Accessible Formal Methods for Verified Parser Development

Letitia W. Li, Greg Eakman
FAST Labs, BAE Systems
Burlington, Massachusetts, USA
letitia.li@baesystems.com, gte@ieee.org

Elias J. M. Garcia, Sam Atman
Special Circumstances
Brussels, Belgium
{ejmg, sam}@special-circumstanc.es

***Abstract*—Security flaws in Portable Document Format (PDF) readers may allow PDFs to conceal malware, exfiltrate information, and execute malicious code. For a PDF reader to identify these flawed PDFs requires first parsing the document syntactically, and then analyzing the semantic properties or structure of the parse result. This paper presents a language-theoretic and developer-accessible approach to PDF parsing and validation using ACL2, a formal methods language and theorem prover. We develop two related components, a PDF syntax parser and a semantic validator. The ACL2-based parser, verified using the theorem prover, supports verification of a high-performance equivalent of itself or an existing PDF parser. The validator then extends the existing parser by checking the semantic properties in the parse result. The structure of the parser and semantic rules are defined by the PDF grammar and fall into certain patterns, and therefore can be automatically composed from a set of verified base functions.**

Instead of requiring a developer to be familiar with formal methods and to manually write ACL2 code, we use Tower, our modular metalanguage, to generate verified ACL2 functions and proofs and the equivalent C code to analyze semantic properties, which can then be integrated into our verified parser or an existing parser to support PDF validation.

***Keywords*—LangSec, PDF, Formal Methods, ACL2**

I. Introduction

Portable Document Format (PDF) Readers are a known target for attacks with multiple known vulnerabilities [1]. Malware in executable format can masquerade as PDFs, or PDFs can execute malicious code [2]. Circular references can cause certain PDF readers to crash [3] or display infinite recursion [2]. PDFs also include actions such code execution, providing additional avenues of attack to run malicious scripts or use actions to exfiltrate data from an encrypted PDF [1], [4], [5].

A PDF reader's parser should detect if the input is malicious or invalid [6], which can occur via syntax (form of the input language) or semantics (meaning of terms within the input). For example, an invalid sequence of characters is a syntax error (such as parsing a PDF without the "%PDF-<version>" header), while structures violating the PDF standard are semantic errors [7].

Semantic properties from the PDF standard determine if a document is readable, safe and acceptable to all readers (e.g. there are no cycles in certain fields). Syntax-level parsing, however, cannot check certain semantic properties in the process of parsing. During syntax parsing, objects referenced in the field of another object may not have been yet parsed, and therefore we cannot check any of the attributes of the referenced object (such as its type, or even its existence). Instead, these semantic properties are checked after the first parsing of the document, whether within the parser or separately.

In the DARPA SafeDocs project, we address the insecurity of PDF readers/parsers with a Language-theoretic Security approach by first defining a PDF grammar, and then developing verified parsers to accept only well-formed input according to that grammar [8]. In this paper, we propose an accessible formal methods approach in ACL2 for safe, secure, and efficient parser development covering both syntax and semantics, demonstrated on the PDF standard. While other parsers focus on mainly syntax parsing, we add semantic validation intended to fully cover the PDF standard.

Formal methods offer mathematical proofs of correctness of software and more rigorous and complete coverage than manual testing [9]. These techniques help detect and remove errors by providing unambiguous statements of correctness to verify on the system [10]. Theorem-proving languages, like ACL2, involve the verifier writing functions and then corresponding theorems to prove guarantees on the output and correctness of those functions [11]. ACL2 uses first-order logic, and is based off a subset of Common Lisp [11]. It has been used for the verification of real-world systems such as microprocessors, algorithms, and virtual machines [12], [13], [14].

However, formal languages like ACL2 are not commonly used for final product development nor designed for performance. Other formal languages, such as Coq, can sometimes extract performant code automatically, while we write a verified equivalent to an existing syntax parser written in a language more accessible to developers, such as the Hammer-based parser in C [15]. If we can demonstrate the equivalence of the operational and verified parsers, guarantees of our verified parser can then extend to the operational parser. Next, as the operational parser is not written in a formal language, integrating the semantic checking of the verified parser requires expressing the same, equivalent functions in the language of the operational parser. We apply the known technique of generating equivalent proven and unproven (but easier to use and generally faster) functions to assure correctness of functions written in a non-formal/theorem proving

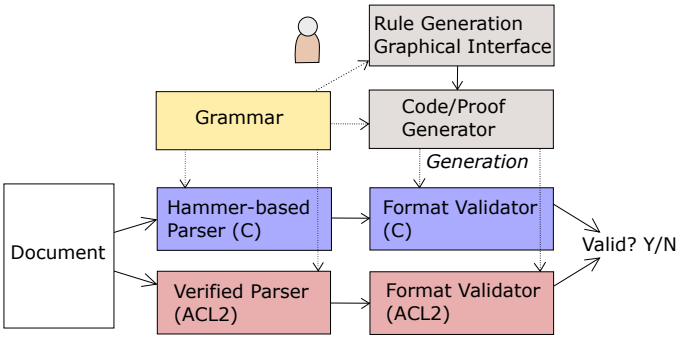

Fig. 1. Document Validation Overview

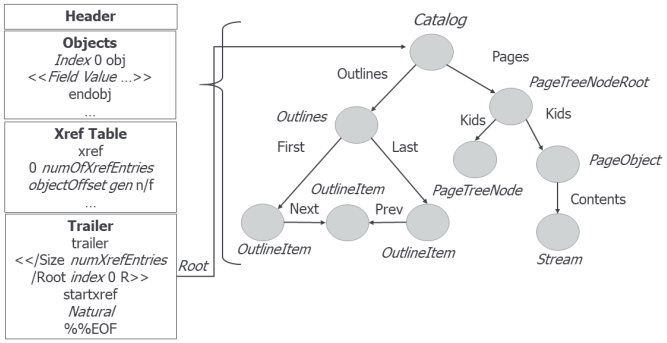

Fig. 2. Structure of the PDF document with object structure

language [16].

