Toward Automated Grammar Extraction via Semantic Labeling of Parser Implementations

Carson Harmon
Trail of Bits
New York, NY
carson.harmon@trailofbits.com

Bradford Larsen
Trail of Bits
New York, NY
brad.larsen@trailofbits.com

Evan A. Sultanik
Trail of Bits
New York, NY
evan.sultanik@trailofbits.com

***Abstract*—This paper introduces a new approach for labeling the semantic purpose of the functions in a parser. An input file with a known syntax tree is passed to a copy of the target parser that has been instrumented for universal taint tracking. A novel algorithm is used to merge that syntax tree ground truth with the observed taint and control-flow information from the parser's execution, producing a mapping from types in the file format to the set of functions most specialized in operating on that type. The resulting mapping has applications in mutational fuzzing, reverse engineering, differential analysis, as well as automated grammar extraction. We demonstrate that even a single execution of an instrumented parser with a single input file can lead to a mapping that a human would identify as intuitively correct. We hope that this approach will lead to both safer subsets of file formats and safer parsers.**

***Index Terms*—dynamic tainting, input formats, grammar synthesis, lexical analysis**

I. Introduction

Almost all modern programs interact with third-party data represented in a variety of formats. For example, modern web browsers must parse HTML, JavaScript, CSS, XML, and a multitude of other document formats. Most document formats such as DOCX and PDF are actually containers that can store binary blobs of arbitrary formats and encodings. Some document formats are neatly described with formalisms like context-free grammars and ASN.1 encoding. However, the majority of widely used document formats are specified either through a reference implementation (such as older Microsoft Office binary formats), a complex and often ambiguous human-readable specification (such as PDF 2.0), or are not formalized at all (such as CSV). This makes it challenging to precisely characterize the set of input documents that a web browser (and more generally, an arbitrary program) will process without error. In practice, this devolves to a task of reasoning about parser implementations.

Parsers of complex file formats are difficult to implement correctly, particularly when they are not derived from a formal grammar, or when they are written largely by hand rather than by tools like parser generators. This creates the possibility of several types of implementation defects and software vulnerabilities, such as mishandling of well-formed documents, behavioral differences between alternative implementations (so-called *schizophrenia* [1]), accidental Turing-completeness [2], steganographically hidden payloads, and data exfiltration.

Our goal is to identify safer subsets of file formats devoid of misfeatures and ambiguities that lead to the aforementioned defects. We hypothesize that the "unsafe" portions of a file format exist in the symmetric difference of the grammars accepted by its various parser implementations. The portions of the file format to *keep* are the ones accepted and interpreted equivalently across all implementations. For this, we need a technique for automatically extracting the grammar specifying the inputs accepted by a parser. This paper describes the first step toward this goal: a novel algorithm for labeling the semantic purpose of each functional component in a parser implementation.

We are developing a collection of tools that expose a parser's processing dependencies on its input document. In this paper:

- We extend the DataflowSanitizer [3], [4] in LLVM for low-overhead dynamic taint tracking of individual input bytes in C and C++ programs (and eventually any program represented in LLVM/IR).
- We introduce a tool that maps an abstract syntax tree generated from an instrumented, permissive parser to the bytes of an input file. By "permissive", we mean a parser that is maximally resilient to malformations and deviations from the specification.
- We present a novel matching algorithm to combine the output of these two tools to annotate the functions within a black-box parser with their semantic purpose.
- We present a case study on PDF and JPEG parsers.

Several areas could benefit from these tools, including automated grammar recovery, high-quality input synthesis for fuzz testing, and augmentation of reverse engineering capabilities.

II. The Approach

There are three steps to semantically labeling a parser:

1) Label the ground truth semantic hierarchy of an input file.
2) Instrument the parser for universal taint tracking.
3) Merge and refine the output of steps 1 and 2 to produce a labeling that maps types within the input file to the function or set of functions within the parser that are most specialized for processing that type.

Each step can be executed automatically. While elements of steps 1 and 2 are novel, the primary contribution of this paper

is the merging algorithm of step 3, as well as the novelty of the overall approach toward grammar extraction.

The following three sections discuss the steps, respectively.

A. Labeling the Type Composition Hierarchy of an Input Stream

Ground truth for the semantics of an input file can be obtained manually by labeling the semantic meaning of the bytes within the file. However, ground truth can also be obtained automatically by modifying an existing parser or parser-generator to retain lexical context throughout the parse. Then the abstract syntax tree or parse tree emitted by the parser can be annotated with the original byte offset of each node's token. Thus, a semantic meaning—effectively a composed type—can be assigned to each byte in the input file.

