Corpus-wide Analysis of Parser Behaviors via a Format Analysis Workbench

Pottayil Harisanker Menon*
Galois, Inc.
hari@galois.com

Walt Woods*
Galois, Inc.
waltw@galois.com

***Abstract*—As the number of parsers written for a data format grows, the number of interpretations of that format's specification also grows. Often, these interpretations differ in subtle, hard-to-determine ways that can result in parser differentials – where one input passed to two parsing programs results in two semantically different behaviors. For example, two widely-used HTTP parsers have been shown to process packet headers differently, allowing for the exfiltration of private files. To help find, diagnose, and mitigate the risks of parser differentials, we present the Format Analysis Workbench (FAW), a collection of tools for collecting information on large numbers of parser/input interactions and analyzing those interactions to detect and explain differentials. This tool suite supports any number of file formats through a flexible configuration, allows for processing to be scaled horizontally, and can be run offline. It has been used for results including the analysis of more than 1 million PDF files and unifying parser behaviors across these files to identify a gold standard of validity across multiple parsers. The included statistical tools have been used to identify the root causes of parser rendering differentials, including mislabeled non-embedded fonts. Tools for instrumenting existing parsers are also included, such as PolyTracker, allowing for the analysis of blind spots which might be used to craft differentials for other parsers, or to exfiltrate large quantities of data. Through allowing users to characterize parser behaviors at scale against large corpuses of inputs, the FAW helps to mitigate security risks arising from parser behaviors by making it tractable to resolve examples of differentials back to their behavioral causes.**

I. Introduction

As data formats age, they tend to become more complex as convenience features are added. In turn the programs used to parse those formats also become more complex, particularly as multiple different parsing programs are developed. The authors of these programs are likely to make different decisions regarding the interpretation of the format specification, which can lead to dangerous parser differentials. Parser differentials arise when two parsers interpreting identical inputs produce semantically different behaviors [1]. These behavioral differences can be exploited by attackers to access private files or take control of hardware. For example, a recently documented HTTP parser differential[1] demonstrated how a well-crafted packet with disagreeing `Content-Length` and `Transfer-Encodering` fields results in an HTTP request that passes an outer gateway and leaks intranet files. Once found, these vulnerabilities are easy enough to patch, in this case by having the gateway rewrite packet headers that disagree. However, finding these vulnerabilities can be quite challenging given the scale of both parser program behaviors and the possible inputs that might break them. Fuzzing is an effective pre-hoc testing framework for finding potentially dangerous inputs; we instead focused on post-hoc analyses that might help analysts find actively exploited parser differentials, and help identify the root causes of those differentials to help format authors learn to avoid those root causes in the first place.

A. Motivation

Files or other inputs that actively exploit a parser can be difficult to find in the wild. While there have been efforts to collect millions of files from available data sources [2], sorting through these files to identify parser differentials or exploits remains a time-consuming task. Our goal has been to turn something that previously took weeks into mere hours or days: to help a format analyst go from a vague idea of an exploit or differential to concrete evidence and an understanding of how that exploit compares to millions of other files.

Often, a format analyst must decide which kinds of differentials or exploits they are looking for, and produce ad hoc tools that look for these signs. Next, after these tools have been run against the corpus, it is likely that these tools must be tweaked and re-ran against the corpus. These round-trip queries against the collection of parser inputs can take days or weeks, motivating a need for tooling to accelerate these loops, and make the manual parts of the process more automated.

In practice, we additionally found that considering the outputs of multiple parsers could bring additional context to each others' errors, or to hits from ad hoc tools looking for differentials or exploits. As a direct result of the scaling desired when running these ad hoc tools to find potentially insecure files in the wild, it became clear that large corpuses were also rich in statistical correlations, allowing for more concrete discoveries that would help explain the various outcomes from parsers and tools for identifying parser security violations. This would also enable, at scale, the integration of various taint tracking or other instrumentation techniques that would shed additional light on parser behaviors.

By integrating all of these tools and techniques into a single workbench, users are empowered to create new tools, compare them to extant parsers for formats, and compare/contrast

*These authors contributed equally to this work.
[1]https://portswigger.net/research/browser-powered-desync-attacks

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00024

© 2023, Walt Woods. Under license to IEEE.

DOI 10.1109/SPW59333.2023.00024

implementations, across millions of files, without needing to manually run or integrate much information.

II. Contributions

While designing our approach, it was important to refine a list of capabilities that should be covered. To ensure utility for the analysis of parser/input interactions, and to qualify interactions of interest, we identified several additional use cases:

- *When debugging an error within a single parser, it would be convenient to find other inputs that trigger the same error.* This could be solved by keeping a database of parser/input interactions, and providing a means of looking up similar inputs based on observed errors.
- *It must be possible to find disagreements amongst multiple parsers.* Therefore, any movement toward a solution in this space would be able to store a parser/input cross product, and allow for the comparison of validity or other qualities across parsers.
- *Once a error or differential is found, it would be convenient to have tooling to help narrow down the root cause, ideally all the way down to the grammar level.* Particularly for formats with data-dependent parses, modern parsers can exhibit very complicated interactions amongst bytes in the input. By distilling these interactions to smaller root causes – ideally, as a succinct grammar that perfectly captures inputs triggering the error or differential – users could be presented with enough information to locate source code connected to the issue and construct minimal test cases to verify a fix.
- *The stdout/stderr of a parsing program would probably be insufficient to find these root causes; that is, deeper features are needed.* By leaning on modern technologies such as taint tracking or grammar inference, we could greatly expand the resolution of proposed root causes, allowing for tooling that would continue to improve as these methods improve.