To apply verified semantic checks, we generate verified functions based on the language specification to analyze the parsed objects. However, the specification expert or developer may not be experienced in formal methods, which is widely accepted to be difficult to learn to comprehend or use [17]. Not all developers are trained in the mathematics of formal methods, which is another barrier to its widespread usage [10]. To make formal methods more accessible, we developed an automated generator for ACL2 and C which allows users to easily write rules using a graphical interface, and an S-expression metalanguage named Tower. The metalanguage separates the user from the underlying formal proofs, so that users are not required to understand the details of the theorem prover.

Figure 1 shows the overview of our approach, where we first parse a document with equivalent verified and operational syntax parsers, and then use the graphical interface to generate semantic validation rules, which forms the "Format validator". Currently, we use a Hammer-based PDF parser for syntax parsing, which provides us with the parse result in the form of an Abstract Syntax Tree (AST) [15]. We then transform and cast the AST into a list of PDF objects (as described in the next section), which is then passed into the format validator.

The grammar of the format determines the syntax parser, semantic structure of parsed objects, and the layout of the graphical interface, as we will explain in this paper. The current ACL2-based verified parser is a proof-of-concept verified parser that can be extended into the equivalent of the Hammer-based PDF parser or another existing parser. Once we've demonstrated their equivalence, the verified parser assures the correctness of the operational parser, while we mainly use the operational C parser for actual document analysis. The advantage of our approach is that we only require the reviewer check the translation for helper functions and single expression code translations, instead of longer functions with multiple inner expressions. The proofs of correctness and other properties assured on the ACL2 functions then apply to the generated C function.

We begin by introducing the PDF structure in Section 2, which explains the source and structure of a PDF. Section 3 discusses PDF grammar on which we base our syntax parsing and semantic validation, and Section 4 applies the grammar to developing a verified syntax parser in ACL2. Section 5 discusses the types of rules used for semantic validation of PDFs, and Section 6 describes how we generate ACL2 functions and theorems of correctness to apply those rules on the parsed PDF. Section 7 discusses related work in PDF parsing and validation, as well as other parsers. Finally, Section 8 concludes the paper and offers directions for future work.

II. PDF Structure

PDF Documents are composed of a header with the PDF version, a body of objects (pages, fonts, etc.), a cross-reference table listing objects, and the trailer [18]. The header nominally dictates which version the PDF document may conform to while the objects in the body of the PDF specify the actual design elements and text. The cross-reference table, called an xref table, lists the location of both used and unused objects by their offsets from the start of the file. The trailer includes the location of the last xref table and other special objects, such as a reference to of a PDF's root object. The root object, known as the Catalog, contains references to child objects in the document hierarchy.

Listing 1
ACL2 type definition for the PDF object

```
(fty::defmap attrdict
  :key-type stringp
  :val-type any-p)

(fty::defprod obj
   ((type stringp :default "")
   (attrs attrdict-p :default (list nil))
   ))
```

Figure 2 shows the structure of a simple PDF, where the Catalog is the root object, and it references Outlines and PageTreeNodeRoot objects. The Outlines object in turn references individual OutlineItems, while the PageTreeNodeRoot contains the page tree of either PageTreeNodes or individual pages. The text content on a page is then contained in a stream object. Every major structure in PDF, including the xref

table and trailer, is constructed from the eight built-in types of objects: boolean values, integer and real numbers, strings, names, arrays, dictionaries, streams, and the null object [7].

Once parsed, the objects in use within the PDF form the tree of semantic "PDF objects". PDF objects consist of an object number, type, and an attribute dictionary containing key:value pairs to indicate properties such as length, parent object, etc. Listing 1 shows the ACL2-definition for the PDF object type.

III. PDF Grammar

A first attempt at parsing could be to parse from the start of the document, and parse a list of objects by searching for the "⟨integer⟩ 0 obj" tag and ending with the "endobj" tag. However, this parse would fail if there were "endobj" or "⟨integer⟩ 0 obj" strings within a stream and would still require that we check offsets against the xref table to determine if the offsets are correct and which parsed objects were in use. In addition, looking for the "xref" tag as the start of the xref table may not be accurate if that string is also present in stream text.

Listing 2
PDF Grammar

```
Pdf := Header Objects Xref Trailer ;
Header := "%PDF-" integer "." integer ;
Trailer := "trailer" newline "<</Size
   integer /Root integer 0 R>>" newline
   "startxref" newline XrefOffset "%%EOF"
        → peg/match (XrefOffset) from
      file start, apply rule "Xref";
Xref := "xref" newline "0" integer
   XrefEntry* ;
XrefEntry := UsedXref | NotUsedXref ;
UsedXref := Offset Generation "n" → peg
   /match (offset) from file start, apply
    rule "IndirObj" ;
NotUsedXref := Offset Generation "f" ;
Offset := integer ;
Generation := integer ;
Objects := IndirObj* ;
IndirObj := integer "0" "obj" newline Obj
    newline "endobj" ;
Obj := ObjVal | StreamObj ;
ObjVal := "<<" AttributeDict ">>" ;
StreamObj := "<<" AttributeDict ">>"
   StreamContents ;
StreamContents := "stream" linebreak char
   * "endstream" ;
AttributeDict := (Field Value)* ;
Value := String | IntegerList | Integer |
    RefList | Ref | AttributeDict | ... ;
Ref := integer "0" "R" ;
```

Due to the combined concerns of both syntax and semantic evaluation that occur in the process of parsing a PDF file, a context-free parsing is not assured to be correct. Context-free grammars like Augmented Backus–Naur form (ABNF) cannot express the jump to offsets required in PDFs, which can instead be described with Parsing Expression Grammar rules [19].

Listing 2 shows our PDF grammar: an ABNF grammar supplemented with Parsing Expression Grammar rules for jumping to certain bytes. We start parsing from the unambiguous trailer, containing the byte offset of the xref table. Then, we parse the xref table for the header and entries containing offsets to PDF object locations. Finally, PDF indirect objects contain "⟨object integer⟩ 0 obj", then the attribute dictionary with field:value pairs where the value can be multiple different types, including lists of objects. For stream objects, the character stream is found after the attribute dictionary.