We have released an open-source tool, PolyFile, that can automatically and efficiently perform semantic labeling of files [5]. PolyFile supports a variety of formats, including notoriously hard-to-parse formats like PDF and HTML, as well as any format specified in a Kaitai Struct grammar [6]. It is available at https://github.com/trailofbits/polyfile.

B. Precise Taint Tracking from an Input Stream

We propose to gather ground-truth information about parsers by performing *universal taint analysis*. Universal taint analysis is a program analysis technique that tracks each input byte throughout the execution of a program. 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 pre-existing 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 related 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.

There are challenges when performing universal taint analysis on real-world software. When the cyclomatic complexity of a program is large, the amount of new taint labels generated from even a small input is enormous. As an example, many document formats such as PDF use compression to keep document sizes small. When PDF parsers decompress the data, not only do we need to track the original bytes, but we need to create a new taint label for every combination of bytes and the newly decompressed data. This phenomenon is known as *taint explosion*, which generally occurs when a function performs a large number of combinatorial operations on input data.

Another challenge with universal taint analysis is implicit control flow. There are scenarios in which an input byte will be used in a comparison, resulting in a new operation on another variable. Technically, the tracked byte never directly touches the new variable, but the byte *does* influence the variable's value. In such a case, we choose to create new labels only when two existing labels are directly operated on. That is, we intentionally ignore some control flow, resulting in what is referred to in other work as *undertainting* [7]. This is in contrast to *overtainting*, where new labels are created based on the implicit control flow; Figure 1 shows the difference between the two. In the analysis of large codebases, overtainting quickly causes taint explosion. Furthermore, creating new labels based on control flow might mean the data collected will be less than ground truth.

```
char buff[3];
...
int num_bytes = read(fd, buff, sizeof(buff));
if (strcmp(buff, "PNG") == 0) {
    // y will only be is tainted when overtainting
    int y = 10;
}
```

Fig. 1: An example of undertainting versus overtainting. The variable `y`'s value is constant and therefore not affected by input bytes. However, the control flow that will cause `y` to be instantiated *is* dependent on the string `buff`, which *is* tainted by user input. Therefore, the *existence* of `y` is predicated on tainted data. If we consider `y` to be tainted, this is so-called "overtainting".

Beyond undertainting, to help reduce taint explosion and increase accuracy further, we have implemented an exponential decay system for taint labels. The decay system is implemented by assigning every taint label a *strength*. As new labels are created from interactions with each other, the strength of each interacting label is reduced. When a label's strength becomes zero, the label decays to nothing. With this approach, taint explosion is avoided, and some completeness is sacrificed for more accurate data. This allows us to continue tracking as many bytes as possible even in the presence of functions with high cyclomatic complexity.

There are several existing projects that achieve universal taint tracking, using various methods. The best maintained and easiest to use are AUTOGRAM [8] and TaintGrind [9]. However, the former is limited to analysis within the Java virtual machine and the latter suffers from unacceptable runtime overhead when tracking as few as several bytes at a time. For example, we ran `mutool`, a utility in the muPDF project, using TaintGrind over a corpus of medium sized PDFs, and in every case the tool had to be halted after over twenty-four hours of execution for operations that would normally complete in milliseconds without instrumentation.

The LLVM DataflowSanitizer (dfsan) [3], [4] can instrument a program at compile time to perform universal taint tracking at runtime. However, its design severely limits the number of input bytes it can track to only about sixty-five thousand taint unions during execution. We developed a novel data structure capable of exploiting the inherent sparsity in taint unions to increase this amount to *billions*, enabling universal taint tracking with marginal runtime overhead. This has resulted in an open-source tool, PolyTracker, that can automatically instrument software represented in the LLVM intermediate representation with this universal taint tracking approach [5]. It is available at https://github.com/trailofbits/polytracker.

C. Associative Labeling

The third and final step of the approach is to merge the results to label the functions of the parser by their semantic purpose. Recall that universal taint tracking provides a list of offsets from input bytes used by each function during execution. We must first map those byte offsets to elements in the ground truth semantic labeling from Step 1 of our approach (described above in §II-A). This mapping can be performed efficiently in linear time using an *interval tree* data structure to represent the ground-truth semantic hierarchy (lines 2–5 of Algorithm 1).

