PolyDoc: Surveying PDF Files from the PolySwarm network

Prashant Anantharaman, Robert Lathrop, Rebecca "bx" Shapiro, Michael E. Locasto
Narf Industries

***Abstract*—Complex data formats implicitly demand complex logic to parse and apprehend them. The Portable Document Format (PDF) is among the most demanding formats because it is used as both a data exchange and presentation format, and it has a particularly stringent tradition of supporting interoperability and consistent presentation. These requirements create complexity that presents an opportunity for adversaries to encode a variety of exploits and attacks.**

To investigate whether there is an association between structural malforms and malice (using PDF files as the example challenge format), we built PolyDoc, a tool that conducts format-aware tracing of files pulled from the PolySwarm network. The PolySwarm network crowdsources threat intelligence by running files through several industry-scale threat-detection engines. The PolySwarm network provides a *PolyScore*, which indicates whether a file is safe or malicious, as judged by threat-detection engines. We ran PolyDoc in a *live hunt* mode to gather PDF files submitted to PolySwarm and then trace the execution of these PDF files through popular PDF tools such as Mutool, Poppler, and Caradoc.

We collected and analyzed 58,906 files from PolySwarm. Further, we used the PDF Error Ontology to assign error categories based on tracer output and compared them to the PolyScore. Our work demonstrates three core insights. First, PDF files classified as malicious contain syntactic malformations. Second, "uncategorized" error ontology classes were common across our different PDF tools—demonstrating that the PDF Error Ontology may be underspecified for files that real-world threat engines receive. Finally, attackers leverage specific syntactic malformations in attacks: malformations that current PDF tools can detect.

1. Introduction

The Portable Document Format (PDF) is complex yet ubiquitous [22]. However, given the size of the specification and how long it has been in use, it is a notoriously hard format to parse correctly. As a result, parsers often deviate from the specification and produce errors on entirely different sets of files [3].

PDF files also represent an exploit delivery mechanism [30], [38], [41]. A PDF tool that processes a file may crash due to a bug in the parsing logic triggered by a slight deviation from the specification—in which case the error message the parser outputs would be helpful to identify what malformation may exist [1]. An attacker could also carefully craft an exploit that leverages some vulnerable PDF libraries [9].

Attackers may engineer exploits to target a software application's vulnerable code and conditions. However, at the same time, the overall PDF file may be generally well-formed, meaning that the input "carrier" of the said exploit, such as a protocol message or data file consumed by the target software application, is syntactically valid. What is unclear is whether *crafted* exploits, so carefully wrought in one dimension, are also carefully constructed in another and to what extent the rules of a complex format may either restrict or permit such compliance with the general rules of the format.

In other words, are exploits necessarily contained within "bad" files? Is there something about most vulnerabilities and exploits that requires the conveying content to be ill-formed? Intuition could swing either way on this score: malicious files could exist that are very well-formed and indistinguishable from normal or benign files [31], [33], also known as *werewolf files* [9], [23] (modulo the exploit code itself, as Song et al. [44] and Mason et al. [34] have shown). On the other hand, exploits could be something that cannot be cleanly wrapped or represented as a mostly-normal instance of the given protocol or file format—and good, compliant parsers may be able to identify them.[1]

This paper considers the hypothesis that malicious files, i.e., files containing malware, do not possess many (or any) structural or syntactic malformations. However, it is not evident that the opposite inference is valid. It may be true that malicious files may either be very well-formed or so malformed as to be barely recognizable as an instance of the format. In the extreme, the only information indicating that the malicious sequence of bytes has anything to do with a particular format may be extraneous or irrelevant metadata like a file name extension. In any case, the attacker intends to consume the file by a particular target pile of code and logic so that the malicious bytes trigger the intended effect. Hence, well-crafted, largely valid files may include or contain an exploit that hardly perturbs the overall syntactic structure of the host format.

To investigate this hypothesis, we built *PolyDoc*—a system that interacts with the PolySwarm network to gather files. PolySwarm employs various threat-detection engines

1. A counterexample is the 2005 windows WMF exploit [13], which used an existing format feature to handle anticipated errors and malformations in the graphics printer setup language to transfer control to an error-handling function. Here, even a slight isolated malformation was enough to trigger the unsafe control transfer to attacker-supplied code.

2023 IEEE Security and Privacy Workshops (SPW) | 979-8-3503-1236-2/23/$31.00 ©2023 IEEE | DOI: 10.1109/SPW59333.2023.00017

DOI 10.1109/SPW59333.2023.00017

to investigate malware in various files. PolyDoc also deploys a set of analyzers and tracers to investigate how various PDF tools parse files and why they may fail. We then compare the malformations detected by PDF tools and whether PolySwarm found malware.

We found that today, malicious PDF files contain many malformations — many more than benign files, which contain a certain level of malformations considered benign — a kind of background radiation. We also ran *bpftrace* with our analyzers to demonstrate how benign and malicious files are parsed differently and may contain different syntax. In addition, different types of malware, such as viruses and cryptominers, are also contained in malformed PDF files.

