 additions, we are now in the odd position that we cannot determine the PDF version until after we have created the DOM. So although we can check that the created `dom` is consistent with `version'`, we cannot use `version'` to validate any of the other processing in stages 1–4. For instance, one would like to verify that Object Streams are not used when the version is PDF-1.4 or earlier. Such a check would be possible if we were to update the spec to pass more information through the stages so we could verify more after computation of the DOM.

V. Assessing The Specification

A. Comparison with a one-stage implementation

One might ask if it's possible to write a single stage version of the specification. I.e., Can we do without so many stages? The answer is *Yes, but at a cost we don't want to pay!* Let's refer to the our multi-stage specification as the S (staged) specification. We will compare it to a specification defined as a single stage, the D (dynamic) specification, in which stage 1 is the same but stages 2, 3, and 4 are merged together; the result is considerably more difficult to understand, which is seemingly inherent to the design. In our experience, PDF tools generally have structure more similar to D than S. D is intrinsically an imperative algorithm, iterating and recursing over a structure that will be incrementally updated; see Fig. 9 for a sketch of a possible implementation in C-like notation. The key things to note about D are

- `XREF_ENTRY` and `DOM_ENTRY` are both types that mutate progressively from unparsed/unknown to fully evaluated.
- `deref()` and `parseObject()` are mutually recursive functions.
- implementing `infiniteLoopDetected()` is non-trivial.

Also to note about D:

- it is *not* equivalent to S: D potentially reads less of the input file and accepts more input files. it is naturally "lax": it would for instance allow a `Length` to be stored in an `ObjStm` stream as long as an infinite loop wasn't detected. It would be possible to extend (and complicate) D to approximate S better.
- it is nicely lazy if the PDF tool doesn't need to `deref` every object identifier, even more lazy than S.

S is strongly preferable to D for two primary reasons:

- It corresponds to the standard, and does so obviously.
- It does not use general recursion, so it is obvious that S algorithm terminates on all inputs.

And there are numerous secondary advantages of S:

- The functional, declarative, and typed structure enables us to understand the stages conceptually. (Even if one chooses not to implement in a like manner.)
- It demonstrates the non obvious fact that one can implement stages 2–4 and keep the "state of evaluation" of each object identifier the same in each pass.
- We know exactly what is and isn't parsed, regardless of the order in which one traverses the `xref`.
- It is intrinsically parallelizable due to the extensive use of map-like combinators.
- The declarative nature of S makes it very amenable to modification: e.g., the simple addition of the `validate` operator.

B. Incremental Updates and Signatures: What A Tangled Web

PDF supports the ability to use digital signatures to sign documents as part of the PDF standard itself. Signed documents can also subsequently be signed or "incrementally updated," repeatedly.

Digital signatures are just a special case of an incremental update, and thus any PDF reader will inherently support and process them as any other incremental update. PDF digital signature are implemented as a PDF annotation which enables backwards compatibility and allows parsers that don't specifically know about digital signatures to still parse them (albeit

Ref.	Summary
[28]	Mandated indirect reference requirements not enforced: are they real requirements?
[27]	Hybrid-reference PDFs should be deprecated
[26]	Clarification on DSS/DTS incremental updates
[25]	Add explicit file requirements for incremental updates on signed files?
[32]	`/XRefStm` key description is confusing/wrong
[29]	Clarification on xref sub-section line syntax
[30]	Semantics for use of ObjStm `/Extends` key are unclear/insufficient
[31]	Trailer `/Size` entry definition is described inaccurately/ambiguously

TABLE I
RELATED ISSUES IDENTIFIED IN THE PDF STANDARD.

the digital signature itself is unvalidated and the document integrity is unknown).

However, digital signatures and attempted support by a reader can induce complications, such as Shadow Attacks [20, 24]. To validate a PDF with a digital signature entails identifying at which iteration of the PDF document the digital signature was applied and then validating the digital signature in the context of that specific DOM and the objects that were in effect at that instance in time. This means that later incremental updates must be excluded from that processing.