Let $T = \{\text{PDF-OBJECT, PDF-DICTIONARY, PDF-DICTIONARY-KEY, JFIF-HEADER, } \ldots\}$ be the set of semantic types and let $E = \{e_0, e_1, \ldots\}$ be a sequence of semantic elements where each $e_i = \langle \text{offset}, \text{length}, t \in T \rangle$ corresponds to the ground-truth labeling of the input file. Let F be the set of parser functions recorded by the instrumentation (from Step 2, described above in §II-B). The interval tree, therefore, gives a mapping, $M : E \to 2^F$, from the semantic elements to their corresponding set of functions.

In this raw mapping, the majority of types will map to generic utility functions used for operations like string copying and decoding. For example, PDF object streams are typically compressed, so every PDF object stream element in the input file will be mapped to whichever function in the parser is responsible for LZSS decompression [10]. As another example, the ZIP file format contains many dependently typed elements (*e.g.*, dynamically sized strings), which will all likely map to generic utility functions like `strncpy`. Therefore, we ultimately want to refine this mapping to contain only the function or functions most *specialized* in operating on a specific type. We want to avoid associating generic utility functions that may operate on every instance of an input element of a given type, but are not specialized for operating on that type. Note that such a mapping is not necessarily injective; one parser might have a single, monolithic function to operate on a set of semantic types, while another implementation might split those operations into several functions.

We use information entropy to measure function specialization. For each type $t \in T$, collect the set of functions that operate on that type (Algorithm 1 lines 6–15):

$$F_t = \bigcup_{e \in E} \begin{cases} M(e) & \text{if } e.t = t, \\ \emptyset & \text{otherwise.} \end{cases}$$

Next, calculate the probability of a specific type occurring within a function, $P : T \times F \to [0, 1]$. If $f \notin F_t$, then $P(t, f) \mapsto 0$. Otherwise, if $f \in F_t$:

$$P(t \in T, f \in F_t) \mapsto \frac{|\{e \in E : e.t = t \wedge f \in M(e)\}|}{|\{e \in E : f \in M(e)\}|}.$$

This can be used to calculate the "genericism" of a function, $G : F \to \mathbb{R}$, via its Shannon entropy (Algorithm 1 line 20):

$$G(f \in F) \mapsto -\sum_{t \in T} P(t, f) \log_2 P(t, f).$$

A value with lower entropy—or genericism—indicates a function that is specialized to process its associated type elements. Higher entropy, on the other hand, indicates a more common utility function, like character decoding or decompression. The less generic a function, the more specialized it is at processing its associated type elements.

We use G to sort the functions associated with a type, discarding all but the smallest (most specialized) standard deviation (Algorithm 1 lines 22 and 23). This produces a succinct mapping, $M' : T \to 2^F$, from types to the functions most specialized for operating on that type. Standard deviation is used here as a threshold to maintain multiple functions in the event that they are almost equally specialized; it has proven to work well in our experiments, and further investigation into the sensitivity of the algorithm to this threshold is planned.

A parser's functional implementation will rarely be isomorphic to the type hierarchy or syntax tree of the input file and, therefore, M' will rarely be a perfect bijection between the types and functions. For example, a parser might have *multiple* functions that are collectively responsible for parsing a given type, and are therefore equally specialized in that type. Conversely, an insufficiently modularized parser might have a single function responsible for parsing a multitude of types. We therefore introduce two refinements to the mapping to address these cases.

First, we address the case of multiple functions specializing on a single type. If those functions are always called sequentially in the course of execution, then we ideally only want to identify the single function that *initiates* the sequence. To accomplish this, we first calculate the *dominator tree* [11] of the runtime control-flow graph (Algorithm 1 line 16). For each type $t \in T$, we remove any functions in $M'(t)$ that have an ancestor in the dominator tree that is also in $M'(t)$ (Algorithm 1 line 23).

Finally, we perform the dual of this operation to address the case in which a single function is specialized for multiple types. Up to this point, only the frequency of types in the input file has been utilized. We can make one final optimization to the mapping by extracting the *type composition* hierarchy of the input file. For example, a file of type PDF can be decomposed into a hierarchy like that pictured in Figure 2. Similar to the prior refinement, we calculate the dominator tree of the type composition hierarchy, as in Figure 3 (Algorithm 1 lines 25 and 26). Then we remove a function from the mapping for a type, $M'(t) \setminus \{f\}$, if there exists an ancestor of t in the type hierarchy dominator tree that also maps to f (Algorithm 1 lines 27–33). In the event that this refinement would cause the mapping of a type to become empty, $M'(t) \mapsto \emptyset$, we only include the functions shared with the deepest ancestor in the dominator tree with which there is overlap.