To provide capabilities for all of these use cases within a shared framework, we present the Format Analysis Workbench (FAW)[2], a publicly available, open-source workbench for file format analysis. While the intent of these use cases all live in a similar vein – diagnosing extant risks and helping to mitigate future risks – multiple distinct analysis approaches tended to complement one another. As such, the FAW is an easy-to-extend framework and consolidated user interface (UI) for input corpus analysis. Its architecture and a selection of high-level capabilities are shown in Fig. 1.

The FAW enables corpus-wide analysis of features both through novel technical ideas for finding relationships amongst features, and in providing a workbench for applying these techniques to a user's corpus, analyzing results, and prototyping new analyses or parsing programs.

[2]https://github.com/GaloisInc/FAW

A. Data model

Broadly, we adopted the view that binary features are a well-suited, general way of storing information concerning the interactions between a parser and its input. Is the input document of type PDF1.7? Did the program crash when trying to parse its input? Was an error message about a malformed header printed? Was a specific function called within the program? Then, understanding interactions between a parser and its input became a question of associating these specific binary features: e.g., perhaps PDF 1.7 causes malformed header messages, indicating an implementation bug. This generalized insight was broadly owed to work from Ambrose *et al.*, who argued that this view of program/input interactions constituted a topology [3].

In general, we found that no single technique or method of dealing with these features was optimal for all use cases. Therefore, the extensible design of the workbench, coupled with the flexibility of the underlying data model, provides users with a suite of tools for navigating and better understanding complex relationships between program behaviors and specific input patterns. The FAW has enabled an unprecedented breadth of results within Defence Advanced Research Projects Agency (DARPA)'s Safe Documents (SafeDocs) program; those results are summarized in Section III, with additional technical details in Section IV.

III. Results

To demonstrate the utility of this unified framework approach, we highlight several results relevant to DARPA's SafeDocs program. These results were all achieved through the technologies included in the FAW.

Offline, parallel processing enabled ingestion and analysis of more than one million files. The processing time of the FAW is highly dependent on its configuration and the input format being explored. Heavy workloads benefit from the distributed computation management handled by the FAW, which allows multiple computers to work together when populating the parser/input cross-product database, without requiring access to the internet. For example, we ran 20 separate parsers against 1 040 628 PDF files in about 13 days, leveraging 96 cores spread across 4 separate machines. Some of these parsers – such as a FAW-specific test for rendering differentials between MuPDF and pdftoppm[3], which takes about 10 seconds per file on its own.

On a 10 000 subset of that corpus, the parsers included result in 320 717 novel binary features; novel features scale sub-linearly to corpus size. Some of these are stored as real-valued key/value pairs in the database, which are thresholded to translate them into binary features suitable for the included analysis tools.

Achieved parity with a gold standard of input validity. The SafeDocs goal for being able to detect inputs that were not valid PDF according to another team's generated gold standard was 0.01% false positives with as low of a false negative