A plausible explanation for these findings is that most code injection attacks or exploits require violation of syntactic file properties (even though, and especially in a format like PDF, any single or even a tiny tuple of syntactic malforms are ignorable or recoverable). However, it could also be the case that attacker toolkits have quite a lot in common and do the bare minimum regarding syntactic validity to place the exploit at the required place without attackers caring about full theoretic syntactic validity. They may not try to fully mask or transform the file to make it look as benign as possible to structure or format-aware tracers and scanners.

The truth is likely a blend based on the nature of the kinds of threats one considers. For example, for specific categories of attack or exploitation, exploiting an underlying code vulnerability may require significant malformation in a set of bytes that purports to be a specific file of a particular format. On the other hand, it could also be that only a minimal amount of malformation is strictly required by the exploit, but the attackers do not bother to fix up the overall appearance. Furthermore, certain attacks may have little to do with syntactic malformation. Instead, they are malicious because they contain malicious subformat (like Javascript [25]) or are a known phishing payload that can appear syntactically valid (and legitimate) except for where the input collection is targeted or transferred.

In this work, we use a real-world data source of malicious PDF files to assess the seemingly simple question: are malicious files necessarily malformed? We investigate this question by not just syntax-checking the input by employing a set of PDF parsers but also the format-aware tracing of parser logic. To that end, the contributions of this paper are as follows.

- We present the first-of-its-kind study exploring a corpus of known malicious files—comparing the output of parsers with threat-detection engines (Section 4). Over 60% of the files we collected from PolySwarm have PolyScores greater than 0.8.
- We demonstrate that many malicious files contain malformations that existing state-of-the-art PDF tools can detect (Section 5). Files that are assigned malware labels, such as Trojans and Crpytominers, do contain syntactic malformations that are detected by various PDF tools.
- Through our experiments, we identify the need to improve the PDF Error Ontology (Section 6). The ontology does not recognize many of the errors PolyDoc encountered in the *live hunt* study.

2. Design Space Overview

Parsers are the first line of defense protecting the program's address space. Insufficient parsers have been the cause of several exploits. Characterizing parser behavior is, hence, vital to securing programs [35].

Tracers exist for general computer programs, and they trace actions defined with respect to the ISA—the underlying "language" or "virtual" machine model of computation [17]. This information is extremely detailed and valuable, but it is too low level, often quite invasive and performance impactful, and has some other challenges (e.g., comparing information across runs or platforms). Nevertheless, it is therefore "easy" to trace sequences of events like instructions, basic blocks, system calls, or functions [8]. This is because developers usually want to debug the operation of the program logic on the actual hardware or within the runtime environment concerning those rules.

But parsers contain their own virtual machine: the interpreter for the language format whose symbols they may be consuming. We thus want to build a tracer for that interpreter. We think of this as implementing a "gdb(1) for format F", where only the parser logic within the larger program is instrumented and can be examined, single-stepped, etc. Instrumenting "just" the parser logic is not straightforward because few parsers are cleanly separated from the corresponding processing logic (e.g., to render a PDF or support editing of a DOCX) [7]. Some parsing logic represents a mixture of the primary container format and one or more subformats. Parsers often include a writer or transducer, often working alongside the "reading/data ingest" logic of the parser to manipulate a shared data object representing the AST or DOM of the data.

A motivating goal (but by no means the only goal) is to produce a "byte accounting" in the form of a "story of a byte's journey" from a position in an input file to the "final resting places" in the process memory right before it is read or written by arbitrary processing code [20], [49]. This working goal explicitly acknowledges that the story of that byte may seem to begin at only a single position, but this is not really true: because the byte can exist with respect to multiple abstract "address spaces," i.e., relative to many constructs at different layers of abstraction, we need to begin our efforts by building in support for tracking the byte relative to each of these starting and intermediate address spaces. As a simple example, a byte of content in a PDF file may start by existing in at least three namespaces (which are not at all aliases): (1) the OS level meaning of the zero-offset byte in the OS file, (2) the offset relative to the start of the PDF File, as File is meant in the PDF-2.0 specification and (3) an offset relative to a specific instance of a PDF format feature (i.e., some object in the PDF DOM or structured context itself).

Several tools available today may aid in flexibly observing parser behavior. On one end, typical Unix tools consider the content as only a flat byte stream, e.g., grep and xxd. Conversely, we "detonate" the file in a heavily dynamically instrumented VM to conduct malware analysis [4], [18], [51]. In this case, the focus is on the malware, not the envelope or delivery mechanism.

Control flow tracing tools operate at several different layers of abstraction; some only library level, others only syscall level, and some others at the machine level provide reads and writes and get very fine-grained authoritative information [8], [17], [20], [29]. All these tracers produce too much information at too high a performance cost, so we need to filter, compress, and discard information and then boil it down to coarse summaries. Several tools also require access to source code. PolyTracker [49] addresses this problem to a large extent. It produces a mapping of input and their corresponding memory locations, allowing us to track parser behavior more carefully.

These varied approaches have different memory and time costs and often require extensive human input and intervention for meaningful results. Hence, as a tradeoff, we use *bpftrace* to trace parser behavior in this work [47]. This presents the optimal point in the design space where we can monitor control flow, system calls, memory, and error messages without additional instrumentation and intervention.

There are several reasons to consider bpftrace as the optimal tracer for our use cases.