III. Initial Results

We have applied this approach to a number of parsers that implement the PDF file format. PolyFile is used to semantically label the bytes of an input file, and PolyTracker is used to instrument the parsers at compile time for taint

Algorithm 1 Semantic labeling.

procedure MAP-TYPES-TO-FUNCTIONS($T, F, E, \mathcal{F}, G$)

Require: T is the set of types, F is the set of parser functions, $E = \{e_0, e_1, \ldots\}$ is the sequence of semantic elements $e_i = \langle \text{offset}, \text{length}, t \in T \rangle$ of the input file, $\mathcal{F} : F \to 2^{\mathbb{N}}$ maps each function to the set of byte offsets on which it operated, and G is the runtime control-flow graph of the parser.

Ensure: $M : T \to 2^F$ is the resulting mapping. /* Figure 4 */

/* record the bounds of every semantic element within the input file */

$R \leftarrow$ INTERVAL-TREE()

for all $e \in E$ **do**

R.ADD($[e.\text{offset}, e.\text{offset} + e.\text{length}), e$)

end for

/* construct mappings between elements, functions, and types */

Let $L : F \to 2^E$ be a dictionary mapping functions to sets of semantic elements, initialized s.t. $\forall f \in F : L[f] \mapsto \emptyset$.

Let $Y : T \to 2^F$ be a dictionary mapping types to sets of functions, initialized s.t. $\forall t \in T : Y[t] \mapsto \emptyset$.

for all $f \in F$ **do**

for all $b \in \mathcal{F}(f)$ **do** /* for each byte sequence on which function f operated */

for all $e \in R$.INTERSECT(b) **do** /* for each semantic element operated on by function f */

$L[f] \leftarrow L[f] \cup \{e\}$

$Y[e.t] \leftarrow Y[e.t] \cup \{f\}$

end for

end for

end for

/* prune the mapping by selecting only the most specialized functions */

$D \leftarrow$ DOMINATOR-TREE(G)

for all $t \in T$ **do**

Let $S : F \to \mathbb{R}$ be a dictionary mapping functions to their entropy

for all $f \in Y[t]$ **do**

$S[f] \leftarrow$ SHANNON-ENTROPY($e.t : e \in L[f]$)

end for

$\sigma \leftarrow$ STANDARD-DEVIATION($S[f] : f \in Y[t]$)

$M[t] \leftarrow \{f : S[f] \leq \sigma\} \setminus \{f \in Y[t] : f \text{ has an ancestor in } D \text{ that is also in } Y[t]\}$

end for

/* further prune the mapping by selecting the functions that operate on the shallowest types in the type hierarchy */

$G_T \leftarrow$ the type hierarchy graph constructed with one node for each $t \in T$ and a directed edge from t_i to t_j if $\exists e_i, e_j \in E : e_i.t = t_i \wedge e_j.t = t_j \wedge [e_i.\text{offset}, e_i.\text{offset} + e_i.\text{length}) \supset [e_j.\text{offset}, e_j.\text{offset} + e_j.\text{length})$ /* Figure 2 */

$D_T \leftarrow$ DOMINATOR-TREE(G_T) /* Figure 3 */

for all $t \in T$ **do**

for all $f \in M[t]$ **do**

if $\exists f' \in M[t] : f'$ dominates f in D_T **then**

$M[t] \leftarrow M[t] \setminus f$

end if

end for

end for

end procedure

tracking. The instrumentation overhead is negligible, on the order of seconds of execution for a typical PDF and parser.

A single PDF input file alone is sufficient to produce a mapping from the type hierarchy to implementation functions. For example, the mapping produced for the popular MuPDF reader is in Figure 4. This is the result of a single 4.4 kilobyte PDF—specifically, file 000021 from the GovDocs corpus [12]—that was rendered through an instrumented copy of `mutool`. Since the instrumented parser was compiled with symbols, it is clear to a human from the well-named functions in MuPDF that the mapping is intuitively correct. For example, the mapping correctly identifies the `pdf_read_start_xref` function as being responsible for parsing the `PDF Start XRef` type of the PDF specification. The method is agnostic to function names; it is based solely on the structure of the parser and the observed data-flow and control-flow. As such, the approach will work on any instrumented black-box binary, even in the absence of symbols. On a 2019 MacBook Pro, automatic

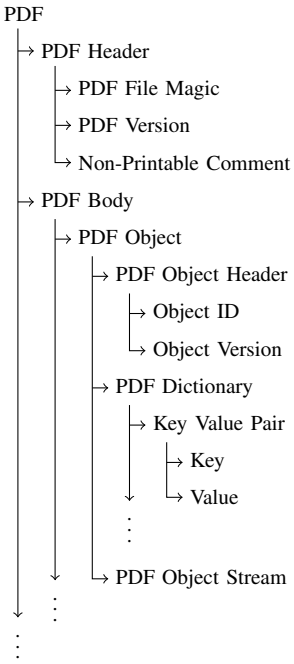

Fig. 2: Composite type hierarchy (or syntax tree) of a typical PDF file, generated from an instrumented parser (*e.g.*, PolyFile). Arrows denote "has-a" relationships: A PDF is comprised of a Header, followed by a Body, followed by a Trailer, *&c.*, each of which can be decomposed into sub-objects denoted by the arrows.