But this method completely *breaks* the incremental update abstraction. As we discussed, stages 2 and 3 handle all the incremental update processing and then the information is gone as it is unneeded by subsequent stages. This was a reasonable design decision until "incremental updates" were unadvisedly repurposed for signatures. Now an incremental update has critical semantic content, and a reader supporting validation and display of signed PDFs will need to keep track of the contents of the DOM at each signature.

Our specification does *not* validate signatures. Supporting such a feature is non-trivial but feasible within the S approach, but it is unclear to what extent it could be defined within the D approach.

C. Analysis of results

The specification as presented adheres to the latest PDF 2.0 ISO standard. In producing the format definition, we raised multiple issues concerning the PDF standard with the PDF Association (see Table I); several of them have already been addressed by updating industrial PDF processors.

The specification's conformance to the standard arguably limits its applicability given that many PDF tools are permissive and allow multiple variances from the standard. However, the opportunity for us is to use our specification to encode some of these common extensions and to explore if the combination of these extensions are truly unambiguous.

It would be feasible to refactor our specification to support different PDF variations, specifically:

1) strict PDF-2.0.
2) strict PDF-2.0, validating everything.
3) PDF-2.0 with some common extensions that are determined to be unambiguous.

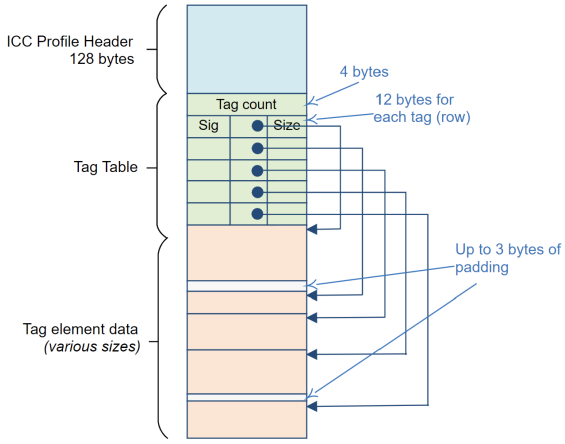

Fig. 10. ICC File Structure.

Our declarative approach lends itself to this, we've already demonstrated how we can easily achieve both 1 and 2 with the `validate` combinator.

Our specification owes its concision to multiple factors:

1) our intention to make it as clear as possible,
2) ignoring some "engineering" aspects (e.g., informative error messages, recovery, etc.), and
3) omitting code for some of the simple, tedious functions.

Future work might also involve adding further validation checks, including **(1)** processing linearization data and ensuring consistency with the non-linearized DOM; **(2)** processing both "branches" of a hybrid PDF and ensuring some form of consistency between pre 1.5 readers and 1.5+ readers; **(3)** adding signature validation, see Sec. V-B.

See Sec. VI for details of the transition from specification to implementation.

D. Relevance to Other Formats

Many of the terms and general concepts that have driven our PDF work can be applied to other formats. Potentially any file format that supports random access or relies on file offsets may have a Trust Chain.

For example, the authors also reviewed the ICC [11] and iccMAX [10] binary file format that define color profiles. ICC profiles can be embedded within PDF, JPEG, TIFF and many other formats, but are also used independently in operating systems and with embedded devices for managing color in printers, displays, cameras, etc.

As shown in Fig. 10 the ICC file structure comprises a 128 byte Header, followed by a Tag Table containing byte offsets to various Tag Element Data sections later in the file. The Tag Table itself comprises a 4-byte Tag Count followed by tag records, with one record for tag. Each record contains 3 4-byte fields: the Tag itself, a file offset to the Tag element data, and the size of the Tag element data (in bytes). The ICC specifications permit that Tag element data can be reused, so multiple tags in the Tag Table can refer to the same file location. The ICC specification also requires that tags start on

4-byte boundaries which means that up to 3 bytes of padding may be required between Tag element data segments.

However when multiple ICC parser implementations were tested, many of these file structure requirements were not being enforced allowing the authors to create fully functional ICC/PDF polyglots without detection.

As a result of our work and that of other researchers in the DARPA-funded "SafeDocs" program, several improvement suggestions have been provided to the ICC.