[3]https://mupdf.com/ and https://poppler.freedesktop.org/

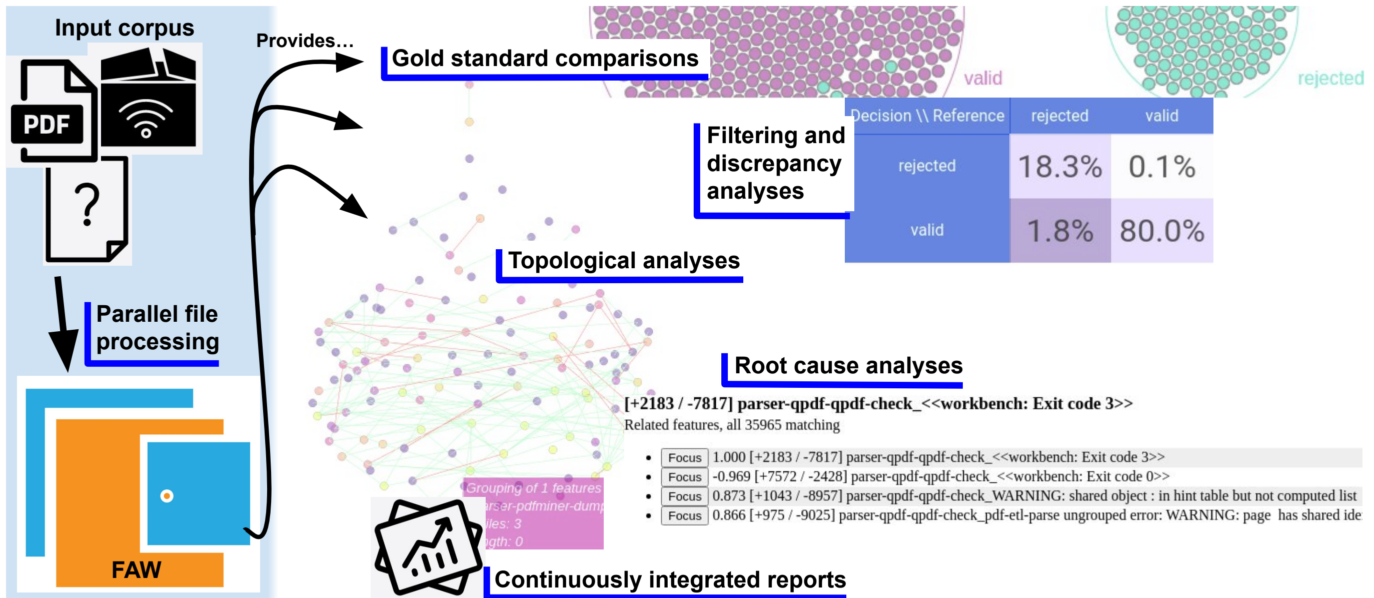

Fig. 1: An overview of the FAW. A parallelized backend populates a database with the cross product of all inputs with all available parsers. Multiple UI tools are provided to aid users in understanding parser behaviors, and can be used for gold standard comparisons, topological analyses, filtering and analyzing discrepencies, finding root causes, and generating continuously up-to-date reports.

rate as possible. The FAW's tools allowed for us to rapidly piece together alarm logic that matched the gold standard with 0% false positives and 0.5% false negatives. Achieving these numbers required about 2 hours of analyst time. The regular expressions derived to achieve these results were relatively complex, such as this one which finds unembedded fonts that are not of a family commonly embedded in operating systems:

```
^parser\-xpdf\-pdffonts_FONT(?!
↪  Arial[-,]).*EMB no.*(?<!/(arialbd|[Cc]
↪  our\S*|[Hh]elve\S*|[Tt]imes\S*)\.ttf)$
```

Building this expression was relatively quick due to the efficacy of the confusion matrix filtering included in the FAW. If a user would like to find parser features not expressible with regular expressions, then wrapping those parsers in a basic script to convert them to a more easily searchable format is supported.

Created a new distribution for analyzing NITF files in 10 minutes through developer-friendly infrastructure. Designed to also be useful for proprietary and private formats, the FAW was extended with a new NITF distribution implementing two parsers with only 10 minutes of developer time. This was supported through continuous rebuild-and-reevaluate functionality built into the FAW. Multiple formats are included out of the box, including iccMAX, NITF, and PDF, while private distributions leveraging non-public parsers have been produced for MavLink, netcap, RTPS streaming data, DDS, and JPEG.

Proven ability to identify root causes for generic error messages. Parser error messages can be confusing or incomplete. To help analysts rapidly find root causes from these vague messages, the FAW includes statistical tools for finding IMPLIES relationships amongst error messages, further detailed in Section IV-D. For example, the popular QPDF[4] parser has a generic "file is damaged" error. Without looking at the source code, this message does not imply any specific cause. The tools included with the FAW show that this error message is implied by MuPDF's "cannot find startxref" and "object missing 'endobj' token" errors, helping ground expectations around what QPDF considers a damaged file.

Identified causes of significant parser differentials between PDF renderers. Through a number of filters, an image comparison tool was developed that flagged 422 files (out of 10000) with significant visual differences when rendered through the MuPDF and pdftoppm programs. Via the statistical tools previously mentioned, and by pulling in data from other PDF parsers, we were able to trace the rendering differentials to a number of root causes, primarily embedded fonts that were labeled incorrectly. For example, multiple fonts were marked with the font type of "Type," and all files exhibiting this behavior resulted in a differential.

Identified polyglot files via PolyFile integration. PolyFile[5] is a utility for inspecting the semantic structures of files, and includes the ability to detect polyglots through the more than 260 included parsers for different file types. By pairing this with the infrastructure provided by the FAW, we were able to find 24 polyglot files within a 10 000 file corpus.

Found common blind spot triggers, where a bad actor might exfiltrate data or craft parser differentials, via PolyTracker integration. PolyTracker[6] adds taint-tracking capabilities to existing parsers via a recompilation step. Part of this tooling includes the ability to locate "blind spots," or bytes

[4]https://github.com/qpdf/qpdf
[5]https://github.com/trailofbits/polyfile
[6]https://github.com/trailofbits/polytracker

where most values would not affect the outcome of parsing the input [4]. By combining this with the FAW's tools for finding root causes, we were able to identify root causes for some commonly ignored bytes. For example, the `/Identity-H` encoding in PDFs is often created by the Joomla! CMS, and permits users to exfiltrate whatever information they'd like without affecting the rendered output.

Provided ability to run state-of-the-art grammar inference on any corpus, yielding byte patterns that are more discriminative than n-grams. The ideal outcome of root cause analysis would be the capability to point to specific grammar elements or byte payloads that trigger a specific behavior or error message, at scale and across an entire corpus. To that end, we have experimented with including the Reinforcement Learning for Grammar Inference (RL-GRIT) algorithm [5] in the FAW, allowing users to look at specific parts of their input corpus and automatically generate annotate inputs with byte patterns that are likely to trigger behavior in a parser. While the underlying technology needs to be further developed, we have been able to draw links between e.g. Javascript errors in PDF files and the inclusion of the `/Url` key. By expanding these capabilities, we plan on moving further toward being able to identify specific grammar elements that trigger parser behaviors of interest.

IV. Capabilities

Understanding an input corpus at scale requires tooling for running that corpus through one or more parsing programs that can output details about the input's interaction with that program. The FAW consists of a backend, which performs all of the heavy lifting for the parser/input cross product, and a frontend for understanding the interactions between the input corpus and parsers of interest (Fig. 1).