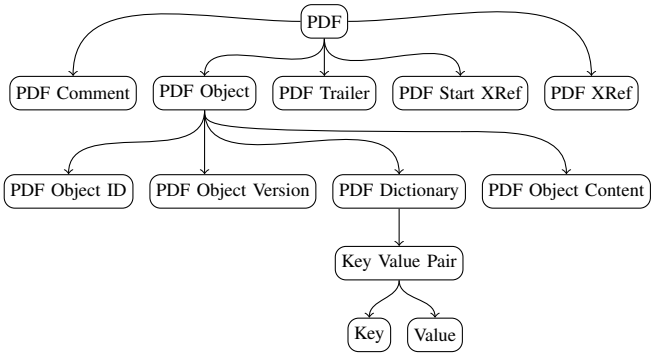

Fig. 3: Dominator tree of the simplified type composition (*i.e.*, the syntax tree of Figure 2) for a small PDF. This is calculated in Algorithm 1 lines 25 and 26.

ground-truth labeling of the PDF using PolyFile (step 1) required 1.2 seconds of CPU time, rendering the PDF with instrumented MuPDF (step 2) required 0.2 seconds of CPU time, and merging the results (step 3) required 95.2 seconds of CPU time—in total, less than 100 seconds of CPU time.

Despite running an order of magnitude longer than the first two steps, the merging algorithm of step 3 does in fact run in polynomial time; the apparently large runtime is an artifact of constant factors. To see this, observe that the interval tree will have $|E|$ nodes, where E is the set of semantic elements in the input file. Let n be the number of bytes in the input file, and observe that $|E| \leq n$. Therefore, the interval tree can be constructed in $O(n \log n)$ time. Lookup for a single byte offset to the set of intervals in which it exists takes $O(\log n)$ time, so mapping all functions to the set of elements on which they operated will require worst-case $O(|F| n \log |E|)$ time, where F is the set of all parser functions instrumented in the program. Since the number of bytes in the input file will almost always be greater than the number of functions in the execution, $n \gg |F|$, the entropy calculation and dominator tree refinements will run in linear time. Therefore, the worst-case runtime of the merger algorithm of step 3 is $O(|F| n \log |E|)$.

As another example, Figure 5 is the mapping resulting from parsing a single JPEG through an instrumented version of the ubiquitous `libjpeg` library. While having more defined types than PDF, it too produces an intuitive mapping. For example, the `jpeg_make_d_derived_tbl` is correctly identified as being responsible for processing the `huffman_table` type in the JPEG specification. Likewise, the `latch_quant_tables` function is identified as specializing in operating on the `quantization_table_id` type of JPEG segment components.

IV. Related Work

This is not the first effort toward discovering the relationship between a format and its parser. The work of Lin, Jiang, Xu, and Zhang [13], [14] has demonstrated that a hybrid approach like ours, utilizing both static and dynamic analysis, is viable.

A format analyzer, afl-analyze [15], is built on the popular fuzzer AFL [16]. It uses AFL to individually mutate bytes in an input stream to observe how changes to these bytes affect control flow. When an input byte changes control flow, afl-analyze attempts to classify the byte's type based on how it is used throughout the program's execution. For example, in an image parser, bytes that represent pixel data do not alter control flow, and so they can be classified as "data" bytes, whereas bytes that *do* change control flow might be classified as length fields, magic numbers, headers, *&c.* Unlike our work, afl-analyze appears to work best with mostly static, binary, non-recursive, and non-dependently typed formats. Also, afl-analyze does not take advantage of any ground-truth knowledge about the input file; it is designed to reverse engineer a *file format* given a parser, whereas we desire to reverse engineer a *parser* given knowledge of the file format.