VI. Implementing The Specification

Ideally, the progression to implementation would have been achieved by (1) fully comprehending the Standard, (2) finding all problems with the Standard and clarifying them, (3) writing a formal specification of a PDF parser, then finally (4) implementing a PDF parser. In practice, the progression was the following: (1) we implemented a basic PDF parser; (2) we added features one by one (updating the design often); (3) we become aware of parts of the PDF Standard that were unclear and incorrect, for which a srictly correct implementation was not obvious; thus motivated (4) we wrote the pre-DOM specification; and finally, (5) we used the specification as a guide to re-writing some of the subtle parts of pre-DOM processing in our implementation. And concurrently with all of the above steps, we were both learning the PDF Standard and finding problems and ambiguities with it.

Our implementation consists of a full PDF 2.0 parser. It (safely) ignores Linearization data; when parsing hybrid-reference PDFs, it ignores conventional cross-reference tables provided for pre-1.5 readers. It processes but does not validate signatures (as incremental updates).

The primitive parsers are defined in the DaeDaLus [9] language. Other pre-DOM processing is implemented in Haskell. The implementation could be viewed as a relatively complete reference implementation of PDF. However, in the implementation, the clarity of the code is hampered by various software engineering requirements: creating better error messages, handling other PDF complexities such as encryption, etc.

It successfully parses a large number of PDFs in the wild (a high percentage of which PDFs diverge from the standard in some manner). Our implementation in most cases discovers more PDF errors than most PDF tools and readers.

We think, in future work, it would be valuable to use our specification more directly in our implementation (which is currently closer to the *D* implementation); see Sec. V-A above for why this would be desirable.

VII. Related Work

The PDF Association maintains the primary effort to specify the PDF format, which is done informally in the PDF standard ISO 32000, and has now been backed partially be a machine-readable definition of the Document Object Model [33]. The Caradoc project aimed to define a machine-readable specification of PDF with a verified parser, using the Coq interactive theorem prover [8]. Caradoc demonstrated that a small—but meaningful—core subset could be formalized in Coq; however its definition does not include a complete definition of many of the key features used to define referential context, including cross-reference streams and incremental updates.

A wealth of recent work has discovered methods (named *Shadow Attacks*) for subverting the end-to-end guarantees of document integrity that are provided by digital signatures in PDF [20, 22, 23, 24, 34]. Many of these attacks rely on mutating the referential structure of a document by updating its cross reference data via incremental updates. These attacks published to date only perform basic updates to the DOM, and do not rely on nuanced features or semantics of cross-reference streams and object streams. Recent work [13] has also introduced automated tools for determining if a given PDF document may likely be extended to form a Shadow Attack, which perform an analysis of a given document's DOM to identify patterns that indicate that the document has likely been crafted in order to perform a Shadow Attack. The search for such patterns does not involve a rigorous definition of data structures used to refer to objects in the DOM, which is provided by our approach.

Parser combinators are a well-studied abstraction for constructing parsers programmatically and are available as libraries for several industrial-strength languages, including the `parsec` library [16] originally implemented for Haskell and ported to go, F#, C#, and Java; the `nom` parser combinator library in Rust can be used to parse messages with *zero copies* [7]. Several experimental data-description languages, such as Parsley [21] and hammer [4]. Many of these libraries provide constructs for explicitly capturing and setting input streams; thus, we expect that many of them could be used to define the aspects of the PDF related to referential context that we have explored in this work. We believe that this and related aspects of the PDF definition will provide compelling motivating case studies in the design of these languages that will drive studies of how they can used to define "trust chain" features precisely and completely.

VIII. Conclusion

Data formats that define a document object model often rely on a rich notion of referential context: data structures that identify data objects, which are themselves used by data objects to refer to each other. Because referential context is itself is a rich data format, there is a natural motivation to reuse powerful abstractions defined in the format for organizing data to define the structure of context itself; done imprecisely, this invites circular definitions and ambiguities, along with incorrect implementations in practical processors.