IV. Verified Syntax Parsing

A language-theoretic security approach requires verifying that parsers accept only well-constructed input according to the grammar [8]. We compose our parser from verified functions, and then, using ACL2, we develop proofs of soundness and completeness: that our verified functions rejects malformed input and accepts all valid input.

The verified parser structure, shown in Figure 3, implements the grammar from Listing 2, first parsing the trailer to find the xref section, then parsing the xref section to find objects. This parser focuses on parsing the objects to build the AST for the validator, assumes the PDF matches the grammar, and ignores the header check. The parser is built off of sequential executions of parsing strings, numbers, and jumping to offsets. Certain values contain semantic meaning to be used in the following parse steps, such as offset values. Others, such as the object number, are used to instantiate the semantic PDF object. In future work, we could build another parser to first verify that the PDF matches the grammar in Listing 2.

Figure 4 shows how an example xref table would be parsed. The parser first checks that the xref table starts with "xref" tag, then finds the number of entries, and for the xref entries referencing objects in use, parses the offset of the start of the object. A failed parse on any of the steps returns an empty list or NIL as the remaining text, resulting in an empty list of xrefs. While not directly used in parsing, the parsed number "number of entries" is checked against the final number of xref entries as a semantic property. In this implementation, objects not referenced in the xref table will not be parsed.

The verified functions shown are constructed from built-in and verified ACL2 functions, like "search" to find the index of a substring. The search function allows for matching to the first instance or the last instance of a substring. For example, Listing 3 shows the ACL2 code for the top level function for parsing the list of semantic objects from a file "filename", returning a list of objects and the state as output. The text preceded by ';' are comments. While ACL2 provides multiple file reading functions, we read the file as a list of characters in order to jump to offsets. In addition, ACL2 uses a single-threaded object 'state' for reading files, of which there can be only 1 copy, and every function with state as an input must also have it as an output. This single-threaded object is intended

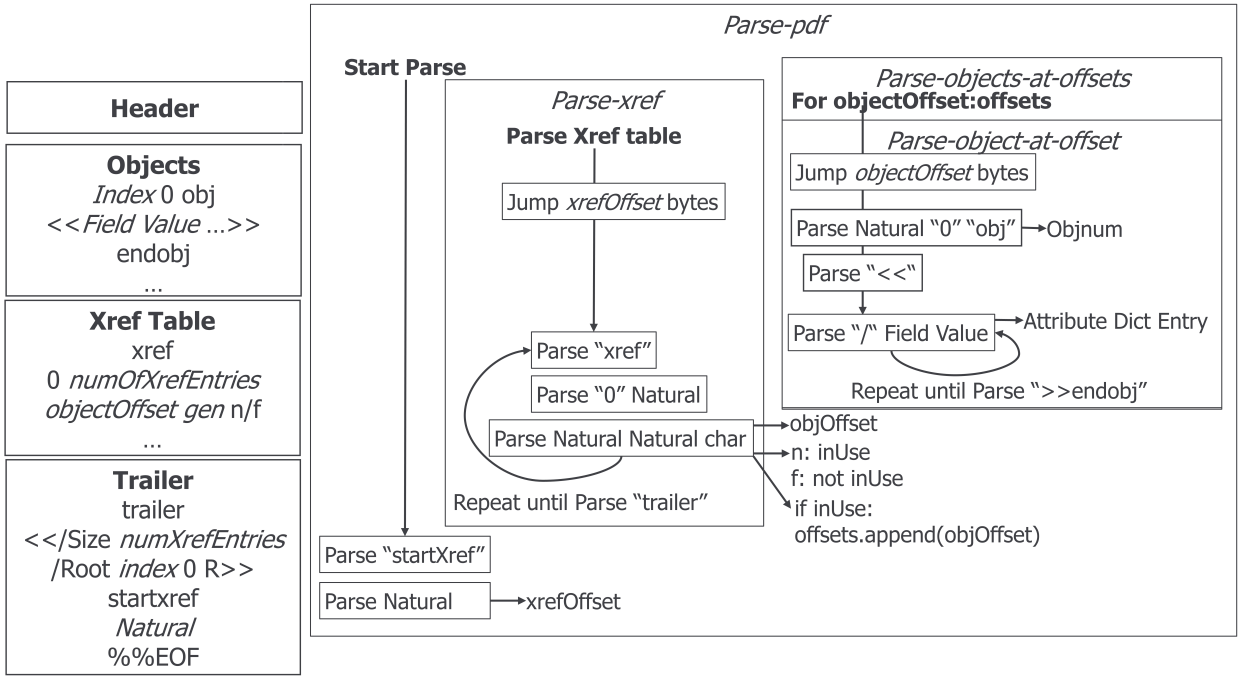

Fig. 3. PDF Syntax Parsing Overview

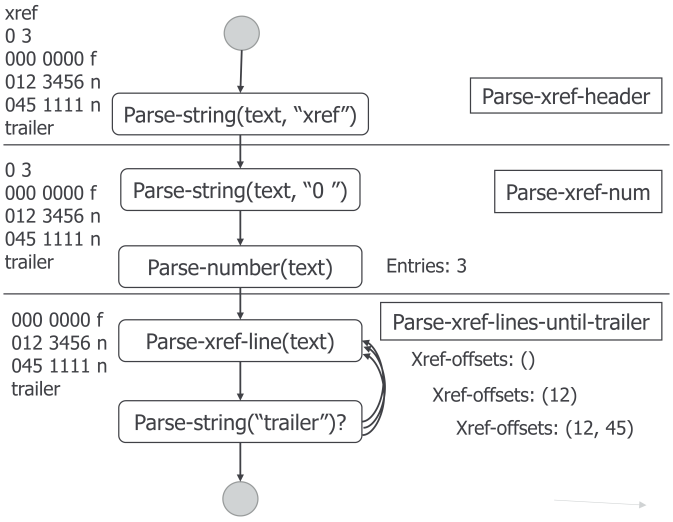

Fig. 4. Xref section parsing

for better performance and solves concurrency issues [20]. To access and operate on the content read from a file, we need to use the *(mv-let (chars state) (read-file-characters filename state) code)* format, where within "code", we can process the character list "chars".