1) It traces the program at multiple layers of abstraction: kernel, syscall, library, and application.
2) It provides "optimal" performance for a software system: executes in kernel space, not incurring any emulation or simulation costs, or unneeded traps to and from more privileged layers (e.g., as is done in traditional VM-based introspection techniques)
3) It takes advantage of an existing flexible read/write scriptable supervision framework.

A parser tracing framework must aim to understand how a parser does what it does; this can be represented at varying degrees of granularity, from simple accept/reject to general error messages to instruction–level traces of application execution.

3. Related Work

This work builds on work done by several other researchers. This section draws on some past work to distinguish and motivate PolyDoc.

Polyglot files. There is a large body of work dealing with detecting Polyglot files—files that may be interpreted into more than one filetype [24]. Ange Albertini developed several tools to demonstrate how to create such Polyglot files.[2]

Trail of Bits has authored PolyFile [48] and PolyTracker [19], [49]. PolyFile is a tool to find Polyglot files, whereas PolyTracker tracks which bytes of input are operated on by which functions in the code by adding an LLVM pass to instrument the programs. Our paper uses the name PolyDoc because it relies on the PolySwarm network to capture malicious files and gather malformation labels and not because it has any relation to any of these other projects.

Strengthening the PDF Specification. The PDF specification has already gone through several iterations [22]. In 2021, Wyatt et al. proposed building the Arlington PDF Model [52], a machine-readable version of the PDF specification. Subsequently, PDF products have adopted the Arlington model to ensure better compliance [36]. In 2022, Tullsen et al. [50] studied the trust chain of PDFs and how the process of building the PDF DOM can be better formalized.

In this paper, PolyDoc does not implement portions of the PDF specification. Instead, it relies on existing PDF tools, such as Mutool, Caradoc, and Poppler, to enforce the PDF specification and report any errors.

PDF Malware Detection and Mitigation. Containment and blocking malicious files and bad input have been common themes in security research for the past few decades. Vigilante detects fast-spreading worms by receiving self-certifying alerts and adding corresponding filters [12]. Bouncer adds filters based on static and dynamic analysis to generalize filters that are harder to bypass [11]. On these lines, there has also been a body of work focused on detecting malicious PDF files using machine-learning approaches.

Blonce et al. [5] present a security analysis of the PDF format in their Blackhat Europe talk. They discuss how phishing attacks and privilege escalation attacks can be triggered by using valid PDF files. Raynal et al. [39] present a survey of how exploits targeting Adobe Reader are constructed. Ange Albertini also shows how we can construct polyglot files to confuse various file readers [2].

Laskov et al. [25] extract JavaScript code in PDF files and statically analyze them. Smutz et al. [43] use a Random Forests classifier on features extracted based on document metadata and overall structure to identify malicious PDFs. Maiorca et al. [32] also use a feature extractor that leverages the overall syntactic structure of PDF files and uses it to build an accurate classifier that detects malicious files. Barnum applies deep learning to hardware execution traces to detect control-flow anomalies in PDF and Word document execution [54].

Balzarotti et al. [4] run files in heavily instrumented and uninstrumented tools to compare their performance. They found that malware engineers these days design tools to evade detection. Running such a comparison of the memory performance would reveal "split personalities."

Parser Tracing. In the last two decades, we have seen a massive improvement in binary instrumentation and dynamic analysis tools. In 1999, Hunt et al. [21] built Detours,

2. https://github.com/corkami/mitra

a tool to debug and profile arbitrary Win32 functions on x86 machines. They presented a novel design using *trampolines* to preserve the uninstrumented binaries as subroutines. Dtrace [8] allowed pausing processes, and analyzing system calls and kernel data structures in runtime in Solaris. Pin [29] and DynamoRIO [16] instrument binaries and observe an application's behavior by counting basic blocks and monitoring registers, instructions, IO operations, and memory accesses.

Lin et al. [27] built REWARDS, a system to reverse engineer data structures using dynamic analysis. They tag each memory location with a timestamped type attribute and propagate types across memory locations and timestamps to reverse engineer the entire structure. MACE applies various state-reduction algorithms to concolic execution to improve state exploration [10].

Henderson et al. [20] present an approach to virtual machine introspection based on instruction-level tainting. Their binary analysis system is built on top of QEMU. Avatar^2 [37] is a multi-target orchestration framework that supports different binary analysis frameworks, operating systems, debuggers, and physical devices—supporting internal state transfers.

PLATPAL [53] is the closest in terms of design and goals to PolyDoc. To build PLATPAL, Xu et al. use OS-level semantics such as heap management, system libraries, syscalls, and memory layouts to differentiate between execution on different operating systems. Suspicious files are submitted to multiple virtual machines running different operating systems. PLATPAL then logs divergent behavior between the two traces as malicious. PolyDoc takes a similar approach by relying on various PDF tools rather than instrumenting the same tool in different operating systems. We will investigate merging the two approaches in future work.

4. System Overview

In this paper, we aim to identify if there is a co-occurrences between malware identified by threat-detection engines and common PDF tools. To that end, we built the PolyDoc, a system comprising several components.

