Research Report:
ICARUS: Understanding De Facto Formats By Way of Feathers and Wax

Sam Cowger, Yerim Lee, Nichole Schimanski, Mark Tullsen, Walter Woods, Richard Jones, EW Davis, William Harris
Galois, Inc.
{sam,ylee,nls,tullsen,waltw, richard,wrharris}@galois.com

Trent Brunson, Carson Harmon, Bradford Larsen, Evan Sultanik
Trail of Bits
evan.sultanik@trailofbits.com

***Abstract*—When a data format achieves a significant level of adoption, the presence of multiple format implementations expands the original specification in often-unforeseen ways. This results in an implicitly defined, de facto format, which can create vulnerabilities in programs handling the associated data files. In this paper we present our initial work on ICARUS: a toolchain for dealing with the problem of understanding and hardening de facto file formats. We show the results of our work in progress in the following areas: labeling and categorizing a corpora of data format samples to understand accepted variations of a format; the detection of sublanguages within the de facto format using both entropy- and taint-tracking-based methods, as a means of breaking down the larger problem of learning how the grammar has evolved; grammar inference via reinforcement learning, as a means of tying together the learned sublanguages; and the defining of both safe subsets of the de facto grammar, as well as translations from unsafe regions of the de facto grammar into safe regions. Real-world data formats evolve as they find use in real-world applications, and a comprehensive ICARUS toolchain for understanding and hardening the resulting de facto formats can identify and address security risks arising from this evolution.**

I. Introduction

Understanding the security properties of a data format poses unique challenges due to the necessity of understanding the consequences of a data format on the three states of data: data at rest, data in use, and data in transit. While the security properties of data at rest rely only on the format itself, security properties for data in use or data in transit also rely on factors outside of each format's definition. A parser implemented in C, for instance, might be subject to *Return-oriented programming* (ROP) attacks. Another parser might be safe in the context of its own control stack, but vulnerable to attacks such as zip bombs [10], [5]. Data in transit presents orthogonal risks, such as mechanisms for exfiltrating sensitive or protected information. Further complicating the story is the emergence of *de facto standards*: "dominant implementations of these formats extend the [published] standards by deliberately accepting non-compliant inputs without any indication to the users that the document contains malformations silently presumed benign"[1]: what now *is* the de facto standard? And how benign is it?

A. The ICARUS project

Our project, ICARUS, has three major goals: (1) develop methodologies and tools to *discover*, as well as describe, de facto data formats, (2) develop methodologies and tools to identify safe, unambiguous subsets of a de facto format, and (3) develop tools to translate from a de facto format to a safe subset thereof. We have been focusing on accomplishing these goals for the PDF format, although with a view toward general applicability of our methods.

[1]SafeDocs Broad Agency Announcement.

B. Summary

In Section II we discuss some of the approaches we are taking to understanding and learning de facto formats:

- Labeling and categorizing data sets used for grammar inference using extant parsers. Corpora of samples of a given data format tend to include both positive and negative examples of the format. These examples often do not come with labels indicating how strictly they adhere to the target language. By leveraging extant parsers, we generate sets of labels for each example in a corpora to build a feature space allowing this classification to be performed.
- Detection of sublanguages within formats. Many formats make use of sublanguages to improve their expressive power; this power naturally comes at the cost of increased complexity of the format at large. In solving the problem of inferring the structure of an entire format, it is worthwhile to detect, isolate, and possibly operate on strings in these sublanguages.
- Grammar inference via reinforcement learning. Modern grammar inference tools are predominately focused on identifying structures in natural languages [8], [18], [7], [4] or on data formats more simple than PDF [22], [12], [11]. We consider a novel algorithm for inferring data structures in *Visibly Pushdown Languages* (VPLs) [2] by combining reinforcement learning with bottom-up merge parsing.

In Section III we discuss our approach to defining safe subsets of de facto formats as well as creating tools to transform data into safe subsets, without sacrificing file validity and semantic equivalence.

ICARUS will tie the above methods together, providing a framework which can be used to understand and identify security concerns for a format. The framework takes advantage of having both a corpus of files in the de facto format *as well as* having a set of preexisting (de facto) parser implementations. In other words, we study the format from the supply side (the corpus) as well as the demand side (the preexisting parsers). Even the most robust data specification grammar often is extended for usability (e.g., JSON, or less-originally-safe, PDF), and these extensions open new facets for exploiting the data streams using the format. Applying the aforementioned tools to both examples of an extended data format grammar and to the parsers designed for handling those extensions, we hope to illuminate common issues in format design which lead to exploitable parsers, and to provide tools for safely subsetting data files to enhance file safety, even in untested parsers.

II. Learning De Facto Formats

Learning a de facto format is an inexact endeavor. For example, the PDF ecosystem consists of many parsers which do not agree as to what constitutes a valid file. Here we discuss some of the techniques we have been exploring for learning de facto formats, and comment on their effectiveness.