ProFuzzer is another fuzzer that attempts to learn more about an input [17]. ProFuzzer, like afl-analyze, monitors

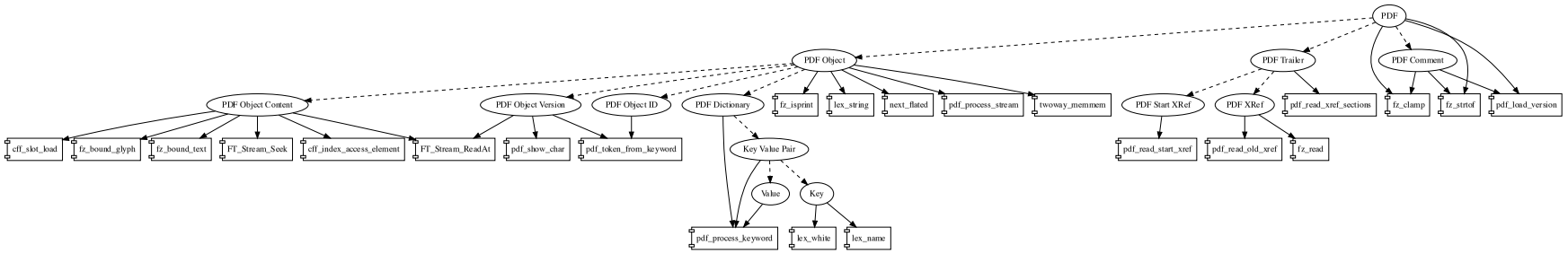

Fig. 4: Mapping produced from a single, relatively small PDF run through an instrumented version of `mutool` from the MuPDF project. Semantic types are round nodes and parser functions are rectangular. Solid edges represent the mapping of semantic types to functions, and dashed edges represent the extracted type hierarchy/syntax tree.

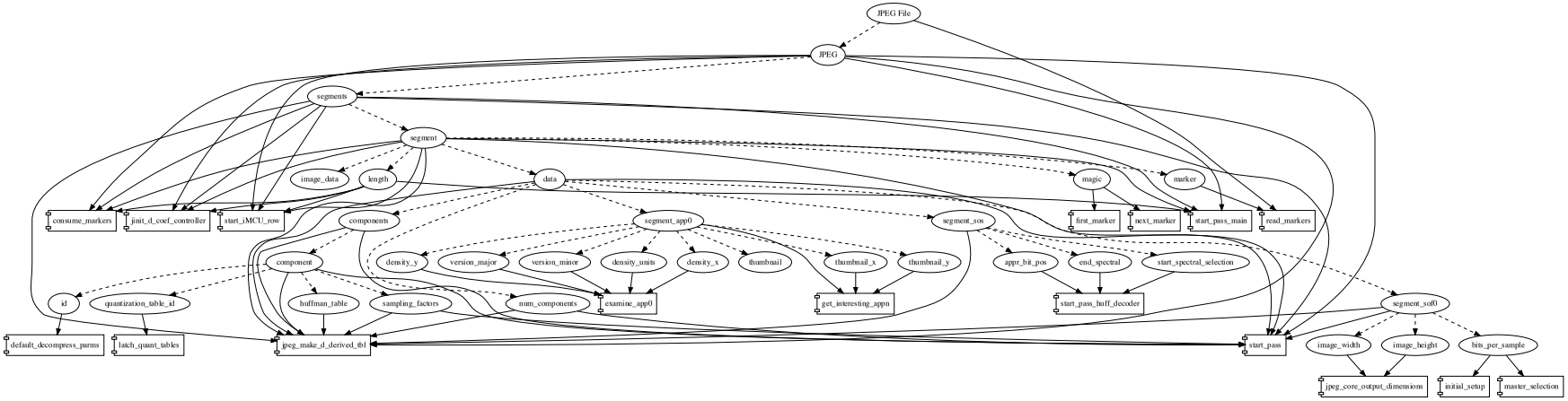

Fig. 5: Mapping produced from a single JPEG image run through an instrumented version of `libjpeg`. Semantic types are round nodes and parser functions are rectangular. Solid edges represent the mapping of semantic types to functions, and dashed edges represent the extracted type hierarchy/syntax tree.

the fuzzing process to see how mutations affect program execution. Unlike afl-analyze, ProFuzzer avoids mutating data segments and attempts to focus on discovering input fields that are important to fuzzing. For example, if ProFuzzer's type probing identifies bytes that represent an input size, ProFuzzer will mutate at the field level instead of the byte level. By avoiding mutations on data segments, and by mutating entire fields instead of individual bytes, ProFuzzer greatly reduces the number of unproductive mutations. Overall, ProFuzzer's mutation strategy provides 60% more code coverage than afl-analyze, and less false positives for inferred fields.