The verified parser starts by finding the xref table location, where we find the last "startxref" string, then parse the xref offset as the number between "startxref" and the ending "%%". The xref section itself is then the characters between the xref offset and the start of the trailer, which is then parsed into a list of offsets, which is then in turn passed into "parse-all-objects-at-offsets" function, which then parses the object starting at each offset into an object number and attribute dictionary.

Listing 3
ACL2 CODE FOR PDF PARSING

```
(defun parse-pdf (filename state)
 (mv-let (chars state)
  (read-file-characters filename STATE)
  (let* ((start (search (explode "
     startxref") chars :from-end t)) ;
     Find offset of startxref"
   (end (search (list #\% #\%) chars :
      from-end t)) ;Find offset of end of
       file tag
   (xref (parse-number (char-ranges chars
       (+ start 10)(- end 1)))) ;The xref
       table location is the character
      list between "startxref" and "\%\%"
       which should be parseable as a
      number|
   (offsets (parse-xref (char-ranges
      chars xref start))))
   )
   (value (parse-all-objects-at-offsets
      chars offsets xref)  )
 ) ) )
```

By admittance into the ACL2 theorem prover, functions are already guaranteed to terminate. As following object references is known to potentially not terminate in case of cyclic references [24], we first do not follow object references

in syntax checking, and instead parse the object reference's integer without checking that reference exists.

In the next section, we describe how we ensure termination of functions when checking object references. ACL2 theorems then assure the correctness of individual functions, such as parse-number returning NIL if the input contains any non-digit characters, and that parse-number returns a number for all valid input, or lists of only numeric digits. We also use the ACL2sedan extension to define functions with input contracts and output contracts to restrict the allowable types of inputs and guarantee certain properties about outputs [21].

V. Semantic Validation

After syntax parsing, we cast the parsed strings to form PDF objects as previously shown in Listing 3. We can then apply semantic rules to check individual objects or check the structure of the ensemble of objects. The International Organization for Standardization defines the official PDF specification in the ISO 32000-2 document [7]. It is a hand-written 700+ page document that uses natural language with some structure indicated through the use of formatting and tables. It provides rules on required fields of objects, types of objects which can be referenced, types of circular references that render a PDF invalid, etc [7]. The PDF Association, as part of SafeDocs, has recently created a specification-derived machine-readable definition of PDF 2.0 known as the "Arlington PDF DOM" [22]. This machine-readable definition used a semi-automated method to extract data from the tables in the ISO 32000-2, and was then supplemented with manual correction. Checking a document against all the rules is a difficult manual process, and by using the PDF DOM, we propose to automate the process.

Table I shows an extract of the PDF specification for the "PageTreeNode" Object, which contains multiple rules. For example, all the 4 keys (which we refer to as fields or attributes in this paper) are required, meaning that every PageTreeNode object must contain these fields. The Possible Values Column then indicates allowed values for certain fields. For example, the Type field must contain a value "Pages", and Count cannot be negative. The Link field then specifies the allowed types of referenced objects in this field. For example, the Parent of a PageTreeNode must be either a PageTreeNode or PageTreeNodeRoot.

TABLE I
PDF standard rules for the PageTreeNode Object

Key	Type	Required	PossibleValues	Link
Type	name	TRUE	[Pages]	
Parent	dictionary	TRUE		[PageTreeNode, PageTreeNodeRoot]
Kids	array	TRUE		[ArrayOfPageTreeNodeKids]
Count	integer	TRUE	[>0)]	

Violations of the specs can cause various errors in the PDF reader, including cyber exploits or unintended behavior such as error messages, a blank document, or a document displayed differently than desired. For example, if the Kids field did not point to a Page Tree or Page, then some readers would display blank pages and Adobe Reader would display an error message. Missing required fields, such as Count, can also cause errors, where regardless of the number of PageObjects referenced, the document will only contain a single page. As provided by the PDF specification rule tables above, we currently express 4 types of rules:

- Required field rules: If a field is indicated as "Required" in the specification for that object type. For example, OutlineItem objects must have a Parent field.
- Referenced object type rules: The type of referenced object must be of a certain type, as indicated in the Link Field. For example, the 'Prev' field of an OutlineItem must be another OutlineItem
- Value type rules: the value of the field may be fixed to certain types. For example, Count must be a positive integer, or it must be a reference to an object which is an integer.
- Cycle rules: Circular references on certain fields, such as /Kids in a PageTree and /Next in an outline may cause errors for the reader.

These rules cover the rules on all types of PDF objects, and in the future, we will expand to expressing rules with conditions, such as rules which only apply to certain PDF versions. Instead of manually writing functions and theorems for each of these rules, we introduce the Tower Metalanguage for ACL2 function and theorem and C function generation. Our metalanguage provides an easy way to capture and encode the PDF rules for semantic validation. Tower meta-expressions are written as S-expressions, in the form (FuncName expression expression ...). This metalanguage is named Tower since it builds a top-level expression from building blocks of lower level expressions.

An example of a pseudocode expression and the corresponding Tower Translation is shown in Figure 5. This example expresses the rule that for all objects of type "PageTreeNode", all objects referenced by the /Kids field must be of type "PageObject" or "PageTreeNode", which we write as needing to be a member of the list ["PageObject", "PageTreeNode"]. We had to distinguish between "member" and "string-member" since ACL2's "member" function can determine if certain objects are contained within a list, but does not work on finding strings within a list of strings, and we had to define our own. In the next sections, we explain how each of these Tower expressions translate to ACL2 and C.

In addition, instead of writing an expression for each rule, we also use variables as arguments for groups of rules with identical structure but with different constants. For example, for the expression above, type==PageTreeNode and obj2.type in [PageObject, PageTreeNode] will be replaced with type==TypeVariable and obj2.type in ValueVariable respectively. Each value of TypeVariable and ValueVariable are then passed as arguments into the generated function.