The backend generates and aggregates results from the cross product between a corpus of inputs and a population of parsers. Parsers only require an executable binary file, and can be written in any language. This processing can happen on a single machine or in parallel, using what the FAW calls a "teaming" setup. Teamed processing can happen on a private network without external internet access.

Once this parser/input cross product is generated, or even once it is partially populated, the FAW allows the user to specify *analysis sets*, or groups of files that should be examined together, as a way of controlling the scale (and thus speed) of the user's analyses. To help users understand the vast output from processing an input corpus against all parsers, users are provided with multiple distinct tools for exploring the interactions between inputs in the analysis set and the parsing programs that were run against those inputs.

Following are specific technologies and capabilities contained within the FAW; unless otherwise noted, similar results are all available out of the box for a user-provided corpus, without additional work from the user.

A. Easy to get started

An instance of the FAW can be launched by first cloning the FAW repository and then invoking the startup script `workbench.py` with two paths, one to a *distribution* folder and another to the file corpus being studied. A FAW distribution is essentially a specific configuration of the FAW and typically associated with a specific file format or corpus. Section IV-C1 describes distributions, their content and how to make your own in more detail.

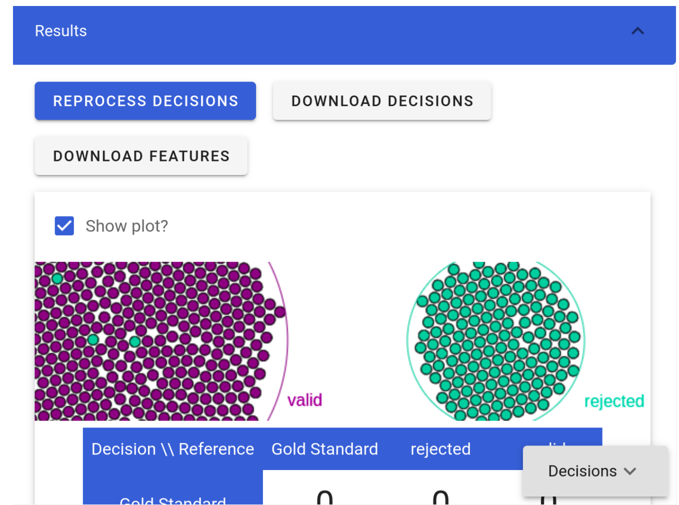

Fig. 2: A snapshot of the FAW UI.

The FAW repository contains example distributions for a number of extant input formats, including PDF. These can be used as-is or as a template when creating new distributions. The repository includes small test corpora as well, to allow for easy experimentation. For example, an instance of the included PDF distribution and test corpora can be launched from the root directory using the command: `./workbench.py pdf test_files/pdf`. The workbench script builds a docker image for the specified distribution on the fly and launches a FAW instance that can then be accessed via a web browser pointed to the `localhost` and customizable port (default: `8123`). Figure 2 shows a snapshot of the FAW UI after it starts up and the user clicks "Reprocess Decisions."

Additionally, the workbench script is capable of pre-building and saving a single, unified docker image for a FAW distribution. In addition to running locally, the FAW can also be deployed in a multi-server "teaming mode", powered by pyinfra[7], which can be particularly useful when working with large corpora. The repository contains the `workbench-teaming-init.py` script which can be used to create the necessary pyinfra inventory and deploy files, which can then be used to automatically install the prerequisites on target machines, deploy FAW images and set up a FAW system service on each machine. In this mode, the target machines process files independently and in parallel, but share a single database, with the FAW UI accessible via a webserver running on the database's machine.

[7] https://pyinfra.com/

B. Filtering and discrepancy analyses

FAW distributions include file parser plugins to parse files and extract features from it; additional details on specifying these are in Section IV-C2. While it may be possible to determine the validity of a file by perusing the features manually, at least for small corpora, it is often more convenient to have these decisions made automatically based on user-supplied criteria. The FAW supports a simple, custom domain specific language (DSL) for expressing decision criteria and provides a user interface to quickly edit and reprocess the decisions.

The DSL allows decision criteria to be specified in terms of *filters* and *outputs*. Filters are themselves defined as groups of regular expressions and are matched against features generated by parsers; if any of the features generated for a specific file match any of the filter criteria, the file matches the filter. Outputs represent aggregations of filter criteria and are defined in terms of compound boolean expressions over filters. Listing 1 shows a snippet of a decision DSL that defines two filters based on the features emitted by the `mutool clean` command: `MuCleanAccept` accepts a file as long as `mutool` exited with a zero exit code, while `MuCleanError` accepts a file if it *failed* to exit with a zero exit code *or* it emitted a feature that matches the specified regular expression: `.*[Ee]rror(?!: .*marker)`. The snippet also defines an output: the status is set to be "valid" if and it satisfies the conditions specified by `MuCleanAccept` and does not satisfy those specified by `MuCleanError`; otherwise the status is set to "rejected".

Adjusting the decision DSL also influences the FAW UI directly. Filter and output clauses in the DSL, like `status` and `MuCleanAccept` from the previous example, will appear in the UI as radio buttons. When selected, these filter the file list to show only those files that satisfy the criteria specified in the DSL.

In situations like regression testing or working with a known file corpora, the expected features and status (valid/invalid) associated with each file in the corpus may already be known. The FAW allows users to upload reference decisions as structured JSON files. These "gold standard" decisions are then automatically compared against those made by the current decision DSL, with discrepancies highlighted to allow users to directly investigate the specific files that failed to match expectations.