4.1. PolyDoc Architecture

Figure 1 shows the architecture of the PolyDoc system. The PolyDoc system comprises the components we built: Orchestrator, Analyzers, and Storage Services. In addition, it also uses the following off-the-shelf software: PolySwarm network, Celery Task Manager, PostgreSQL database, and Grafana Dashboard.

Orchestrator. The orchestrator provides a REPL (Read, Evaluate, Print, and Loop) prompt. This REPL allows users to submit files to the PolySwarm network and start analyzer tasks on these files. The orchestrator also allows us to check the results of analyzer tasks by supporting queries.

Storage. Our storage system downloads and caches PDF files from the PolySwarm network. We store local copies of PDF files to analyze them further. We store file metadata, such as SHA256 hash, last read time, and file size, in our PostgreSQL database for future lookups.

Analyzers. Our analyzers run various PDF tools and bpftrace on PDF files. The analyzers store outputs of the parsers and bpftrace in our database. We must ensure that these analyzers run with strong isolation tools since they run PDF tools on arbitrary PDF files—especially malicious ones from PolySwarm. PolyDoc uses nsjail for strong isolation. Table 1 shows the PDF tools we instrument and the exact commands we execute.

Caradoc. Endignoux et al. [14] presented a restricted PDF specification to reduce parsing ambiguity. They implemented this restricted implementation and released it as an open-source tool. They implemented an $LR(1)$ grammar to define the structural syntax of a PDF file. They built an additional type checker to impose stricter rules on the PDF objects. Despite a normalization step to relax the checks the parser imposes, Caradoc is still too strict in its validation in comparison to other PDF tools. Caradoc only supports one stream compression filter and validates the graph constructed with indirect references.

PDF Tools. The VIPER verification framework provides verification tools for various programming languages, including Java, Python, and Rust. The PDF parser produced by Didier Stevens is a part of the VIPER framework and allows users to parse PDF and displays the data of various PDF objects present [45]. The tool can generate statistics and list the type of objects in a PDF file. The tool was primarily created to help with understanding PDF malware.

4.2. PolySwarm

PolySwarm is a malware intelligence platform that crowdsources malware classification by running files through several popular threat-detection engines and experts (a total of 49 engines listed online[3]). Although several popular engines may detect common attack patterns, PolySwarm also uses targeted and unique threat engines that may detect unknown or zero-day threats.

The PolySwarm API also returns a PolyScore value—rating the probability that a given file contains malware. They use a proprietary threat scoring system assigning weights to various engines they consider. The PolyScore values range from 0.20 to 1.[4]

The API also provides PolyUnite labels—specifying various malware classes found by the threat engines. Table 4 lists some possible labels provided by PolySwarm.

3. https://polyswarm.network/engines

4. Given that their scoring system is proprietary, we are unable to investigate why these values start from 0.2 in practice, and not 0.

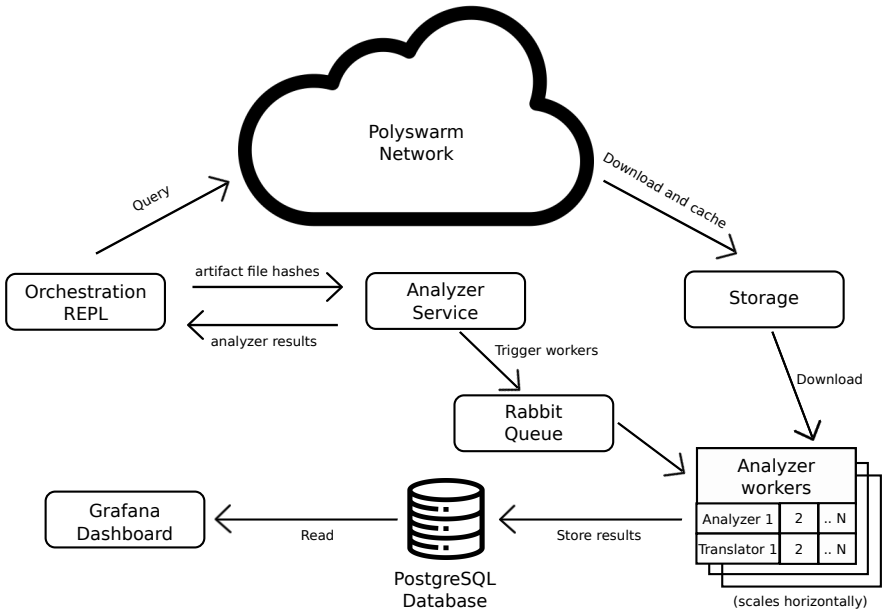


Figure 1: Polydoc Architecture showing various analyzers, translators, and the Grafana UI where the user sees the visualized results.

Table 1: PDF tools we trace as part of our study

Tool	Version	Command
Caradoc[a]	0.3	`caradoc extract --verbose --decode-streams --relax-streams {file}`
Mutool[b]	1.18.0	`mutool clean -s -d -i -f {file}`
Poppler[c]	0.84.0	`pdfinfo -box -meta -js -struct -struct-text -isodates -dests {file}`
Pdftools[d]	0.7.4	`pdf-parser.py -v -O {file}`

a. https://github.com/caradoc-org/caradoc/
b. https://github.com/ArtifexSoftware/mupdf
c. https://gitlab.freedesktop.org/poppler/poppler
d. https://didierstevens.com/files/software/pdf-parser_V0_7_4.zip

4.3. Implementation

Most of PolyDoc is implemented in Python. The Orchestrator REPL provides a command-line interface to interact with the other components of PolyDoc and to submit files and query the PolySwarm network. All the PDF tools we instrumented in PolyDoc run in their Docker containers. Using Docker containers for these analyzers allows us to upgrade parser versions easily without dealing with dependency issues.