A. Cleaning grammar inference datasets via existing parsers

To understand a de facto format, there must be a clear separation between good and bad examples of the format. A large corpus of example files, such as GovDocs [14], tends to be assembled indiscriminately, making it difficult to effectively understand the format and learn a grammar. A tool which helps classify members of an indiscriminate corpus either as valid members of the de facto format, or as files to be ignored, is necessary to tackle this problem.

When one or more parsers exist for a data format, each parser contains information regarding the format's valid documents. One method of leveraging this empirical knowledge of the de facto format is to run existing parsers on a corpus of data, and then to have a format expert annotate the parser outputs to determine which outputs denote a file that is not in the format. Using these annotations as a guide, files which may be bad examples of a data format may be filtered out from training, or included as negative examples. This step improves the quality of the corpus used for grammar inference.

Initial results

We have built a proof-of-concept file processing pipeline, temporarily called the "PDF Observatory," which consists of several phases:

1) Run all available parsers on each file in the corpus.
2) For each parser and file, iterate over lines output to `stdout` and `stderr` when running the parser on the file. For each output line, the tool strips information which looks instance-specific—such as memory addresses or numbers—and aggregates that information into a count of the number of times each message was seen.
3) For each parser, we then aggregate all error messages across all files. A similarity metric (Python's `difftool.SequenceMatcher`) is used to further group together messages with similar semantic content.

The output of this pipeline is a list of grouped error messages, and the files to which they pertain. These groups of

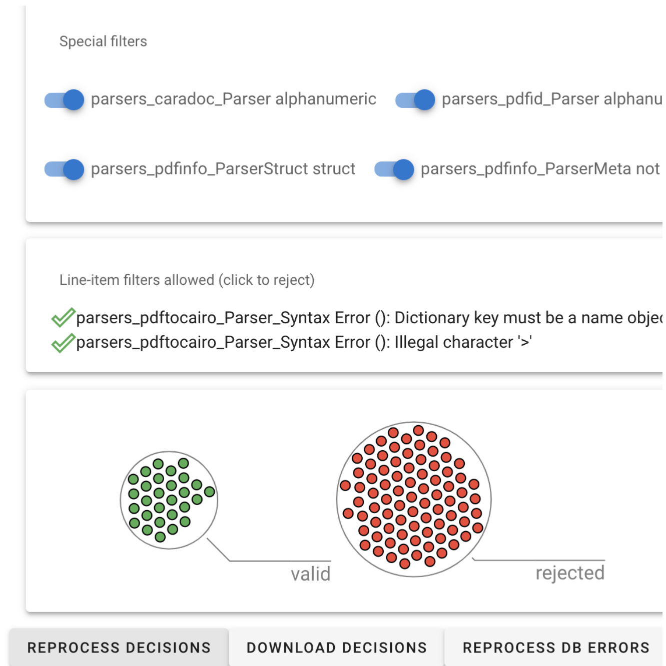


Fig. 1. Screenshot of PDF Observatory UI. Using the tool, a format expert might decide that e.g. pdftocairo produces a couple error messages which might apply to in-format files. By adjusting the filters, the user can observe population-level changes in the example corpus of the de facto format.

messages are then annotated by a format expert as indicating a file is in- or out-of-language, and then files containing an out-of-language error message in the parse are excluded from subsequent grammar inference. On Govdocs [14], this results in including only 58 % of the corpus as in-language examples for subsequent PDF grammar inference.

The UI is demonstrated in Fig. 1. To investigate the effects of classifying certain error message groups as in-format, such as "Illegal character <> in hex string," a format expert might select those messages as allowed and then re-process all file decisions. The UI reported that, after allowing non-name dictionary keys and illegal characters in hex strings, eight additional files would have been classified as in-language.

B. Identifying Sublanguage Segments

Grammar inference algorithms are flexible but slow because they must consider many permutations of possible features. The search for a suitable grammar might therefore be considerably sped up through pre-processing steps to identify sublanguages within the composite data format grammar. Recent successes for dealing with composite languages in the parser space further motivates this approach [6].