We have studied in detail critical aspect of PDF's definition of referential context using cross-reference data. Our work has produced a novel machine-readable definition of the format's more complicated features for defining cross-reference streams, namely incremental updates and cross-reference streams, structured as a well-defined sequence of parsing passes over increasingly rich semantic values. Our work has resulted in multiple open issues concerning cross-reference streams submitted to the *International Organization*

for Standardization (ISO). We believe that it will be possible to instantiate the structure of our proposed definition to define and parse other complex formats that themselves use similar features, namely data offsets and lengths. We believe that such definitions will serve as a foundation for detecting general problematic structures that may occur in PDF documents and may occur in such formats as well, including file cavities.

In the future, we hope to extend our current definition to give a complete, machine-readable definition of all PDF features that define referential structure, including cross-reference streams in full detail, Linearized PDF, and hybrid-reference PDF files. We plan to use such a definition, and its associated parsers, to **(1)** deeply inspect security-relevant features of PDF documents, including freedom from file dead objects and file cavities; **(2)** rigorously develop implementations that expose document updates so that a document user is not lead to misinterpret what document content is attested by a digital signature; and **(3)** develop complete validators of higher-level document structure that manifests in the DOM, and which depends directly on the integrity of referential structures.

Acknowledgments

This research was supported by DARPA awards HR001119C0073 and HR001119C0079.

References

[1] Supriya Adhatarao and Cédric Lauradoux. How are PDF files published in the Scientific Community? In *2021 IEEE International Workshop on Information Forensics and Security (WIFS)*, pages 1–6, Montpellier, France, December 2021. IEEE. ISSN: 2157-4774.

[2] Andrew Mangle and Farida Keter. Analysis of machine learning techniques for detecting malicious PDF files using WEKA. *Journal of Business, Economics and Technology - Spring 2021*, 24(1):64–71, May 2021.

[3] The PDF Assocication. CVE-2022-25641, February 2022. https://github.com/pdf-association/safedocs/tree/main/Miscellaneous%20Targeted%20Test%20PDFs#dual-startxrefpdf.

[4] SERGEY Bratus, LARS Hermerschmidt, SVEN M Hallberg, MICHAEL E Locasto, FALCON D Momot, MEREDITH L PATTERSON, and ANNA SHUBINA. Curing the vulnerable parser. *usenix. org*, 2017.

[5] MITRE Corp. CVE-2020-9842, February 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-9842.

[6] MITRE Corp. CVE-2022-25641, February 2022. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-25641.

[7] Geoffroy Couprie. Nom, a byte oriented, streaming, zero copy, parser combinators library in rust. In *2015 IEEE Security and Privacy Workshops*, pages 142–148. IEEE, 2015.

[8] G. Endignoux, O. Levillain, and J. Migeon. Caradoc: A Pragmatic Approach to PDF Parsing and Validation. In *2016 IEEE Security and Privacy Workshops (SPW)*, pages 126–139. IEEE, May 2016.

[9] Galois, Inc. Galoisinc / daedalus, February 2022. https://github.com/GaloisInc/daedalus.

[10] ICC. Specification ICC.2:2019 (Profile version 5.0.0 - iccMAX) Image technology color management - Extensions to architecture, profile format and data structure [REVISION of ICC.2:2018], 2019.

[11] ISO TC 130 JWG 7. ISO 15076-1:2010 Image technology colour management - Architecture, profile format and data structure - Part 1: Based on ICC.1:2010, December 2010. Identical to ICC.1:2010 (Profile version 4.3.0.0) published by ICC. Japanese document X9207 is a translation of ISO 15076-1:2010. ISO publications are licensed and are not freely available.

[12] ISO TC 171 SC 2 WG 8. *ISO 32000-2:2020 Document management - Portable Document Format - Part 2: PDF 2.0*, volume 2 of *ISO 32000*. ISO, December 2020.

[13] iText. Investigating PDF shadow attacks: In-depth PDF security using iText (part 3), February 2022. https://itextpdf.com/en/blog/technical-notes/investigating-pdf-shadow-attacks-depth-pdf-security-using-itext-part-3.

[14] M. Iwamoto, S. Oshima, and T. Nakashima. A Study of Malicious PDF Detection Technique. In *2016 10th International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 197–203, July 2016.

[15] Simon Peyton Jones. *Haskell 98 language and libraries: the revised report*. Cambridge University Press, 2003.