Additionally, we manage analyzer tasks using Celery and RabbitMQ.[5][6] Celery allows us to queue and track tasks to completion, so we can extract results and store them in our PostgreSQL database once the tasks are complete.

We used Grafana dashboards to track the progress of the analyzer tasks and check how many PDF files had been processed so far. The parser tracing logs, error messages, and output are stored in the PostgreSQL database—available to be queried for more in-depth analysis.

5. https://docs.celeryq.dev/en/stable/
6. https://www.rabbitmq.com/

4.3.1. nsjail. Downloading and running analyzers on malicious files has risks [26]. Hence, we used nsjail with bpftrace to analyze files [40], [46]. The command we used is as follows:

```
bpftrace -f json -c nsjail --config
nsjail.cfg -- {command}
```

The above command formats the output as JSON and tracks the raw system calls and other kernel events. Nsjail allows us to constrain a shell by applying Linux namespaces, resource limits, and system call filters. We created nsjail and bpftrace configuration files to use with our scripts to standardize executions.

4.4. Heatmaps

Heatmaps are an output rendering of the Error Ontology [42] and the PDF tools we consider in this study. Figure 2 shows renderings of heatmaps. We produce two normalizations of these heatmaps.

First, we normalize by the error class. The error classes are listed along the X-axis. The frequency of files across different PolyScores for a single error class must add up to one in this normalization mode. This mode allows us to see if, for a specific error class, the majority of the files skew toward higher PolyScores.

Next, we normalize data by the PolyScores. Within a window of PolyScores, the frequencies must add up to 1. This mode allows us to see which error classes tend to have a higher percentage of files than others.

Heatmaps are an effective way to visualize data emitted by parser tracers and the PolySwarm network. In Section 5, we show heatmaps we generated by running a baseline (GovDocs) and a live hunt (gathering PDFs recently submitted to the PolySwarm network).

4.5. Error Ontology

The PDF Error Ontology [42] provides a set of regular expressions for various tools, such as Mutool, Poppler, Caradoc, PDF Parser, and PDFMiner. These regular expressions represent known and commonly seen error conditions in these PDF tools under investigation.

We chose to use the error ontology since it provides a way for us to categorize these errors in a more structured way. These error expressions were selected after studies on GovDocs data to see which error messages occurred frequently. We also used the error intake Python script provided as part of the error ontology that classifies errors into the above categories by taking a JSON error log as input.

5. Findings

This section reports our findings. We ran two primary experiments using the PolyDoc system: (1) on GovDocs data and (2) on PolySwarm live data.

We use our experiments to answer the following questions.

- What PolyScores do we find on GovDocs files that are known to be clean and well-formed?
- What malformation categories from the PDF Error Ontology do we see in malicious files?

5.1. Metrics and Measurements

We define some terms here that we use extensively in this section.

- **Benign Files:** Files that received a PolyScore closer to 0.2.
- **Malicious Files:** Files that received a PolyScore closer to 1.0.
- **Well-formed Files:** Files that produced no errors in any of our four PDF tools.
- **Malformed Files:** Files that produce errors in one or more of our PDF tools.

The benign or malicious status is provided by the PolySwarm network for each file with a PolyScore value. Our analyzers run our PDF tools on each PDF file in consideration. Files that do not fail with any errors are added a "no errors" tag to keep a record of any file that may be well-formed.

5.2. Baseline Experiments

Files submitted to the PolySwarm network are skewed towards malicious. Like VirusTotal,[7] people suspicious of some files they received would submit it to the PolySwarm network. We ran a baseline control experiment using the GovDocs dataset [15].

GovDocs is a dataset freely available containing 231106 PDF files. We selected 2068 files at random that were designated safe by the PDF Association and other participants on the Safedocs program [6]. Figure 2 shows the results of this baseline experiment.

In Figure 2a, we see that even though files produce errors when run through various tools, they all received the lowest possible PolyScore values—in the range of 0.20 to 0.24. However, normalizing using PolyScore values provides more relevant information, as shown in Figure 2b. We see that more than 80% of the files in our GovDocs sample run without producing any errors in the PDF tools we consider.

Table 2 details these findings. Any non-zero number of files creates an entry on the heatmap, despite the frequency often remaining in single digits in such large corpora. Even among the *safe* subset of GovDocs files, some files fail with errors in Mutool and Poppler pdfinfo.