1) Entropy-based methods: Among the sublanguages that one may wish to identify, even if only to exclude from future learning tasks, is compressed data. Such data may easily derail a more sophisticated learner, especially if it represents a relatively large portion of the corpus being learned from. *E.g.*, in PDF, the predominant format for textual and/or visual content is a so-called *stream*, a data structure often filled

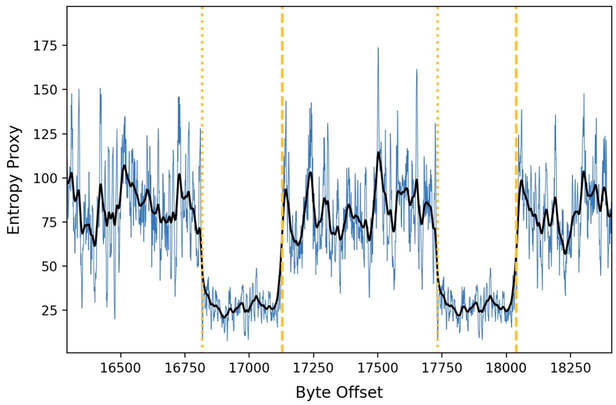


Fig. 2. Measured entropy at different byte offsets within a single PDF file. Dashed lines denote the actual start of a compressed region, while the dotted lines denote the end. (The figure shows three compressed regions - the ending bytes of one, a second in its entirety, and the beginning of third.)

with compressed bytes. With this in mind, and with a goal of furthering our ability to infer and reason about internal uncompressed PDF structures, we set out to tag compressed portions as such.

Calculation of entropy within streams is not a new concept, appearing in [20], although it has only been treated as a curiosity. For a first pass at leveraging entropy (or a proxy thereof), we relied on principles of compression and ASCII character encodings. Human-readable characters all take on values less than 128 when interpreted as bytes; compressed data have no such restriction, and indeed ought to use as many bits as are available. We sought to see if the average difference (within a sliding window) between encodings of adjacent characters, i.e. the characters' byte values, would function as a proxy measure for entropy of text, and therefore a suitable classifier for compressed data. Formally, for each n-gram S composed of bytes $\{S_i | i \in [0, n)\}$ in a file, we calculated

$$\frac{\sum_{i=0}^{n-2} |S_i - S_{i+1}|}{n-1}$$

to efficiently approximate the entropy of that region. We call this measure the *encoding entropy*.

Initial results

Though somewhat rudimentary, this prospect yielded promising results. The average encoding entropy between any two adjacent characters in a large plain-text file and a similarly-sized JPEG image differ significantly (31.3 for the plain-text file, 97.2 for the image). When applying this measure to a sliding window in a PDF file, uncompressed segments presented themselves as significant visual dips in a plot of this entropy figure as a function of byte offset. This can be seen in Fig. 2. When properly smoothed, these entropy measures become a highly usable indicator of compression. With well-chosen parameters, running the tool on several samples of PDF files yield true positive rates (bytes classified as compressed that are in fact compressed) of around 75-98%, and false positive rates of $\sim$1% or less, respectively.

We are then able to take several bytes of context surrounding the newly-found offsets in the original file and synthesize patterns common to several of these windows. We provide a user of the tool with a selection of the most common patterns, presuming that the end of human-readable text would be delimited by a human-readable sentinel string. In practice, the tool is able to provide the user with possible patterns that include the correct delimiters with as little as one PDF file.

Should a user determine another metric they wish to apply to a particular window of text to determine its language or sublanguage, this approach could easily be extended to accommodate it. For example, other metrics of data randomness could be used to yield a base64-encoded segment detector, or an n-gram-frequency model could be employed to determine different human languages under this framework.

2) Taint tracking: Another way to identify sublanguages is to directly leverage the internal structures used by existing parsers. A parser's internal data structures give insight into a de facto grammar by drawing clear associations between related symbols. Symbols which are found to be semantically close might then be considered as forming a sublanguage. For example, the PDF specification provides a format for streams, sequences of arbitrary data whose lengths are stored in dictionaries preceding each stream's content. By looking at an existing parser, the length value in the dictionary preceding the stream data might be associated with the stream data that follows, as they both will be used by the same parser function. One way to gather this relational information from existing parsers is by applying *taint tracking*.

We automatically instrument parsers to track the input byte offsets that influence (or "taint") the operands to each instruction via an open-source tool called PolyTracker [21]. Each byte is assigned a unique identifier known as a *taint label*. As these tainted bytes are processed, new labels are created that denote the combination of two preexisting labels in a hierarchical structure. This hierarchical structure represents the provenance of the bytes and, more abstractly, can be thought of as a forest of unions between dependent types. This structure provides the capability to reason about how different combinations of bytes influence each other throughout a program's execution, and should theoretically embed a notion of the grammar accepted by the parser.

When ground truth is available for an input file (*i.e.*, a syntax tree or semantic labeling of the bytes), then the taint forest produced by PolyTracker can be mapped back to the ground truth to associate *actual* type dependencies. This ground truth can either be generated by hand or through an instrumented parser that maintains lexical information throughout the parse (*e.g.*, Kaitai Struct [1] or PolyFile [21]). For example, if a function in the parser is observed to only operate on bytes tainted from a certain syntactic structure in the ground truth, then we can reasonably conclude that that function is specialized to operate on that semantic type.