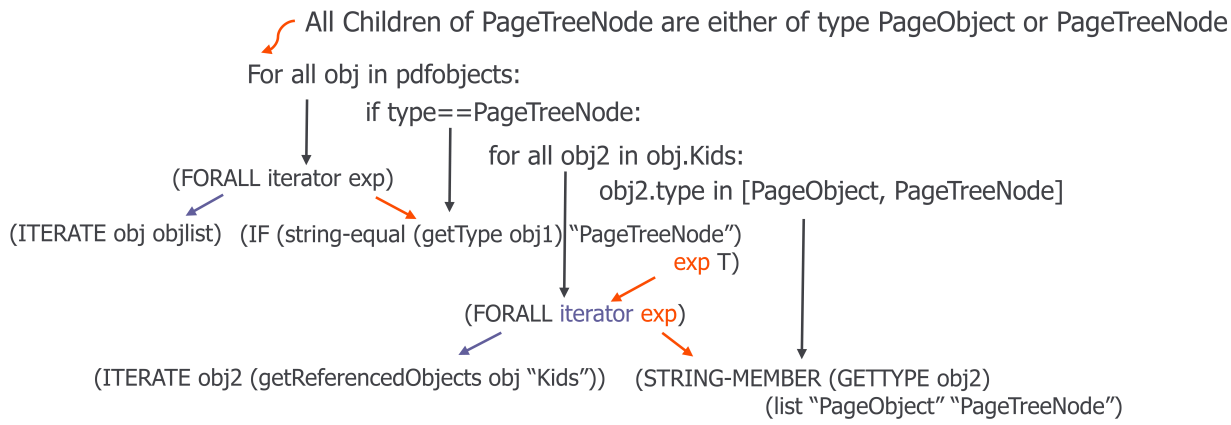

Fig. 5. Translation from pseudocode to Tower expressions for a rule ensuring all objects referenced by the "Kids" field of objects of type "PageTreeNode" are of type "PageObject" or "PageTreeNode"

Due to the modular nature of Tower, expressions are independent of each other and can be swapped out or replaced to create new rules. For example, if we wanted to check that objects referenced had a non-zero object number, we'd replace the String-member expression with (< (getField "Objnum" obj2) 0) expression, while leaving the rest of the expressions unchanged.

Tower expressions are encoded and stored in XML, but a GUI provides ease of use and hides the use of ACL2 from the developer. The GUI displays the rules in a structure more similar to a PDF object structure and provides a better visualization of semantic expressions.

Listing 4
SEMANTIC PDF OBJECT STRUCTURE

```
PDFObject := Type AttributeDict ;
Type := "Catalog" | "PageTreeNode" | "
   PageObject" | "FontTrueType" ...;
AttributeDict := ("Objnum" ObjectNumber)
   (Field Value)* ;
Field := "Kids" | "Count" | "Length" | "
   Subtype" ... ;
Value := String | IntegerList | Integer |
    Ref |  AttributeDict ... ;
Ref := ObjectNumber 0 R ;
ObjectNumber := Natural ;
```

The PDF semantic structure can be expressed as a list of PDF objects, where each object has a string type and attribute dictionary as shown in Listing 4. Each entry of the attribute dictionary is a field:value pair, where the value can be a list, string integer, etc., or reference to another object. Some rules, as we previously described, check first that the referenced object exists, and then check properties of the referenced object (such as its type) instead of the value itself.

As described in the previous section, ACL2 functions must terminate to be admitted into the proof assistant. As cycle-checking functions follow object references will not terminate

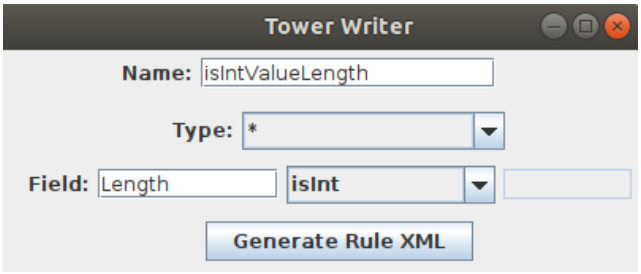

Fig. 6. The Rule generation GUI expressing the rule that the value of the Length field is always an integer

in case of cyclic references, we restrict the number of reference calls of such functions based on the length of the object list. In a PDF without cycles with N total objects, each object refers to the next object, and etc, then at most there can be N-1 references. If the number of expansions exceeds the length of the list, then a cycle must be present and the function terminates.

Based on the grammar, the user interface for a check on a field value is shown in Figure 6. The GUI's elements reflect the semantic PDF Object grammar with the Type and Field:Value inputs. In the first example, the GUI is filled in to check if the Length is always an integer value. From the top of the GUI, the developer first enters a base name for all functions generated. Then, he/she selects the type of object the rule applies to, or "*" if the rule applies to all objects. In this case, we want to check each object that has a Length field, so we do not restrict the check to a single type. Then, the developer fills in the expression, with the field, function name, and optional value. To check that the value in the field "Length", we only need the function "isInt" without a value. If we were checking that Length was instead greater than 0, for example, we would fill in ">" for the function and "0" for the value.

If instead, we wanted to check properties of the objects referenced by a field, like the type of objects referenced by the /Kids field of PageTreeNode objects to be either PageObject or

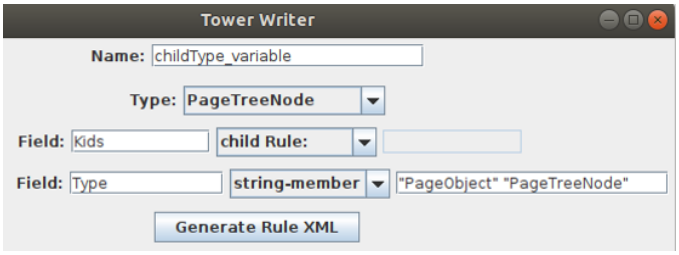

Fig. 7. The Rule Generation GUI expressing a rule on objects referenced in a field: that for all "PageTreeNode" objects, the objects referenced in its /Kids field must be of type "PageObject" or "PageTreeNode"

PageTreeNode as previously shown, we would set the function name to the "ChildRule" field as shown in Figure 7 The GUI then offers another line to indicate a rule on those referenced objects, such as the check that the type of the objects is in ["PageObject", "PageTreeNode"].