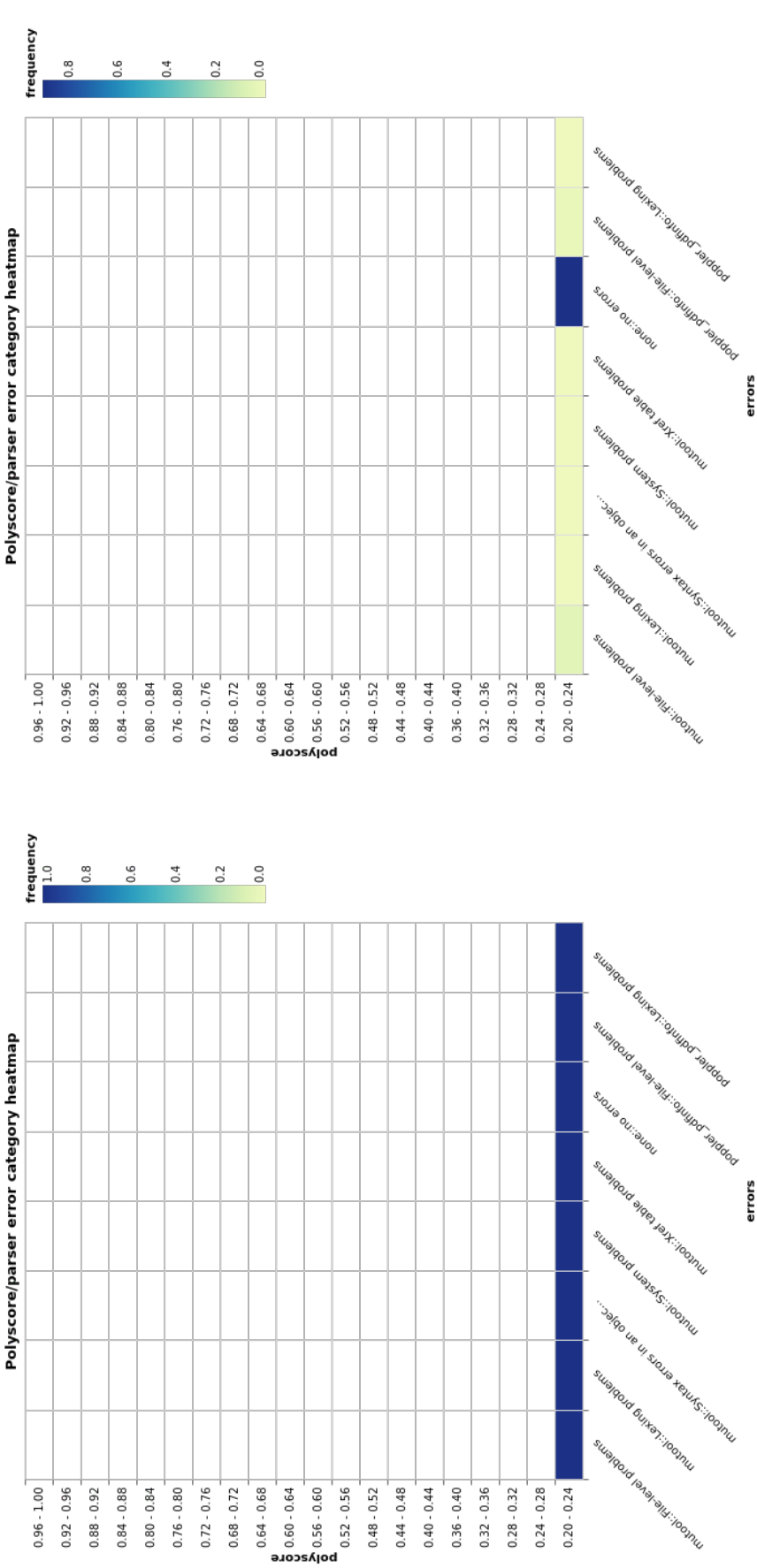
Table 2: Error frequencies when running GovDocs files on PolyDoc

Error Message	Number of PDF files
No errors	1880
mutool::File-level problems	120
poppler::File-level problems	58
mutool::Xref table	4
mutool::Syntax errors	3
mutool::Lexing problems	1
mutool::System problems	1
poppler::Lexing problems	1
Total Files	2068

5.3. Live Hunt View

We also ran the set of PDF tools on live data collected from the PolySwarm network. Users can submit files to PolySwarm by uploading them to the web interface.