This naïve function labeling is predicated on the assumption that there is a bijection between semantic types and functions in the parser. This will not be the case in general: sometimes there will be multiple functions responsible for processing a single type, and in other cases a single, monolithic function will be responsible for processing *multiple* types. We have developed a novel tool and associated algorithm, PolyMerge [16], that handles these cases, producing a robust semantic labeling of functions. PolyMerge can merge the ground truth with taint information to identify the functions most specialized in operating on each type in the input file. The mapping algorithm works by using an *Interval Tree* to efficiently map ground truth types to functions, and then prunes the mapping based upon Shannon entropy (*i.e.*, associating functions with types on which they are maximally specialized) and the dominator forests of both the runtime control-flow graph of the parser, as well as that of the ground truth syntax tree [*Ibid.*].

This work is the first step in combining a data-flow analysis (*e.g.*, the AUTOGRAM grammar extraction algorithm [17]) with control-flow analysis (*e.g.*, the Mimid grammar extraction algorithm [15]), and extending the state of the art by incorporating compositional type information from labeled ground truth input. Existing approaches are limited to extracting context-free grammars from formats that have little or no dependent types. The PolyMerge algorithm helps advance the state-of-the-art toward automated grammar extraction from arbitrary, non-context-free parsers.

Types can also be partially inferred based upon the way in which tainted data are operated on by known functions in the parser (*e.g.*, in standard libraries, runtimes, or system calls). For example, if a byte sequence in the input file is observed to be passed as the second argument to `strcpy`, we can infer that it is a null-delimited string. If another byte sequence is observed to be passed as the third argument to `strncpy`, we can infer that it is an integer length associated with a dependently typed string. The bytes which specifically affect control-flow are also observed.

Taint analysis also provides insight into which bytes in the input file have semantic meaning. For example, if an input byte is *never* observed to have been operated on by the parser, we can infer that either the byte has no semantic value in the input file format (*e.g.*, it is a comment or optional buffer, which could be indicative of the potential for steganographic embedding), or that region of the input file might be an optional feature of the file format not implemented by the specific parser implementation.

This method naturally lends itself to differential analysis. Multiple input files of the same format can be submitted to the same, instrumented parser. Likewise, the same PDF can be submitted to multiple implementations of the same file format to automatically identify differences. A single file can be submitted to two incremental versions of the same implementation to identify new features and behavioral changes.

C. Grammar Inference via Reinforcement Learning

Once sublanguage segments have been identified and learned, a robust grammar inference algorithm is necessary to tie together these identified sublanguage segments. The de facto format under investigation was presumably designed with a specific set of semantics in mind, to which the grammar inference engine has no access. The matter is further complicated because any data format may be described by more than one grammar. Therefore, the goal of grammar inference in the context of de facto data formats is to recover as many semantics as possible in a terse, high-fidelity representation.

The ICARUS efforts have focused on building a grammar inference algorithm from the ground up, leveraging modern *Reinforcement Learning* (RL) advances and designing primitives which explicitly support common data grammar constructs such as the Kleene star and nested data structures. We chose to focus on the production of grammars in *Chomsky Normal Form* (CNF) [3], where every nonterminal symbol in the grammar might produce at most two other nonterminal symbols, or a single terminal symbol. In choosing to infer a CNF grammar, the parse may be done via repeatedly merging neighboring terminal or nonterminal symbols, a process we call *Merge Parsing*. To our knowledge, this is the first usage of the term "merge parsing" to describe this type of parsing, though this style of parser has appeared in the literature without a descriptive name [7]. Merge parsing is therefore a bottom-up approach, and reduces the problem of learning a parse to the problem of determining the correct order of merges, bounding the number of parsing steps on a sentence of length N to $N-1$ merges.

The proposed RL approach to grammar inference is illustrated in Fig. 3. In an RL framework, merge parses may be modeled as an actor with a variable number of actions, with each unique action defined by the position in the sentence and the merge action to be taken. For the example from Fig. 3, in the first step, an actor might merge any two adjacent characters of the input symbols $[A, B, A, B, A, B]$. The second step requires a merge of any adjacent characters from the new terminal symbols, $[A, B, A, BA, B]$. An actor executes at most $N-1$ steps, observing a reward for each action. At the end of the parse, all rewards are rolled back through time using a discount factor and are compared with the computed critic values to update the actor's policy. As RL benefits from dense reward signals, we use a function on the frequency of the token resulting from a merge as the reward function. That is, when merging "A" and "B", the result is "AB", whose frequency in the data samples would determine the reward. Subsequently, the "AB" symbol would be available for merging with its neighbors. This replacement operation would be written $merge(A, B) \implies AB$. Each sub-tree parsed is evaluated based on its own prevalence in the data format.