While these regular expressions might seem time consuming to craft, the FAW's core search UI and various plugin-based capabilities are designed to make it a relatively quick process. Achieving our state-of-the-art results on a large PDF corpus only took a developer familiar with regular expressions about 2 hours.

C. Extensible

As a general purpose tool for understanding corpora, the FAW is expected to work with a diverse set of input formats, format-specific parsers, and visualization tools. Instead of attempting to provide a monolithic set of tools, the FAW uses a plugin architecture with well-defined extension points and a plugin API that allows users to create custom plugins that fit their specific corpus and workflow. For example, when working with a new file format, it is possible to create plugins that parse the format and/or visualize data from inputs in the corpus. The primary types of plugins supported by the FAW are described in the following sections.

```
filters:
  MuCleanAccept:
    ^mutool-clean_.*<<workbench: Exit code
     ↪  0>>
  MuCleanError:
    ^mutool-clean_.*<<workbench: Exit code
     ↪  (?!0)
    ^mutool-clean_.*[Ee]rror(?!: .*marker)
outputs:
  status:
    "valid" is MuCleanAccept &
     ↪  !MuCleanError
    "rejected" else
```

Listing 1: Decision DSL

1) Creating a new distribution: A distribution is typically created as a separate folder under a cloned FAW repository. The distribution folder usually contains a top-level `config.json5` file as well as a collection of plugins. By convention, each plugin is placed in a subfolder of its own and with its own configuration file. However, this is not mandatory and the configuration files are merged with the top-level configuration before any processing. In addition to any plugin-specific configuration, these files can also specify the process to build and install any dependencies that might be needed at runtime; this process can be as simple as installing a system package or as complex as recreating a full development environment and performing a multi-stage build. Further, it is possible to restrict the set of files/binaries that are copied to the final distribution image allowing for a clean separation between the build process for a plugin and the binary artifacts necessary to execute it at runtime.

When the `workbench.py` script is invoked with the path to a distribution folder, the configuration files for that distribution are used to construct a Dockerfile, which is then built and launched as a container running the FAW instance. The container includes the database, frontend, and backend processes required; while a single container potentially causes build dependencies, the ease of distributing the single image underlying the container across networks has proved advantageous for getting users up and running.

Listing 2 contains a simple configuration file (with some fields elided for conciseness) for an example CSV distribution. The `build` subsection of the configuration describes the installation process for the simple CSV validator described in Listing 3: the script is copied in to the docker image with the `ADD` command and the `copy_output` specification copies it to its final location in the image. While unnecessary for this

```
{
  name: 'galois-workbench-csv',
  parsers: {
    csvvalidator: {
      exec: ['csvvalid', '<inputFile>'],
      version: '0.1',
      parse: {
        type: 'regex-counter',
        version: '1',
        stdout: {
          '(.*)': {
            nameGroup: 1,
          }
        }
      }
    }
  },
  build: {
    stages: {
      base: {
        from: 'ubuntu:22.04',
        commands: [
          'RUN apt-get update && apt-get
          ↪  install -y python3'
        ]
      },
      csvvalidator: {
        commands: [
          'COPY ./csv/csv_validator.py
          ↪  /tmp/csv_validator.py'
        ],
        copy_output: {
          './tmp/csv_validator.py':
          ↪  '/usr/bin/csv_validator'
        }
      }
    }
  }
}
```

Listing 2: Configuration file for a CSV distribution

simple configuration, the latter specification is beneficial when the build process is complex, but only a small number of build artifacts are required to execute at runtime. The `parsers` section contains a specification for a parser based on the validation script above: it specifies the command line to run the validator as well as the mechanism to map the validators̈ output to parser features; in the current case, all output lines are captured directly and stored as features.

2) Parser plugins: The FAW provides support for parser plugins that can both validate file formats as well as help to surface format specific features for later examination. When invoked, such plugins execute an external command to parse the specified file. The FAW automatically captures the exit code as well as the standard output/error from the execution

```
import sys
import csv

with open(sys.argv[1], newline='') as f:
  r = csv.reader(f)
  count = len(next(r))
  for row in r:
    assert len(row) == count
```

Listing 3: Simple CSV validator

and offers the ability (via the configuration file) to map these to validation status and/or human-understandable features stored within the FAW database. This mapping is specified using regular expressions, and typically takes the form of a named capture group with the ability to modify it via a regex replacement specification or capture as a count. The CSV validator in Listing 3 is an example of a parser plugin.

3) File detail plugins: File detail views are plugins that visualize the contents of a file (or a subpart of it) by rendering it as HTML or another browser-displayable content format. As with parser plugins, file detail view plugins are configured to run an external command that takes a file as input and streams out the transformed content to the standard output. The FAW captures this input and renders the content in the browser using the specific MIME type provided in the configuration. One prominent file detail view is provided by PolyFile[8], which can turn most file formats into an interactive HTML report that allows users to view and search data structures contained within the file.

4) Decision / corpus-level plugins: Unlike the previous plugins which operate on individual inputs in a corpus, decision plugins operate on the whole corpus and therefore can aggregate and visualize information concerning the dialects or common subpatterns within the corpus' format. As with other plugins, launching a decision plugin involves running a configured external command. However, as one of the most general types of plugins available in the FAW, these commands can potentially access the complete corpus, as well as the features captured during parser invocations for each file. Further, it is also possible to invoke these plugins with additional user-supplied arguments, which are then made available to the external command.