VI. ACL2 Function and Proof Generation

A Tower expression translates into ACL2 functions and theorems as well as an equivalent C function. If the generated functions in ACL2 and C are indeed equivalent, then for any input, both sets of functions will return the same output. ACL2 functions take arguments and return a value, without side-effects, or modifying input variables. Theorems express properties about a function, the theorem prover helps create the proof of those properties. Theorems are usually expressed in an "<condition> implies <conclusion>" format, where the conclusion indicates correctness or other properties on the output of the function.

Instead of coding the entire function in a single block of text, and then writing lemmas to help prove the correctness of the function, we generate ACL2 functions which only express a single expression (such as looping over objects, running a Boolean check, etc).

By generating smaller code snippets from Tower, as shown in Figure 8, we see that checking if the string arg1 is found in the string list arg2 varies if expressed in a functional vs imperative language, but we also can easily examine the Tower-to-C and Tower-to-ACL2 code translations and assure the equivalence of the two functions. For example, we could generate many objects with various types to pass as input into the functions expressed in ACL2 vs C, and test that the outputs are in fact equivalent.

These single-expression functions fit a pattern with a known proof technique. The theorems generated on correctness of a function vary depending on the type of expression, such as "if cond: output1, else output2", or iterations over a list of objects. Later in this section, we show an example of a proof of correctness on a function from our running example.

Expressions based on iterations over a list, such as the "ForAll" expression are proven by induction. The lowest level expression from Figure 5, checking if the type is within the list of allowed types, is a Boolean expression proved trivially by restating that if the function returns true, then the sub-expression (that the type of the checked object was within the list) must be true.

The next highest level expression is a "ForAll" expression translated into the "childTypeIterate" function which applies the "childTypeCheck" function to every element in a list. Figure 9 shows the Tower Expression, and then the pattern for applying a function to members of a list in ACL2, and the correctness of the function is proven by induction. The base case proves that the function is correct for an object list with 1 element, and the inductive step proves that if childTypeIterate returns true (T) for all elements besides the first element and childTypeCheck returns true for the first element, then childTypeIterate returns true on the full list, finally summarized by the conclusion that childTypeIterate should return true if and only if childTypeCheck returned true for each element of the list. ACL2 hints speed up theorem proving by telling the prover that the conclusion proof does not require expanding the childTypeCheck function into its implementation. As shown here, the childTypeCheck function only needs to know that it is passed an object list, and not how the object list was generated or the internals of the childTypeCheck function. The contents of that object list are determined by its Iterate expression and passed in from the call to childTypeCheck.

Similarly, Figure 10 shows the equivalent pattern and generated code for the same iteration in C. The comments in green explain how the function acquires a list of PDF objects to iterate on by passing a pointer to objlist into "getObjlist", which returns the size of the list. Then, the function iterates on each element in objlist and passes it to the child expression.

Listing 5
Evaluation PDF with Semantic Errors

```
%PDF-1.3
1 0 obj
<< /Pages 2 0 R /Type /Catalog >>
endobj
2 0 obj
<< /Count 10 /Kids [ 3 0 R 4 0 R 5 0 R 6
   0 R 7 0 R 8 0 R 9 0 R 10 0 R 11 0 R 12
    0 R ] /Type /Pages >>
endobj
3 0 obj
<< /Contents 13 0 R /CropBox [ 0 0 612
   792 ] /MediaBox [ 0 0 612 792 ] /
   Parent 2 0 R /Resources 14 0 R /Rotate
    90 /Type /StructTreeRoot >>
endobj
...
6 0 obj
<< /Contents 22 0 R /CropBox [ 0 0 612
   792 ] /MediaBox [ 0 0 612 792 ] /
   Parent 2 0 R /Resources 23 0 R /Rotate
    0 /Type /Pages >>
endobj
...
```

Tower Expression	ACL2 Translation	C Translation
(GETTYPE obj2)	(obj->type {obj})	getType(obj)

(STRING-MEMBER arg1 arg2)	(defun string-member (arg1 arg2)
	(if (> (len arg2) 0)
	(or (equal (car arg2) arg1)
	(string-member arg1 (cdr arg2)))
	NIL
	))
		char string[50];
		strcpy(string, arg2);
		char *ptr = strtok(string, arg2);
		while(ptr != NULL){
		if (strcmp(ptr, arg1)==0){
		return 1; }
		ptr = strtok(, " ");
		}
		return 0;

Fig. 8. Code snippets for checking a type of an object is "PageObject" or "PageTreeNode"

We tested our validator on a set of PDFs provided by the Safe-Docs program with known semantic errors. First, we parsed the PDF with a provided Hammer-based PDF parser, and then passed the AST into our validator. We identified 4 PDF with cycles, including the the example provided by Caradoc [2], and another with object reference errors. Listing 5 shows the erroneous object references, and while the known semantic error was a "PageTreeNode" object (object 6) without the required "Kids" field, our validator also detected that the same object was missing the required "Count" field. In addition, from our running example, any object referenced in the "Kids" field of an object of type "PageTreeNode" must be of type "Page" or also "PageTreeNode". However, in this example, the object 2's Kids field references object 3, which is declared to be of type "StructTreeRoot". In fact, closer examination of object 3 also shows that it is likely typed incorrectly, as referencing the PDF specification [7], it is missing all of the characteristic fields of a StructTreeRoot, and instead has all the fields of a Page Object. This semantic analysis thus detects additional PDF specification violations after syntax parsing.

VII. Related Work

Caradoc is a PDF parser and validator in OCaml which examines both the syntax and structure of PDFs [2]. Their work proposed a strict restricted safe specification for PDFs in terms of syntax and semantic properties. Their validator accepted documents only with valid typing of PDF objects, correct hierarchy of PDF objects, and presence of cyclic references, and proposed a strict restricted safe specification. However, their validator did not systematically generate and check rules of the entire PDF specification like our work.