[16] Daan Leijen and Erik Meijer. Parsec: A practical parser library. *Electronic Notes in Theoretical Computer Science*, 41(1):1–20, 2001.

[17] D. Liu, H. Wang, and A. Stavrou. Detecting Malicious Javascript in PDF through Document Instrumentation. In *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 100–111, June 2014.

[18] Christian Mainka, Vladislav Mladenov, and Simon Rohlmann. Shadow Attacks: Hiding and Replacing Content in Signed PDFs. *Network and Distributed Systems Security (NDSS) Symposium 2021*, 2021:17, February 2021.

[19] Tajuddin Manhar Mohammed, Lakshmanan Nataraj, Satish Chikkagoudar, Shivkumar Chandrasekaran, and B. S. Manjunath. HAPSSA: Holistic Approach to PDF Malware Detection Using Signal and Statistical Analysis. Technical report, IEEE, San Diego, CA, USA, November 2021. Publication Title: arXiv e-prints ADS Bibcode: 2021arXiv211104703M https://ieeexplore.ieee.org/document/9653097.

[20] Vladislav Mladenov, Christian Mainka, Karsten Meyer zu Selhausen, Martin Grothe, and Jörg Schwenk. 1 Trillion Dollar Refund: How To Spoof PDF Signatures. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1–14, London, United Kingdom, November 2019.

Association for Computing Machinery.

[21] Prashanth Mundkur, Linda Briesemeister, Natarajan Shankar, Prashant Anantharaman, Sameed Ali, Zephyr Lucas, and Sean Smith. Research Report: The Parsley Data Format Definition Language. In *LangSec 2020*, page 8. IEEE, May 2020.

[22] Jens Müller, Fabian Ising, Vladislav Mladenov, Christian Mainka, Sebastian Schinzel, and Jörg Schwenk. Practical Decryption exFiltration: Breaking PDF Encryption. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 15–29, London, United Kingdom, November 2019. Association for Computing Machinery.

[23] Jens Müller, Dominik Noss, Christian Mainka, Vladislav Mladenov, and Jörg Schwenk. Processing Dangerous Paths. *Network and Distributed Systems Security (NDSS) Symposium 2021*, page 16, February 2021.

[24] NDSS Symposium. NDSS 2021 Shadow Attacks: Hiding and Replacing Content in Signed PDFs, February 2021.

[25] PDF Association. Add explicit file format requirements for incremental updates on signed files?, December 2021. https://github.com/pdf-association/pdf-issues/issues/132.

[26] PDF Association. Clarification on signature field lock dictionary permissions and dss/dts incremental updates, November 2021. https://github.com/pdf-association/pdf-issues/issues/131.

[27] PDF Association. Hybrid-reference pdfs should be deprecated, August 2021. https://github.com/pdf-association/pdf-issues/issues/115.

[28] PDF Association. Mandated indirect reference requirements not followed and not enforced - are they really requirements?, July 2021. https://github.com/pdf-association/pdf-issues/issues/106.

[29] PDF Association. Clarification on xref sub-section line syntax (clause 7.5.4), February 2022. https://github.com/pdf-association/pdf-issues/issues/147.

[30] PDF Association. The semantics for use of table 16 objstm /extends key are unclear / insufficient, February 2022. https://github.com/pdf-association/pdf-issues/issues/148.

[31] PDF Association. Table 15 trailer size entry definition is described inaccurately/ambiguously, February 2022. https://github.com/pdf-association/pdf-issues/issues/149.

[32] PDF Association. Xrefstm key (table 19) description is confusing / wrong, February 2022. https://github.com/pdf-association/pdf-issues/issues/146.

[33] Peter Wyatt. The Arlington PDF Model, September 2021. https://youtu.be/ELAFymRYV30.

[34] Simon Rohlmann, Vladislav Mladenov, Christian Mainka, and Jorg Schwenk. Breaking the Specification: PDF Certification. *IEEE Security & Privacy*, (2021):17, May 2021.