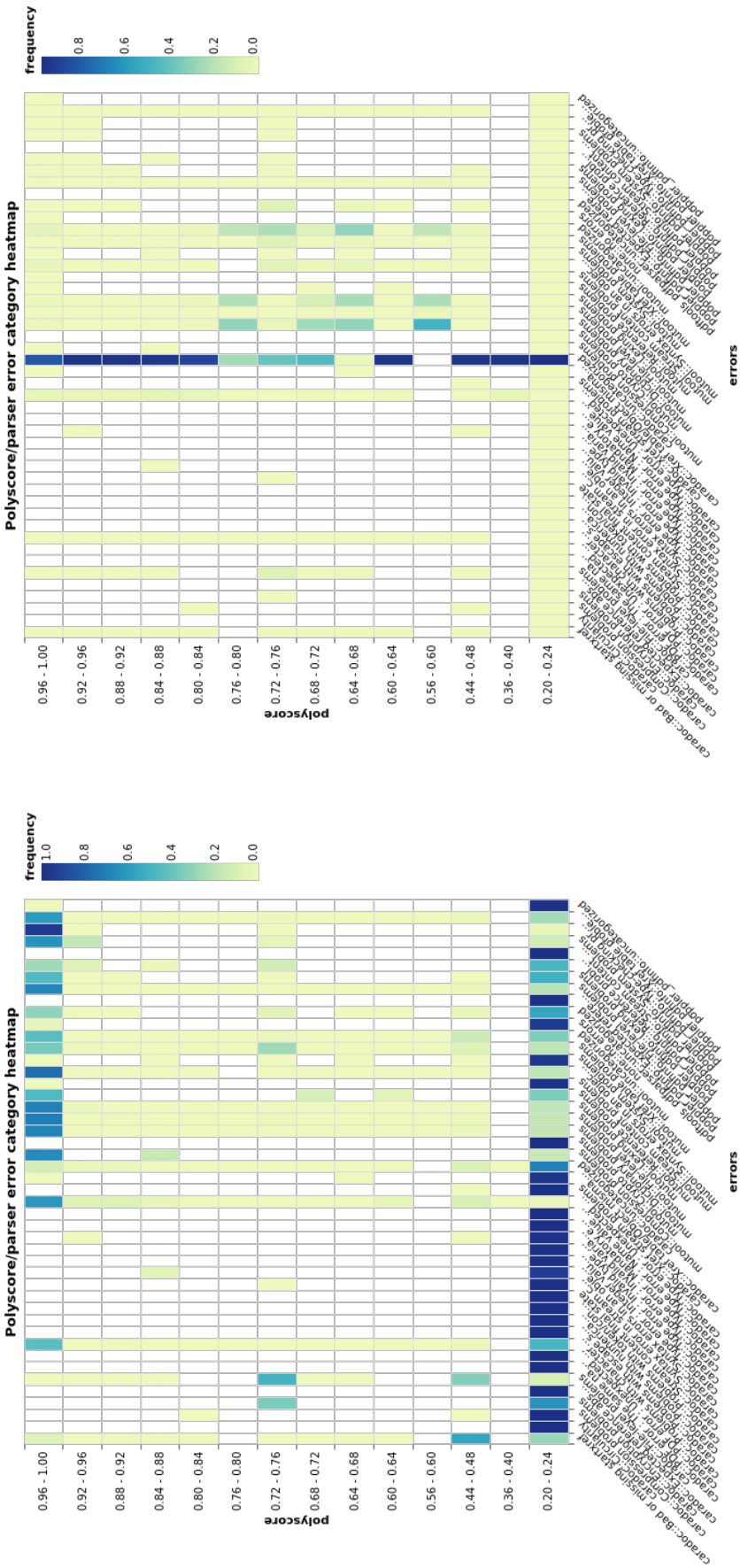
7. https://virustotal.com



(a) Normalized by error class

(b) Normalized by Polyscore

Figure 2: Heatmaps showing error classes and Polyscores for 2068 files from the GovDocs dataset.



(a) Normalized by Polyscore

(b) Normalized by Polyscore

Figure 3: Heatmaps showing error classes and Polyscores for all 58,906 files in our live hunt study.

Table 3: Results of running bpftrace with various PDF tools with files from PolySwarm. This table also shows how many files were under each PolyScore range.

PolyScore	Artifacts	Average Buffer Count	Average Gap Count	Average Gap Size	Average Event Count	Average Read Count	Average Requested Read Length
0.2–0.3	10216	2.08	1.97	936195.41	58.79	53.38	304.62
0.3–0.4	14	1.5	1.5	82842.64	24.35	19.85	201.14
0.4–0.5	7142	1.73	2.08	190841.52	31.72	26.32	240.02
0.5–0.6	7	1.71	14.57	12211783.29	1145.71	1124.14	256.0
0.6–0.7	1905	1.74	1.73	171556.63	33.74	28.10	240.95
0.7–0.8	728	1.69	2.98	382571.20	59.54	54.29	244.86
0.8–0.9	5972	1.72	1.70	236602.52	30.60	24.85	247.37
0.9–1.0	32918	1.72	2.10	284527.70	67.19	61.48	247.65

PolySwarm then submits the file to individual threat engines and waits for them to return with results and confidence scores.

By using the PolySwarm API, we can then download PDF files that were submitted recently and locally run these files through our set of tracers. By creating a "live hunt" mode for PolyDoc, over two days, we collected 58,906 unique PDF files.

Table 3 summarizes the results of our live hunt experiments with bpftrace. We collected files across various PolyScore ranges, but most of our files were in the malicious range "0.9–1.0" and the benign range of "0.2–0.3." We also see that values, such as the number of gaps between offsets and the sizes of these gaps vary drastically between different PolyScore ranges.

Figure 3 shows a heatmap of our Live Hunt findings. In Figure 3a, we see that except for the unrecognized error category and the errors with Character Encodings, all other Caradoc errors lean towards files with low PolyScores—benign files. However, in Mutool and Poppler, except for compression and internal system errors, most other errors found co-occurred with malicious files. In Mutool and Poppler, uncategorized errors tend to be in benign files, whereas uncategorized errors produced by Caradoc skew more toward malicious files.