The JHOVE formal validator tool also checks the syntax and structure of PDFs, including that the catalog object is present, and Catalog, page tree node, page node, font, and stream objects are "well-formed", in that they contain required fields and allowed values [23]. However, their work does not present a grammar or verified parser or validator.

[24] published a formal PDF parser written in Coq, with similar high-level parsing structure but with a different section

Tower Expression (FORALL iterator exp)

Function Pattern

```
(defun childTypeIterate (objlist)
(if (> (length objlist) 0)
    (and {childExp firstObj}
{exp restOfObjs})
    T ) )
```

Generated Function

```
(defun childTypeIterate (objlist)
(if (> (length objlist) 0)
    (and (childTypeCheck (car objlist))
(childTypeIterate (cdr objlist)))
    T  ))
```

Proof Pattern Proof by Induction

Generated Proof

Base Case

```
(defthm childTypeIterate-base
 (implies
  (childTypeIterate (list obj))
  (childTypeCheck obj)) )
```

Inductive Step

```
(defthm childTypeIterate-induct
 (implies
  (and (childTypeCheck (car objlist))
    (childTypeIterate (cdr objlist)))
  (childTypeIterate objlist)) )
```

Conclusion

```
(defthm childType_iterate-conc
 (implies
  (and (member obj objlist)
(childType_iterate objlist))
  (childTypeCheck obj))
:hints (("Goal"  :in-theory (disable
childTypeCheck) )))
```

Fig. 9. Tower Translation to ACL2 functions and proof for checking every object in a list

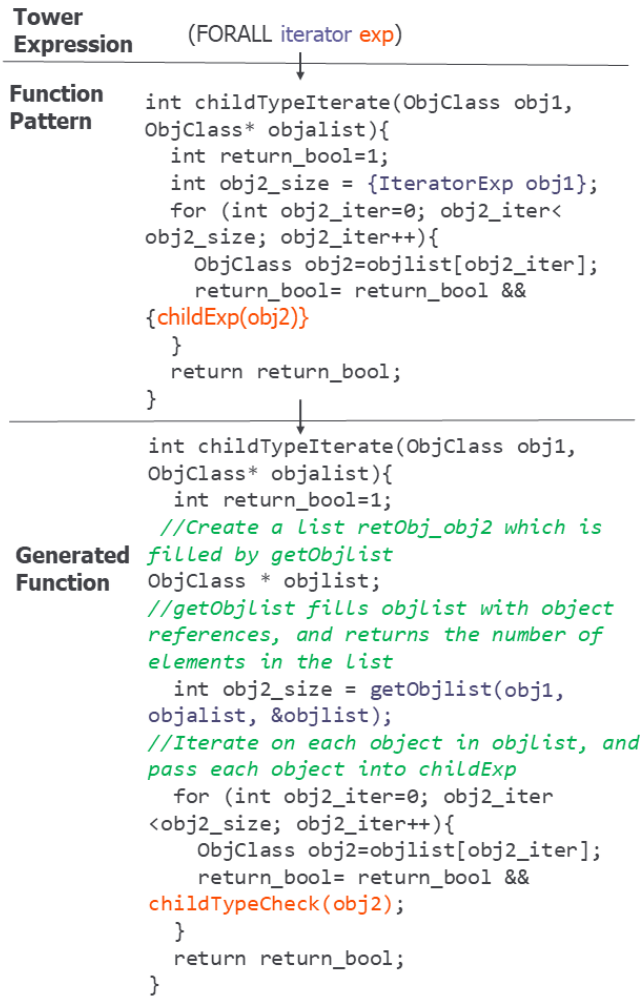

Fig. 10. Tower Translation to C functions for checking every object in a list

parsing approach since Coq uses a pattern matching format instead of the character-list processing like our parser. Their parser syntactically parsed PDFs and provided proof of termination. However, their work does not examine semantic properties from the PDF standard such as checking for required fields, etc.

The Parsley data format provides a formal grammar for reasoning on data formats, including on PDF indirect objects [25]. Besides checking that object numbers are integers greater than 0, they also perform semantic validation like object numbers are unique.

Some verified parser frameworks are general and can be applied to multiple grammars. The Kestrel ABNF parser in ACL2 parses a grammar and proposes some guarantees about the parse result even if the parser of text is not yet implemented [26]. It includes theorems on the soundness and completeness of the parsed grammar, and an unimplemented "parse" function which takes in a grammar and string, which is a direction of our future work. There are theorems on such as the parse function returning a result if the input text is parsable, or if only one parse tree is generated, then the input was unambiguous. However, this parser is still only theoretical.

The Coq-based Narcissus parser combinatory framework generates a verified correct encoder and decoder (parser) of a format, including guarantees that the encoder and decoder are inverses of each other [27]. As Narcissus uses Coq as the proof language, equivalent implementations of executable code in OCaml is extracted with the help of provided scripts.

VIII. Conclusion and Future Work

This paper presents a language-theoretic security approach to development of a secure PDF parser. We presented a verified parser and semantic validator in ACL2, developed based on a PDF grammar. ACL2 verification relies on user-defined theorems to prove correctness of functions, and composing complex functions from simple functions in set patterns provides us with known proof approaches easing the verification process. Through our development of the Tower Metalanguage and graphical interface, we provide developers unfamiliar with formal methods an accessible framework to write verified semantic functions and theorems in ACL2 and equivalent performant code in C.

Currently, the verified ACL2 parser was manually written with the grammar as a reference. The PDF parser is therefore still hard-coded, and responding to changes to the grammar requires manual modifications to the ACL2 parser and its proofs. In the future, we will apply the same Tower approach towards building verified functions for semantic checks to automatically generating parsers from a grammar. Furthermore, we will work towards extending our ACL2-based syntax parser to be functionally equivalent to a currently-used PDF parser such as Hammer, thus ensuring that the correctness properties we proved also apply to the operational parser.