The command is expected to stream its results in JSON line format where each object corresponds to a file in the corpus and contains a series of key/values; the data is captured and stored by the FAW. The FAW offers an additional level of flexibility by allowing these commands to optionally create an custom HTML file at a FAW-specified location; if such a file is produced, the FAW will capture the contents and render it as part of its UI. This enables the plugin to insert entirely new UI elements and actions (via embedded Javascript) in to the FAW UI. The root cause analyses (Section IV-D) provided with the FAW are implemented as decision plugins.

[8]https://github.com/trailofbits/polyfile

5) Pipeline plugins: The FAW allows for complex plugins that involve multiple stages, and can even generate new, data-driven parsers based on inputs from the corpus. See Section IV-G for an example.

6) Transforming inputs and universal output parsers: Parser and file detail plugins described in the previous sections work on individual inputs provided to them by the FAW. By default, the FAW passes individual inputs as-is directly from the corpus to the plugin(s). However, it also provides the ability to dynamically transform inputs before they are processed.

Input transformers are configured to run an external command that accepts a file as an argument and emits the transformed content on the standard output, which is then passed to parsers and plugins. These transformed inputs are treated as *transient* and are not permanently stored on the file system. This allows for the mechanism to work well with transformers that significantly expand the seed input, including input fuzzers like zzuf.

The FAW also supports a related notion of universal parser parsers, which operate on the output of every configured parser and produce additional features for the inputs. They are configured to run an external command that accepts a specially formatted stream on its standard input; the formatted stream includes the name of the parser that triggered this transformer invocation as well as the contents of standard output and standard error emitted by the external command corresponding to that parser. The command is expected to emit a JSON dictionary of new features to the standard output. By invoking a universal parser parser on the output of each other parser, special considerations for a format that apply to all parsers may be unified at a single implementation point.

D. Root cause analyses

It was previously proposed that groups of messages in files could be viewed topologically [3], [6], with the resulting structure providing mathematical relations that could be analyzed to reveal logical sets of related files [7], [8]. This can be visualized as a Dowker plot, which is included with the FAW in a "Dowker Visualization" plugin.

This topological work is excellent at segmenting the input corpus based on minimal differences – that is, it draws edges between file clusters which are closely aligned in feature space. On the other end of file format analysis for the SafeDocs program, many of the questions that needed answering were related more to specific features than file clusters: what caused a given exit code or parser differential? In the extreme, a best-case solution to this would be a grammar succinctly and perfectly matching only inputs that would trigger the condition. Grammar inference being a difficult problem, the next best approache seemed to be finding other features that robustly implied – or at least greatly contributed to the likelihood of – a desired condition.

To achieve this, we turned to the absolute risk reduction metric [9]. For our use case, we adapt it as $ARR(A, B) = P(A|B) - P(A|\neg B)$, where A and B are two features of interest. Conveniently, this metric becomes zero when the two features are independent, 1 when they are identical, and -1 when they are inverted ($A = \neg B$). To filter out noise, we additionally do not consider features with $min(P(A), 1 - P(A)) < \epsilon$. As the metric is asymmetric, a trick we use is to take $ARR(A, B)$ iff $|ARR(A, B)| > |ARR(B, A)|$, and $ARR(B, A)$ otherwise. The result is that IMPLIES relationships have large values, regardless of the direction of implication.

[+433 / -9567] parser-qpdf-qpdf-json_pdf-etl-parse ungrouped error: WARNING: <inputFile>: file is damaged
Related features, all 1447 matching

- Focus 1.000 [+433 / -9567] parser-qpdf-qpdf-json_pdf-etl-parse ungrouped error: WARNING: <inputFile>: file is damaged
- Focus 1.000 [+433 / -9567] parser-qpdf-qpdf-json_pdf-etl-parse ungrouped error: WARNING: <inputFile>: Attempting to reconstruct cross-reference table
- Focus 0.988 [+318 / -9682] parser-qpdf-qpdf-json_pdf-etl-parse ungrouped error: WARNING: <inputFile>: can't find startxref
- Focus 0.988 [+314 / -9686] parser-mupdf-mutool-clean_error: cannot find startxref
- Focus 0.971 [+145 / -9855] parser-mupdf-mutool-clean_warning: object missing 'endobj' token
- Focus 0.968 [+120 / -9880] parser-qpdf-qpdf-json_WARNING: EOF after endobj
- Focus 0.965 [+333 / -9667] pdf-hs-driver_ERROR: Couldn't find EOF

Fig. 3: Excerpt from the "Clustering" plugin showing risk factors for a "file is damaged" warning on a corpus of 10 000 PDF files.

For diagnostics, this simple metric is shockingly powerful – if, e.g., exit code Z only occurs on files which produce error message Y, then exploring either of those features will reveal the other with a large associated risk metric.

This functionality is exposed through the "Clustering" decision plugin in the FAW. Figure 3 shows what the results look like for the "file is damaged" message from the Section III. To make traversal easier, the plugin allows users to focus on any message shown in the plugin, which then shows messages related to that message.

E. Rendering differentials

For precise rendering formats, such as PDF, parser differentials manifest as semantic differences when rendered on different parsers. For example, a paragraph's text might not read the same, or an image might be missing some key detail. A rendering differential detection algorithm specifically for document content was developed and included in the FAW as part of the SafeDocs program. Existing algorithms often reported single pixel offsets or slightly different output color selections as large differences. Our algorithm was designed to reduce these false positives, only flagging files that demonstrated true differentials, allowing for the root causes of such differences to be identified.