ProFuzzer [17] and related fuzzers including Grimoire [18], NEUZZ [19], Learn&Fuzz [20], GLADE [21], and REINAM [22] use what they learn about the input to make better mutations. Each fuzzer has their own internal representation of the input, and that representation is not exported, reusable, or human readable.

AUTOGRAM [8] uses dynamic taint tracking, a grammar inference algorithm based on interval trees, and program instrumentation to take a parser implemented in Java and derive a human-readable context-free grammar describing the inputs accepted by the parser. In experiments, the AUTOGRAM system is able to infer context-free grammars for simple JSON, CSV, and INI configuration format parsers, with varying degrees of accuracy and completeness (*i.e.*, measures of how much the inferred grammar *overgeneralizes* and *overspecializes*). AUTOGRAM does make a few assumptions about parser implementations, necessitating heuristics and concessions such as ignoring data flow induced by parser lookahead. This limits AUTOGRAM's applicability to a certain class of parser implementations, as well as a certain class of grammars.

Mimid [23] generalizes the approach used in AUTOGRAM, eliminating several assumptions about parser implementation patterns, and eliminating parser-specific heuristics. Unlike AUTOGRAM, which operates on Java parser implementations, Mimid operates on Python parser implementations. Also unlike AUTOGRAM, instead of using dataflow-based taint tracking, Mimid analyzes the dynamic control flow of the parser implementation to infer a context-free grammar. In experiments, Mimid has better measures of both accuracy and completeness compared to AUTOGRAM, and supports a wider class of parser implementations, such as PEG parsers [24].

V. Conclusions and Future Directions

In this paper we have introduced two research prototypes for collecting ground truth information about file formats and their associated parsers. We presented a new and novel algorithm for mapping semantic hierarchies to a parser's control-flow graph. This work is the first step in combining a data-flow analysis similar to AUTOGRAM with the control-flow analysis of Mimid, and extending it by incorporating compositional type information from labeled ground truth input. This additional semantic context will aid automated grammar extraction in handling traditionally challenging language features like dependent types. Ultimately, we hope that this will lead to support for more complex, non-context-free file formats like PDF and HTML. PolyFile and PolyTracker, like Mimid, are designed to be interoperable with other tools. Currently, we are able to automatically generate a semantic function labeling, and are in the process of extending this to produce a grammar that can be consumed by other tools and algorithms.

For complex formats like PDF, each parser effectively implements its own dialect of the format, *e.g.*, due to ambiguity in the specification. The next phase of this research will focus on extracting grammars from multiple parsers implementing the same file format. This will allow us to intersect the grammars to obtain a subset of the file format that is mutually intelligible to all implementations. This grammar is the canonical grammar for the format, a minimal grammar specifying the inputs which are accepted by all parsers in a test corpus. This will lead to a simpler—and, we posit, *safer*—subset of the file format that can be used to generate verified parsers.

Performing our semantic labeling across multiple parser implementations of the same file format can also provide a means for differential analysis. If we have semantic mappings extracted from *two* different parsers (like the one pictured in Figure 2), a many-to-many graph matching algorithm can be used to map the functions of each parser to each other, immediately identifying feature differences between the implementations. For example, we would expect Adobe Acrobat to have a set of functions specialized in operating on JavaScript in XFA Forms that would not map to any functions in Poppler, which does not support JavaScript. Likewise, two different releases of the same parser could be compared to each other to determine what features were added or removed between the versions, and which functions were affected.

Similar to applying the method across multiple parser implementations, we can also benefit from combining the output of multiple files across a single parser. Using a single file is limiting because the code coverage of the parser will be limited to the features implemented in that file. For example, a JPEG that does not include any EXIF metadata will not exercise any of the metadata parsing functions within a JPEG parser. Therefore, we plan to extend the approach from operating on single files to instead operate on a corpus of files, merging the output.

In the event that there does not exist a ground truth labeling for the input file (*e.g.*, if the input file format is unknown), we will investigate treating each byte in the input file as its own unique type. We can then exploit the forest of taint unions produced by the instrumentation to *learn* the type composition hierarchy. This can also be augmented through introspection of how the bytes are stored in the data structures employed by the parser implementation.

There is also opportunity for research into the algorithm's sensitivity to its sole parameter: the standard deviation threshold (line 22 of Algorithm 1). We also plan to investigate increasing the resolution of the program instrumentation from the function level to basic blocks. We expect basic blocks to produce a more nuanced mapping from semantic types to code, and potentially address the problem of parsers implemented with monolithic functions.