Thus far, the described algorithm would have difficulties tackling the problem of parsing a Kleene star. Consider the list-like grammar described by the regular expression $ap*le$. The algorithm thus described would represent "ap", "app",

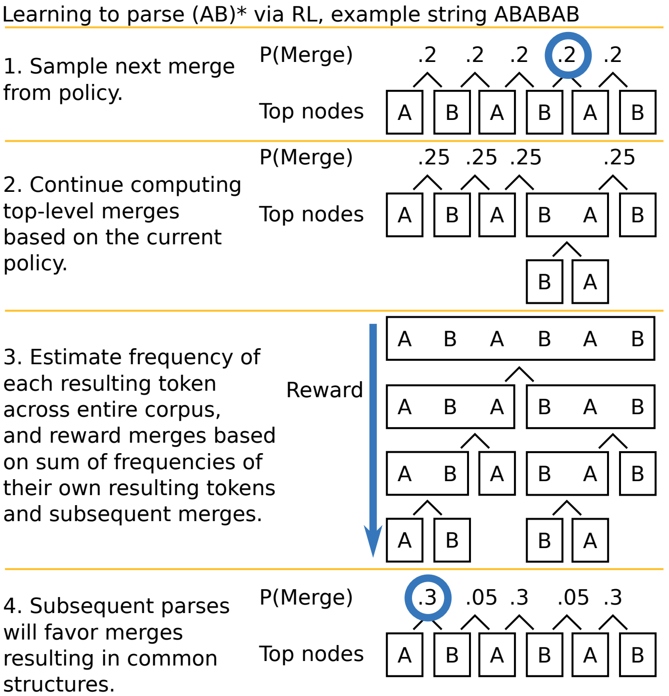


Fig. 3. Overview of the RL-based grammar inference approach. Beginning with a random merge policy, rewards based on the resulting parse tree are observed and the policy is adjusted to maximize the expected reward. This process eventually results in an approximation of an optimal policy.

and "appp" as different tokens, with no relation to one another. One benefit of RL is that it provides a flexible framework which accommodates the addition of new action types. Leaning on some of the assumptions of VPLs [2], we observe that an anchoring action might be added to the actor's action set, which merges two tokens but retains only one, e.g., $anchor(A, B) \implies A$ as opposed to the prior $merge(A, B) \implies AB$. With an anchoring action, each p following an a may be merged into a token which look like the a on its own, handling the Kleene star and keeping the relevant VPL pushdown token visible. As one of the primary goals of ICARUS is not only understanding, but also the manipulation of de facto data formats, retaining tokens from the original input string in this manner helps keep the inferred grammar interpretable, avoiding unnecessary nonterminals or difficult-to-interpret embeddings.

The result is an $\mathcal{O}(N \log N)$, bottom-up parsing algorithm, trained via any RL algorithm, which integrates a flexible action set that can be leveraged in future work for more complicated behavior. We note that high-level concepts could easily be integrated into this action-reward framework, such as "integer" for a sequence of numbers, or the handling of PDF streams which must read and decompress a specified number of bytes. These actions might be further informed by taint tracking, as described in Section II-B2. The tricky part is designing a meaningful reward function. We emphasize that each ad hoc data format was originally designed with unique semantics not visible to the grammar inference process, and as such, it is possible that there is no single reward function which performs best on all data formats. If there were one reward function or metric which captured the semantics of all data formats, then there would be no need for multiple data formats, as every use case would share a single, optimal encoding. The ICARUS grammar inference tool might, therefore, end up offering a suite of suitable reward functions, integrating concepts such as Minimum Description Length.

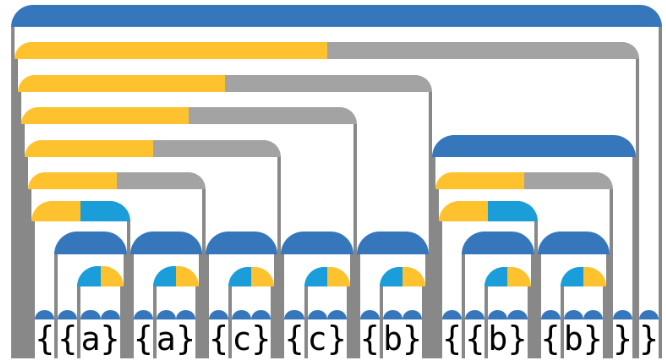


Fig. 4. Example parse of $S \rightarrow$ '{'(a|b|c|S+)'}', input string on bottom. Blue border indicates a standard merge ($merge(A, B) \implies AB$), Half-Yellow border with Half-Gray border indicates an anchor with ($anchor(A, B) \implies A$) behavior, and Half-Yellow border with Half-Light Blue border indicates a subgrammar substitution with $subgrammar(A, B) \implies A\{G\}$ behavior, where $\{G\}$ is a special catch-all token. The RL-based algorithm correctly learned the structure nesting of the format, the Kleene star behavior, and the OR clause regarding which letter is used as terminal data.