The new algorithm, expressed in the "img_diff" function, is available online[9] and follows Algorithm 1.

[9] https://github.com/GaloisInc/FAW/blob/master/pdf/schizo_test/main.py

Algorithm 1: Image differencing algorithm for detecting semantic differentials

input : Two images A, B
output: $\delta =$ A distance measurement; large values indicate differentials

1. $A, B \leftarrow$ trim 1 pixel off all edges ; `/* PDF renderers treat the border differently; fix that */`
2. $B \leftarrow$ align B to A using a geometric transform [10] ; `/* Fix other layout differences */`
3. $A, B \leftarrow hipass(A), hipass(B)$; `/* Apply 1-pixel high pass filter to minimize differences resulting from constant background color differences */`
4. $A, B \leftarrow gaussian(A), gaussian(B)$; `/* A kernel width 2 Gaussian blur to make multi-pixel differences more prominent */`
5. $A_{max}, B_{max} \leftarrow$ maximum absolute value within 9x9 convolutions ; `/* Prepare for darkness correction */`
6. $A_{scaled}, B_{scaled} \leftarrow \frac{\epsilon+A}{\epsilon+A_{max}}, \frac{\epsilon+B}{\epsilon+B_{max}}$; /* Correct for render darkness, $\epsilon = 10$ */
7. $\delta \leftarrow$max root-mean-squared error over 50x50 windows from $|A_{scaled} - B_{scaled}|$; `/* Consider maximum localized difference */`

F. PolyFile / PolyTracker integration

Two parser / file analysis tools included in most FAW distributions are PolyFile and PolyTracker [11].

PolyFile is a unified file format tool that includes parsers for more than 263 MIME file types. Within the FAW, it is used for two purposes: 1) to provide polyglot detection, where an input is a valid instance of multiple file formats, and 2) to provide a file detail view that allows users inspect specific parsed structures or byte regions within the file. PolyFile includes the ability to decompress and scan compressed streams, making it invaluable as a microscope for diagnosing individual file behaviors.

PolyTracker is an instrumentation toolkit for existing parsers that adds a variety of taint-tracking-enabled functionality. Notably, this includes both control flow tracking algorithms that can be used for constrained forms of grammar inference [11], and an ability to identify "blind spots," or bytes that do not affect the parser's interpretation of bytes in a given file, even when mutated [4]. Blind spots are potentially critical to preventing parser differentials, as when parsers disagree on blind spots, one parser's behavior will be unaffected by crafting those bytes for another parser.

G. Grammar inference / pipeline plugins

Data-driven parsers, such as those learned via machine learning (ML), require additional infrastructure to manage the training and application of the model. These are handled via a pipelining plugin system, which allows for an acyclic network of "Tasks" to be defined. Each task is a processing chunk whose results take significant amount of time, motivating the need to save the output once it is computed. These tasks are allowed to potentially crash, and so convergent computing hooks are provided to allow resuming from a previous checkpoint.

Once all tasks are completed, users may define parsers to be included in analysis sets, or file detail and decision plugins using the result of the pipeline. A notable usage of the pipeline plugin system is integration with RL-GRIT [5], which provides an ability to learn grammatical snippets of data that are more efficient and produce fewer records in the database than N-grams. This has shown utility for root cause analysis, though the underlying grammar inference method needs additional work to be of significant utility.

H. Developer-friendly features for rapid parser development

The FAW provides a developer mode to facilitate rapid development of both external parsers and FAW plugins. When operating in this mode, functionality to live-reload various plugins or inputs in the corpus based on the user's changes. When a parser is changed, and the user bumps the corresponding version field in its configuration, the FAW will recompute the necessary differences in the database with as little work as possible. This turns the FAW from a purely analytic tool into a powerful integrated development environment (IDE), allowing for users to explore the consequences of code changes at scale against an entire corpus.

1) Continuously integrated reports: While the FAW provides tools and a customized user interface to examine and visualize the behavior of a parser against a set of files, it also provides support for use as a continuous integration (CI) tool. The CI interface provides access to the core features of the FAW, without the need for a human to go through the UI or needing direct access to the FAW's host machine. In particular, the CI interface supports adding, updating and reconfiguring parsers on a running FAW instance, (re-)executing the new versions against the corpus, and reporting results. The interface can be accessed via a web API or a command line tool, enabling its use in many continuous integration scenarios, for example, to track regressions during the development of a new parser or the modification of an existing one.

When a parser configuration is updated and pushed to the FAW instance via the web api, a rebuild of the updated parsers is initiated in a separate docker container without disrupting to the currently running instance. Once the rebuild is complete, the new binaries for the parsers are copied to the original FAW instance. It then triggers a reparsing of files using the updated parsers. The CI interface provides an endpoint to (re)process the parse results based on a specific decision DSL and retrieve aggregated results.

V. Related work

The FAW is a generalization and refinement of the "PDF Observatory" first introduced in [12]. At the time of that publication, many of the extensibility features were not yet

available, and far fewer analysis tools were available. This work brings a significant increase in maturity of design, implementation, and results.