[35] Charles Smutz and Angelos Stavrou. Malicious PDF Detection Using Metadata and Structural Features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, pages 239–248, New York, NY, USA, 2012. ACM. event-place: Orlando, Florida, USA.

[36] Jamie Willis, Nicolas Wu, and Matthew Pickering. Staged selective parser combinators. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–30, 2020.

[37] Y. Feng, B. Liu, X. Cui, C. Liu, X. Kang, and J. Su. A Systematic Method on PDF Privacy Leakage Issues. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1020–1029. IEEE, August 2018.

APPENDIX

A. Primitive Functions

```
-- misc primitives

validate :: Bool → P ()

validateAction :: P Bool → P ()

-- primitive parsers

readToPrimitive :: Offset → P String

seekPrimitive :: Offset → P ()

keyword :: String → P ()

(.|.) :: P a → P a → P a

parseString :: P a → ByteString → P a

-- higher level parsers

pTopLevelDef_UnDecStm :: P TopLevelDef_UnDecStm

pXrefRaw :: P (XRefRaw,Offset)

pSimpleWhiteSpace :: P String

pDictionary :: P Dict

pValue :: P PdfValue

-- complex parsers

findPDFHeader :: Int → P ((Int,Int),Offset)

findStartxrefThenParseToEOF :: Int → P Offset

-- various functions

decodeStream :: PdfValue → a → Offset → P ByteString

dictToTrailerDict :: Dict → P TrailerDict

trailerDict_getPrev :: TrailerDict → Maybe Offset

getKeyOfType :: Name → PdfType → Dict → P PdfValue

derefTLD :: Map ObjId (TopLevelDef' a :+: b) → ObjId →
P (TopLevelDef' a)

derefValue :: Map ObjId (TopLevelDef' a :+: b) → PdfValue → P PdfValue

updateVersionFromCatalogDict :: DOM → a → P a
```

```
extractDecodingParameters :: Dict → P a

removeFrees :: Map k XRefEntry → Map k (Offset :+: Type2Ref)

thawXRefEntry :: SEEK → XRefRaw → ObjId → P (ObjId, XRefEntry)

getObjIds :: XRefRaw → [ObjId]

unknownKeysInTrailerDict :: [Update] → Dict

-- various predicates for validation

versionAndDomConsistent :: (Int,Int) → DOM → Bool

trailersConsistentAcrossUpdates :: [Update] → Bool

verifyXrefRaw :: XRefRaw → Bool
```

B. Details of Streams and Type 2 References

```
getObjStm :: TopLevelDef' ByteString → P ObjStm
getObjStm (TLD_ObjStm x) = return x
getObjStm _              = error "expected ObjStm"
                           -- cannot recover from this

-- | extractStreamData - since we now know all Lengths:
--   - 1st, with the file offset, read into a bytestring
--   - 2nd, if an ObjStm, decodes/validates the stream
--     - NOTE: this processing done just once, not each time
--       that we "index" into the ObjStm.
extractStreamData :: Map ObjId (TopLevelDef' Offset :+: a)
                  → TopLevelDef' Offset
                  → P (TopLevelDef' ByteString)
extractStreamData _dom' (TLD_ObjStm _)    = error "unexpeced ObjStm"
extractStreamData _dom' (TLD_Value v)     = return $ TLD_Value v
extractStreamData dom' (TLD_Stream d off) =
  do
  len  ← getKeyOfType "Length" T_Int d  -- indirect OR direct
  len' ← derefValue dom' len            -- now an integer direct
  decodingParameters
       ← extractDecodingParameters d
  bs   ← decodeStream len' decodingParameters off
  return $ TLD_Stream d bs

derefType2Ref :: Map ObjId (TopLevelDef :+: Type2Ref)
              → Type2Ref
              → P TopLevelDef
derefType2Ref dom' (Type2Ref oi j) =
  do
  tld       ← derefTLD dom' (oi,0)
  ObjStm ss ← getObjStm tld      -- make sure the object is ObjStm
  s         ← case safeIndex ss j of
                Just s  → return s
                Nothing → error "bad type2 ref: index out of bounds"
  v ← parseString pValue s
       -- note that streams cannot be inside ObjStm
  return $ TLD_Value v
```