The current ACL2 syntax parser has not been tested on a wide range of PDFs containing the less common PDF objects, or a PDF with updates. We will also determine how to handle inheritable fields in PDFs, such as checking that the required MediaBox attribute is either present or inherited from a parent. Additional testing and development is needed to develop a robust syntax parser. In addition, we plan to extend the semantic verification capabilities. For better detection of malicious behavior, we will also add semantic checks for OpenAction items sending data to external websites or executing javascript. While we have focused entirely on Portable Document Format analysis, the verified syntax and semantic analysis framework and Tower Metalanguage can be extended to other document formats. While the next document format's objects may not have types or object numbers, the overall Tower structure of using proof by induction to verify functions checking all objects in a list will remain.

These directions for potential future work will build upon language-theoretic security in developing safe and secure parsers, and better determine how to automatically accurately assess the validity and conformance of a variety of documents to a standard.

ACKNOWLEDGMENT

The authors would like to thank John Sarracino and all the anonymous reviewers on the LangSec Program Committee for their helpful comments on this submission. The authors would also like to thank Sergey Bratus, Clement Pit-Claudel, Paul Vines, and Peter Wyatt for their contributions and suggestions in support of this work. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0072. 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 DARPA.

REFERENCES

[1] A. Blonce, E. Filiol, and L. Frayssignes, "Portable document format (pdf) security analysis and malware threats," in *Presentations of Europe BlackHat 2008 Conference*, 2008.

[2] G. Endignoux, O. Levillain, and J.-Y. Migeon, "Caradoc: a pragmatic approach to PDF parsing and validation," in *2016 IEEE Security and Privacy Workshops (SPW)*. Ieee, 2016, pp. 126–139.

[3] P. Maupin, "pdfrw 0.4," 2017. [Online]. Available: https://pypi.org/project/pdfrw/

[4] J. Muller, F. Ising, V. Mladenov, C. Mainka, S. Schinzel, and J. Schwenk, "Practical decryption exfiltration: Breaking pdf encryption," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 15–29.

[5] M. Maass, W. L. Scherlis, and J. Aldrich, "In-nimbo sandboxing," in *Proceedings of the 2014 Symposium and Bootcamp on the Science of Security*, 2014, pp. 1–12.

[6] S. Bratus, M. L. Patterson, and A. Shubina, "The bugs we have to kill," *; login:: the magazine of USENIX & SAGE*, vol. 40, no. 4, pp. 4–10, 2015.

[7] I. ISO, "Document management — portable document format — part 2: PDF 2.0," *International Organization for Standardization, Geneva, Switzerland*, 2017.

[8] F. Momot, S. Bratus, S. M. Hallberg, and M. L. Patterson, "The seven turrets of babel: A taxonomy of langsec errors and how to expunge them," in *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 2016.

[20] R. S. Boyer and J. S. Moore, "Single-threaded objects in ACL2," in *International Symposium on Practical Aspects of Declarative Languages*. Springer, 2002, pp. 9–27.

[9] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–22, 1990.

[10] E. Serna and A. Serna, "Power and limitations of formal methods for software fabrication: Thirty years later," *Informatica*, vol. 41, no. 3, 2017.

[11] M. Kaufmann and J. S. Moore, "Acl2: An industrial strength version of Nqthm," in *Proceedings of 11th Annual Conference on Computer Assurance. COMPASS'96*. IEEE, 1996, pp. 23–34.

[12] H. Liu, "Formal specification and verification of a JVM and its bytecode verifier," Ph.D. dissertation, 2006.

[13] J. S. Moore and M. Martinez, "A mechanically checked proof of the correctness of the boyer-moore fast string searching algorithm," *pat*, vol. 1, p. 14, 2009.

[14] M. M. Wilding, D. A. Greve, R. J. Richards, and D. S. Hardin, "Formal verification of partition management for the AAMP7G microprocessor," in *Design and Verification of Microprocessor Systems for High-Assurance Applications*. Springer, 2010, pp. 175–191.

[15] M. Patterson and D. Hirsch, "Hammer parser generator," 2014.

[16] G. Eakman, H. Reubenstein, T. Hawkins, M. Jain, and P. Manolios, "Practical formal verification of domain-specific language applications," in *NASA Formal Methods Symposium*. Springer, 2015, pp. 443–449.

[17] P. Gutmann, "The design and verification of a cryptographic security architecture," Ph.D. dissertation, ResearchSpace@ Auckland, 2000.

[18] D. Lukan, "Pdf file format: Basic structure," 2018. [Online]. Available: https://resources.infosecinstitute.com/pdf-file-format-basic-structure/

[19] S. Medeiros and R. Ierusalimschy, "A parsing machine for PEGs," in *Proceedings of the 2008 symposium on Dynamic languages*, 2008.

[21] H. R. Chamarthi, P. C. Dillinger, and P. Manolios, "Data definitions in the ACL2 sedan," *arXiv preprint arXiv:1406.1557*, 2014.

[22] P. Wyatt, "Arlington PDF DOM," 2021. [Online]. Available: https://github.com/pdf-association/PDF20_Grammar

[23] M. Lindlar, Y. Tunnat, and C. Wilson, "A pdf test-set for well-formedness validation in JHOVE-the good, the bad and the ugly," 2017.

[24] A. Bogk and M. Schöpl, "The pitfalls of protocol design: attempting to write a formally verified pdf parser," in *2014 IEEE Security and Privacy Workshops*. IEEE, 2014, pp. 198–203.

[25] P. Mundkur, L. Briesemeister, N. Shankar, P. Anantharaman, S. Ali, Z. Lucas, and S. Smith, "The parsley data format definition language," 2020.

[26] A. Coglio, "A formalization of the ABNF notation and a verified parser of ABNF grammars," in *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer, 2018, pp. 177–195.

[27] B. Delaware, S. Suriyakarn, C. Pit-Claudel, Q. Ye, and A. Chlipala, "Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats," *Proceedings of the ACM on Programming Languages*, vol. 3, no. ICFP, pp. 1–29, 2019.