In this paper, we have introduced a general approach for semantically labeling the functions of a parser by combining both static and dynamic analysis, as well as by automatically producing ground truth from the input. This is the first step in implementing automated grammar extraction from arbitrary, non-context-free parsers. We demonstrated that this approach is both computationally efficient and produces results that are intelligible to a human. A single execution of a parser is sufficient to produce a viable mapping. We hope that this algorithm will lead to further interest in automated grammar extraction, and ultimately produce safer file formats and parsers.

VI. Acknowledgments

This research was supported in part by the DARPA SafeDocs program as a subcontractor to Galois under HR0011-19-C-0073. Many thanks to Trent Brunson, Peter Goodman, Bill Harris, Eric Davis, Nichole Schimanski, Sergey Bratus, Ryan Speers, Peter Wyatt, Dan Kaminsky, and Tim Allison for their invaluable feedback on the approach and tooling.

References

[1] A. Albertini, "Abusing file formats; or, Corkami, the novella," *The International Journal of Proof of Concept or GTFO*, no. 0x07, pp. 18–41, Mar. 2015.

[2] S. Bratus, M. E. Locasto, M. L. Patterson, L. Sassaman, and A. Shubina, "Exploit programming: From buffer overflows to "weird machines" and theory of computation," *;login:*, vol. 36, 2011.

[3] "DataFlowSanitizer," accessed: January 12, 2020. [Online]. Available: https://clang.llvm.org/docs/DataFlowSanitizer.html

[4] P. C. et al., "DataFlowSanitizer design discussion," 2013, accessed: January 12, 2020. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2013-June/062877.html

[5] E. Sultanik, B. Larsen, and C. 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.

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

[7] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. USA: IEEE Computer Society, 2010, pp. 317–331.

[8] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 720–725.

[9] "Taintgrind: a Valgrind taint analysis tool," https://github.com/wmkhoo/taintgrind, accessed: March 2, 2020.

[10] J. A. Storer and T. G. Szymanski, "Data compression via textual substitution," *Journal of the ACM*, vol. 29, no. 4, pp. 928–951, Oct. 1982.

[11] T. Lengauer and R. E. Tarjan, "A fast algorithm for finding dominators in a flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–141, Jan. 1979.

[12] S. Garfinkel, P. Farrell, V. Roussev, and G. Dinolt, "Bringing science to digital forensics with standardized forensic corpora," *Digital Investigation*, vol. 6, pp. S2–S11, 2009, in Proceedings of the Ninth Annual DFRWS Conference.

[13] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic protocol format reverse engineering through context-aware monitored execution," in *Proceedings of the Network and Distributed System Security Symposium*, Feb. 2008.

[14] D. Xu, X. Zhang, and Z. Lin, "Reverse engineering input syntactic structure from program execution and its applications," *IEEE Transactions on Software Engineering*, vol. 36, no. 05, pp. 688–703, sep 2010.

[15] M. Zalewski, "Automatically inferring file syntax with afl-analyze," https://lcamtuf.blogspot.com/2016/02/say-hello-to-afl-analyze.html, Feb. 2016, accessed: January 12, 2020.

[16] ——, "American fuzzy lop," http://lcamtuf.coredump.cx/afl/, 2014, accessed: January 12, 2020.

[17] W. You, X. Wang, S. Ma, J. Huang, X. Zhang, X. Wang, and B. Liang, "ProFuzzer: On-the-fly input type probing for better zero-day vulnerability discovery," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019, pp. 769–786.

[18] T. Blazytko, C. Aschermann, M. Schlögel, A. Abbasi, S. Schumilo, S. Wörner, and T. Holz, "GRIMOIRE: Synthesizing structure while fuzzing," in *Proceedings of the 28th USENIX Security Symposium*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1985–2002.

[19] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "NEUZZ: Efficient fuzzing with neural program smoothing," in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2019, pp. 803–817.

[20] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE 2017. IEEE Press, 2017, pp. 50–59.

[21] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2017. New York, NY, USA: Association for Computing Machinery, 2017, pp. 95—-110.

[22] Z. Wu, E. Johnson, W. Yang, O. Bastani, D. Song, J. Peng, and T. Xie, "REINAM: Reinforcement learning for input-grammar inference," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, pp. 488–498.

[23] R. Gopinath, B. Mathis, and A. Zeller, "Inferring input grammars from dynamic control flow," *arXiv preprint*, December 2019, `arXiv:1912.05937v1 [cs.SE]`. [Online]. Available: https://arxiv.org/abs/1912.05937

[24] B. Ford, "Parsing expression grammars: A recognition-based syntactic foundation," in *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '04. New York, NY, USA: Association for Computing Machinery, 2004, pp. 111–122.