Initial results

The above grammar inference algorithm was tested on a simple, JSON-like grammar, $S \rightarrow$ '{'(a|b|c|S+)'}'. This grammar contains three aspects which are often present in data formats: (1) nested data structures, (2) a union of different data types inside the data structures, and (3) Kleene star behavior.

A parser was trained using the *Advantage Actor Critic* (A2C) algorithm [19], based on 128 samples of the grammar. In addition to the aforementioned anchoring actions, a special OR-oriented action $subgrammar(A, B) \implies A\{G\}$ was added. Here, $\{G\}$ is a special stand-in token which indicates that content was merged without being specific about which content was merged. The result may be seen in Fig. 4. The ICARUS grammar inference algorithm correctly learned that '{'(a|b|c)'}' is the base building block of this data format, and merged those structures first. Further, it correctly learned that the content between the '{' and '}' was unspecific, and substituted a $\{G\}$ token instead of propagating the original content. It also handled the Kleene star of the grammar correctly, and learned to parse nested structures consistently (see e.g. the parsing of the '{{b}{b}}' nested structure in the most-bottom-right part of the figure).

While inferring this simple grammar is a long way from inferring full, functional parts of the PDF specification, we view it as an encouraging step with a flexible base algorithm which may be up to the task.

Related work

Few existing grammar inference algorithms are designed for ad hoc data formats. One such system is PADS / LearnPADS [22], [12], [11], which focuses on the automatic generation of tools for human-readable formats such as web server logs. The grammar inference aspect, LearnPADS, relied on Minimum Description Length to recover data types and an overall structure from data format examples [11]. The LearnPADS project found that some semantics may not be safely recovered, if the example data's coverage of the format is insufficient. For example, they found that inferring the ranges of integer variables always led to overfitting [12]. It appears that LearnPADS was never focused on recovering full semantics for data formats as complicated as PDF.

Outside of PADS, most state-of-the-art grammar inference algorithms focus on natural languages [8], [18], [7], [4], and are not necessarily a good fit for uncovering data format grammars. For example, the readily available JDageem package [4] learns to build dependency tree representations of input strings, nesting each word in the string under other words to develop an understanding of the "root" words in different contexts. When a word is being considered as nested under some other, root word, there are no restrictions about the proximity of the two words. This flexibility may be desired for some queries on natural languages ("she ran to her house," where "her" would refer to "she"), but makes the algorithm significantly more resource intensive in the context of data formats.

For comparison with the ICARUS grammar inference algorithm, the JDageem package [4] was run on the same training data that produced the parser demonstrated in Fig. 4. The resulting parser was unable to discern the nested structure of the data, instead associating most brackets under a single instance of 'a' in the string.

More recent work by Drozdov *et al.* also utilized merge parsing, and demonstrated impressive results [7]. Instead of using RL, their algorithm learns the parse order based on an autoencoder-style loss function. In future work, we aim for a more comprehensive comparison with their algorithm.

III. Safe Subsets of De Facto Formats

Once the suite of tools has identified a common grammar shared by most example files, it is important to consider which aspects of the de facto format are ambiguous or utilize unsafe features. Furthermore, the tool suite might consider repairing files with these undesired features. If a semantically-equivalent, safer subset of the de facto grammar may be identified, then questionable files might be hardened by transforming them into the safer subset.

A. Safer, smaller formats

The motivations for defining a small, safe, unambiguous subset of PDF, or any data format, are many: 1) Unsafe constructs are opportunities for exploits of various forms (zip bombs, ex-filtration, general hacks). 2) Ambiguity—arising from a long-evolving specification and from dozens of diverse implementations—adds nothing to the usability of PDF. There are ambiguities at both the syntax level (how is a string intended to be parsed) as well as the semantic level (how should a construct be interpreted). We want to detect and/or remove these ambiguities because they are a source of errors, as well as contributors to parser complexity. 3) Over-complexity and redundancy in data formats increase the cost to write and maintain parsers; the increase in code size increases the attack surface of the parser. Clearly, if we *can* eliminate over-designed and redundant constructs, we would like to do so. In addition, we assume many PDF renderers have internal code that normalizes or simplifies the document before rendering; it would be useful to remove this burden from the renderer.

B. Transforming documents to subsets

One of the objectives of ICARUS is to develop tools and methodologies that will aid in *transforming* an unsafe data format to a safe subset. Using these, we will be writing a safe subsetting tool for PDF. This will also involve *defining* a safe PDF subset. Preliminary work has demonstrated that it is feasible to develop provably safe and correct parsers for relatively simple and well-understood subsets of practical formats, including PDF [9]. A subsetting tool will be essential in ensuring that such tools can be applied to parse PDF's as they exist in the wild.