The closest external artifact that we are aware of comes from Allison *et al.* [13]. Their "File Observatory" emphasizes use of ElasticSearch capabilities to catalogue and search a large corpus of PDF documents. This allows for better search results at scale on supported queries, but is limited to the queries supported by ElasticSearch, precluding results like those presented in Section IV-D. Additionally, the FAW introduces many features aimed at supporting parser developers in rapidly developing and assessing the impacts of changes to their parsers, including the on-the-fly recomputation of data requiring updates and the CI pipeline (Section IV-H).

VI. Future work

One of the key successes of the FAW has been its developer-friendly orientation. To that end, we envision continuing to add features which encourage specifically parser developers to use the FAW as part of their development workflow. For example, most of the auto-reload features currently provided rely on watching for changes within the FAW distribution's source code. In contrast, many parsers live in their own repositories. Therefore, we want to add additional pseudo-docker commands to the FAW's configs that allow for, e.g., the auto-building and integration of built artifacts from external code sources, automatically bumping parser versions when changes are detected. This would reduce the barrier to entry for new users.

Improvements to the statistical methods used for root cause analysis are imagined. The current framework excels at IMPLIES relationships between two features; however, sometimes one error might be implied by a combination of multiple features. Taking this idea further, we wish to explore partitioning of errors – that is, when a logical OR relationship comes together to cause some overarching effect. The idea of a partition might further be inverted to automatically find groups of features that themselves imply distinct sets of errors – for example, it would be of interest to find that a specific format version were responsible for many errors.

Finally, the grammar inference capabilities integrated with the FAW need to continue to be improved. While initial results are promising, the integration of, e.g., automatic grammar extraction and anomaly detection techniques [14] would allow for additional features which help pinpoint root causes and better understand parser/input interactions.

VII. Conclusion

This work has introduced the FAW, an open-source, extensible tool for studying a wide array of parser/input interactions. It was designed to be developer friendly, while also empowering non-developer users to analyze error messages and parser differentials to better understand program behaviors. The system allows for offline, horizontally scaled processing of files. We were able to use this to achieve $0\,\%$ false positives, and only $0.5\,\%$ false negatives against a gold standard from the SafeDocs program. Creating a new format distribution only took 10 minutes, and we were able to use that distribution to trace root causes for some error messages. Additionally, the FAW demonstrated easy integration of both multi-format parsers, like PolyFile, and taint trackers requiring parser recompilation, like PolyTracker; these were used to identify polyglots and blind spots within a corpus of more than 1 million files. A data-driven parser, the RL-GRIT ML algorithm, was implemented to show the FAW's ability to correlate inferred grammar snippets to error messages. The assembled workbench has already proven useful for understanding multiple file formats, and multiple ideas were presented for continuing to improve capabilities that help analysts identify and mitigate risks caused by the interactions between parsing programs and their inputs.

Acknowledgments

This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR0011-19-C-0076. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA). The authors thank Sergey Bratus and anonymous reviewers for their feedback and advice in editing this work.

References

[1] 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)*, pp. 45–52. IEEE, 2016.

[2] T. Allison, W. Burke, V. Constantinou, E. Goh, C. Mattmann, A. Mensikova, P. Southam, R. Stonebraker, and V. Timmaraju, "Building a wide reach corpus for secure parser development," in *2020 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 318–326. IEEE, 2020.

[3] K. Ambrose, S. Huntsman, M. Robinson, and M. Yutin, "Topological differential testing," *arXiv preprint arXiv:2003.00976*, 2020.

[4] H. Brodin, E. Sultanik, and M. Surovič, "Blind spots: Automatically detecting ignored program inputs," 2023.

[5] W. Woods, "RL-GRIT: Reinforcement learning for grammar inference," in *2021 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 171–183. IEEE, 2021.

[6] M. Robinson, "Looking for non-compliant documents using error messages from multiple parsers," in *2021 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 184–193. IEEE, 2021.

[7] M. Robinson, C. Anderson, L. W. Li, and S. Huntsman, "Statistical detection of format dialects using the weighted dowker complex," in *2022 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 98–112. IEEE, 2022.

[8] M. Robinson, "Cosheaf representations of relations and dowker complexes," *Journal of Applied and Computational Topology*, vol. 6, no. 1, pp. 27–63, 2022.

[9] K. J. Rothman, *Epidemiology: an introduction*. Oxford university press, 2012.

[10] G. D. Evangelidis and E. Z. Psarakis, "Parametric image alignment using enhanced correlation coefficient maximization," *IEEE transactions on pattern analysis and machine intelligence*, vol. 30, no. 10, pp. 1858–1865, 2008.

[11] C. Harmon, B. Larsen, and E. Sultanik, "Toward automated grammar extraction via semantic labeling of parser implementations," in *Proceedings of the Sixth Workshop on Language-Theoretic Security (LangSec)*. IEEE Symposium on Security and Privacy, 2021.

[12] S. Cowger, Y. Lee, N. Schimanski, M. Tullsen, W. Woods, R. Jones, E. Davis, W. Harris, T. Brunson, C. Harmon *et al.*, "Icarus: Understanding de facto formats by way of feathers and wax," in *2020 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 327–334. IEEE, 2020.

[13] T. Allison, W. Burke, D. Graf, C. Mattmann, A. Mensikova, M. Milano, P. Southam, and R. Stonebraker, "Progress on building a file observatory for secure parser development," in *2022 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 168–175. IEEE, 2022.

[14] A. Grushin and W. Woods, "Anomaly detection with neural parsers that never reject," in *2022 IEEE Security and Privacy Workshops (SPW), LangSec*, pp. 88–97. IEEE, 2022.