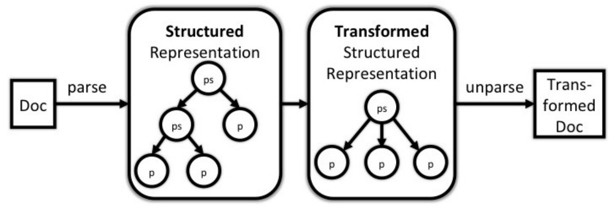
Developing a PDF safe subsetting tool will be challenging on many fronts:

- First of all, if it is to be adopted, it should accept the large number of PDF extensions and malformations that are used *in the wild*, i.e., in the numerous renderers, browsers, and tools existing for PDF.
- It must be able to remove unsafe constructs and ambiguities whenever possible, but at the same time, it should reject PDFs that are too far out of spec and provide clear justifications to the user for rejection.
- PDF is large and complex, and has multiple constructs that require maintaining non-local invariants, such as length fields, cross-references, and fields that represent byte-offsets in the PDF file. These involve transforming PDF at a deep semantic level.
- The tool must be correct; changing the "semantic content" of PDF should only be done when explicitly requested (e.g., when required for preserving safety, such as removing javascript).

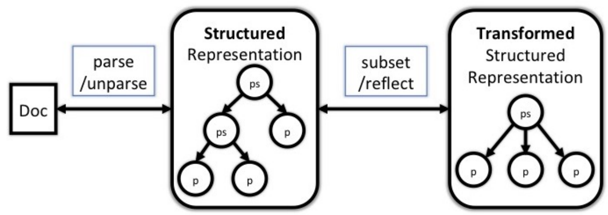
C. Implementing safe subsetting

To mitigate the above challenges and to provide greater assurance of the correctness of our PDF safe-subsetting tool, we will be using the conceptual notion of *lenses* [13]. Using lenses we can define our safe-subsetting tool as a composition of lens combinators and primitive lenses, each of which can be developed and validated separately.

Figure 5 demonstrates two different approaches to safe subsetting. Figure 5a shows the structure of a simple safe subsetting tool. Using the bidirectional nature of lenses, we can describe the safe subsetting tool using two lenses as shown in Fig. 5b. From the composition of these two lenses, we generate



(a) Subsetting: standard approach.



(b) Subsetting: using bidirectional programming.

Fig. 5. Different approaches to subsetting.

the safe-subsetter *function* by following the path from Doc: parse, subset, reflect, then unparse. Note that, given a lens, we can always extract a subsetter when we do this "round trip from the left." The "round trip from the right" will be the identity function. (E.g., parse · unparse $= id$.)

Initial results

The parse/unparse pair is itself a lens, the resulting subsetter it induces is useful for syntactic transformation: it will remove all extraneous white-space as well as transform the PDF syntax into a normalized form.

We have been using lenses to explore subsetting of PDF dictionary objects (an associative table defined by key value pairs). Even this apparently simple construct gives rise to some ambiguities. PDF as specified (1) does not allow for duplicate keys in a dictionary object, however PDF (2) does allow for null objects in the dictionary which should be equivalent to an absent entry. So, in the absence of any indication of which of these two rules has precedence we have an ambiguity in the specification.

For instance, here is a PDF dictionary with two keys `Size` and `Version`, which should be accepted in any implementation of PDF:

```
<< /Size 14 /Version 5 >>
```

According to the specification, dictionaries with duplicate keys should rejected. E.g., the following is rejected:

```
<< /Size 14 /Version 5 /Version 6 >>
```

PDF contains a null object and, per the specification, null dictionary entries shall be treated as absent, thus this dictionary

```
<< /Size null /Version 5 >>
```

is valid and is equivalent to

```
<< /Version 5 >>
```

However, the specification is unclear as to whether the following dictionary should be accepted:

```
<< /Size 14 /Version 5 /Version null >>
```

If we remove nulls before checking for duplicate keys, it would be accepted; if we check for duplicate keys first, it would be rejected. Lenses allow us to formalize these two different interpretations of PDF. To demonstrate this, we encode each rule as a lens, and we observe different semantics based on the order of composition:

$$\text{removeNullEntries} \cdot \text{rejectDictionaryDups}$$

$$\text{rejectDictionaryDups} \cdot \text{removeNullEntries}$$

If a PDF implementation is more lax, allowing for duplicate keys (as some implementations do allow), the number of possible interpretations increases further. Thus, the seemingly innocuous act of "allowing for user error" in the PDF gives rise to ambiguity. In this case, we would now have the possibility of a 'schizophrenic' PDF document, one with different interpretations depending upon the PDF implementation.

It is a moot point whether the PDF standard is currently ambiguous or could be made to be unambiguous. We do not expect various de facto PDF implementations to become perfectly compliant with the current standard. Our objective in ICARUS is to create tools that allow us to understand the implementations as well as to mitigate the problem of multiple implementations with varying interpretations.

IV. Conclusions

While the ICARUS project is still a work in progress, its early results have demonstrated how a toolchain might be assembled for understanding and securing de facto data formats. De facto formats often include features that are poorly defined or expose new security vulnerabilities. Identifying the ways in which a format has been modified is an important first step in securing the resulting format. Relying on existing parsers to discard out-of-language files helps to ensure that time is most efficiently spent understanding the de facto format. Detecting sublanguages via entropy-based methods can help isolate meaningful sections of grammar from file contents within the scope of individual files. Taint tracking can help leverage empirical knowledge, in the form of existing parsers, to uncover structure and data types within the de facto format. Taint tracking also provides additional features to feed into learning algorithms. A grammar inference algorithm based on merge parsing and reinforcement learning has shown promise for uncovering structures in data when an existing parser was unavailable. Subsetting has been discussed as a means for understanding aspects of data formats which lead to exploitable properties, and potentially repairing those exploits through the identification of an in-language, safer subset of the full de facto format. The efforts of the ICARUS project have resulted in prototypes of some of these tools using PDF as a case study, utilizing techniques that we believe will generalize and apply to other de facto formats. As data formats attract users, the formats naturally mutate in ways which can compromise the security of programs using them, making toolchains like ICARUS important for understanding and identifying real-world risks in data formats.

Acknowledgements

This research was supported by the SafeDocs program under HR0011-19-C-0073.

References

[1] Kaitai Struct: a new way to develop parsers for binary structures. https://kaitai.io/. Accessed: January 12, 2020.

[2] Rajeev Alur and Parthasarathy Madhusudan. Visibly pushdown languages. In *Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 202–211, 2004.

[3] Noam Chomsky. On certain formal properties of grammars. *Information and Control*, 2(2):137 – 167, 1959.

[4] S. B. Cohen and N. A. Smith. Covariance in unsupervised learning of probabilistic grammars. *Journal of Machine Learning Research*, 11:3017–3051, 2010.

[5] Russ Cox. Zip files all the way down, March 2010.

[6] Joel Denny. Pslr (1): pseudo-scannerless minimal lr (1) for the deterministic parsing of composite languages. 2010.

[7] Andrew Drozdov, Patrick Verga, Mohit Yadav, Mohit Iyyer, and Andrew McCallum. Unsupervised latent tree induction with deep inside-outside recursive auto-encoders. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 1129–1141, 2019.

[8] Chris Dyer, Adhiguna Kuncoro, Miguel Ballesteros, and Noah A Smith. Recurrent neural network grammars. In *Proceedings of NAACL-HLT*, pages 199–209, 2016.

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

[10] David Fifield. A better zip bomb. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[11] Kathleen Fisher and David Walker. The pads project: an overview. In *Proceedings of the 14th International Conference on Database Theory*, pages 11–17, 2011.

[12] Kathleen Fisher, David Walker, Kenny Q Zhu, and Peter White. From dirt to shovels: fully automatic tool generation from ad hoc data. *ACM SIGPLAN Notices*, 43(1):421–434, 2008.

[13] J. Nathan Foster, Michael B. Greenwald, Jonathan T. Moore, Benjamin C. Pierce, and Alan Schmitt. Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem. *ACM Trans. Program. Lang. Syst.*, 29(3):17–es, May 2007.

[14] Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt. Bringing science to digital forensics with standardized forensic corpora. *digital investigation*, 6:S2–S11, 2009.

[15] Rahul Gopinath, Björn Mathis, and Andreas Zeller. Inferring input grammars from dynamic control flow. *arXiv preprint*, December 2019. `arXiv:1912.05937v1 [cs.SE]`.

[16] Carson Harmon, Bradford Larsen, and Evan Sultanik. Toward automated grammar extraction via semantic labeling of parser implementations. In *Proceedings of the LangSec Workshop*, IEEE Security & Privacy, May 2020.

[17] Matthias Höschele and Andreas Zeller. Mining input grammars from dynamic taints. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 720–725, New York, NY, USA, 2016. Association for Computing Machinery.

[18] Yoon Kim, Chris Dyer, and Alexander M Rush. Compound probabilistic context-free grammars for grammar induction. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2369–2385, 2019.

[19] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[20] Didier Stevens. `pdfid.py`. https://github.com/DidierStevens/DidierStevensSuite/blob/master/pdfid.py, November 2019.

[21] Evan Sultanik, Brad Larsen, and Carson Harmon. Two new tools that tame the treachery of files. https://blog.trailofbits.com/2019/11/01/two-new-tools-that-tame-the-treachery-of-files/, November 1, 2019. Accessed: January 12, 2020.

[22] Kenny Q Zhu, Kathleen Fisher, and David Walker. Incremental learning of system log formats. *ACM SIGOPS Operating Systems Review*, 44(1):85–90, 2010.