5.3.1. Well-formed Files. In our entire PolySwarm dataset of 58,906 files, we found only 22 files that produced no errors in any of the PDF tools. Among these, 21 contained low PolyScores and had no PolyUnite labels assigned to them. One file had the "Trojan" and "Mass Mailer" labels assigned to it. Hence, PolyDoc found that only one well-formed PDF file in our dataset contained malware.

5.3.2. Xref table errors. Cross-reference tables or Xref tables contain lookup indices for all the objects in the PDF file. When a PDF file is updated, the previous Xref table is often left in place, and a new Xref table is added to append to the previous Xref table. As a result of several updates, Xref tables often contain numerous errors. Most modern PDF tools ignore the indices in the Xref tables and reconstruct internal Xref tables by parsing the entire PDF file. Hence, errors in Xref tables may be very frequent.

5.4. Malware Categories Found

PolySwarm provides a PolyScore and a set of PolyUnite labels for every file request submitted. Table 4 shows all the malware categories found by PolySwarm using the PolyUnite labels.[8] Each of these categories of files produces different trends in terms of malformations and malice. We studied the top five malware categories in more depth.

- **Trojan:** Figure 5 compares the errors and PolyScore values in detail. Figure 5a shows that some of the Caradoc errors, such as Compression problems, Encryption problems, and Xref table problems, are associated more frequently with lower PolyScore values. On the other hand, most other Poppler and Mutool errors tend to co-occur more with malice.
- **Mass Mailer:** Figure 6 shows the errors encountered for the Mass Mailer malware category. We see that most error conditions have a significant spread across the PolyScores. However, the Poppler lexing problems and Mutool compression problems are exhibited in files with high PolyScores. The "no errors" tag in Figure 6a indicates files that produced no errors in any categories we consider. This category contained only one file—and had the PolyUnite labels of Trojan and Mass Mailer.
- **Security Assessment Tool:** Figure 7 shows the results of our experiment on files with the Security Assessment Tool tag. These files are not necessarily malware but may be attempts by exploit engineers to perform reconnaissance. Most of the files with this tag represented a PolyScore greater than 0.6. Since the PolyScores are a probability measure, a number of the threat engines under PolySwarm may not have identified these files as threats.
- **Virus:** Figure 8 shows the results of our experiments for the PolyUnite label Virus. Most of the files under this label are clustered closer to the 1.0 end of the PolyScore. Other than the unrecognized labels of Caradoc and Mutool, files producing most of the other errors end up with higher PolyScore values.
- **Cryptominer:** Figure 9 shows the results of our study with the PolyUnite label Cryptominer. These figures present an exciting overview of the files in

8. A file may be assigned multiple of these labels. Several files were also uncategorized.

the PolySwarm network with this label. None of the files resulted in "no errors," and none resulted in low PolyScores. All files fail with the same Caradoc error of "Problems with character encoding." Similarly, Mutool often fails with lexing problems or dictionary problems, and Poppler found file-level problems and Xref table problems. Other than the "File-level problems" error message being common between Mutool and Poppler, none of the other error categories reported by these PDF tools are common across tools.

Upon a closer analysis of the Cryptominer files, we found that all of these files contained the PDF header but did not contain a trailer or Xref table. Instead, these files contained an entire Windows executable. Running the `file` command on these files confirms that these files are PE32+ executables built for Windows.

5.4.1. Emerging patterns. From Figures 5, 8, and 9 representing the malware labels of trojans, viruses, and cryptominers, we see that particular error messages in PDF tools identify more with malware. With careful comparisons and a deeper study of these files, we can build scanners and threat engines that rely on PDF tools instead of known, prior-seen patterns.

Table 4: Malware Labels provided by various threat engines on PolySwarm. These are PolyUnite labels provided by PolySwarm for each artifact.

Malware Category	Count
Trojan	28807
Mass Mailer	25950
Security Assessment Tool	9046
Virus	337
Cryptominer	245
Downloader	79
Prepender	45
Exploit	43
Worm	37
Nonmalware	30
Backdoor	25
Greyware	16
Browser Modifier	15
Dropper	8
CVE	4
Keylogger	4
Password Stealer	4
Injector	3
Adware	2
Spyware	2
Clicker	1
Bot	1
Banker	1

6. Discussion

6.1. Follow-on Work: Format aware Tracing

As noted in Section 2, PolyDoc used bpftrace because it was at the optimal point of the tradeoff in terms of the performance hits, level of instrumentation, and information reported. Tools such as PolyTracker [49] effectively correlate input locations with process memory. However, there is a strong need for format-aware tracing tools.

Parser tracing tools cannot be one-size-fits-all. Each data format or network protocol has its requirements and parsing patterns. For example, in the PDF format, we scan the header for the PDF version and then the trailer to find the Xref table and the Catalog dictionary. However, network protocols rarely support seek operations, but support length fields, checksums, and numerous bitvectors.

Given that each format has its own set of features, we need metrics for what is meaningful information for a particular format. For example, "cavities" or "gaps" are a vital metric for PDF files, but they may not translate to other data formats such as JPEGs or network protocols since they are not designed to have offsets and hence cannot have cavities.

In future work, we will investigate building a database of complex features that developers need to track to understand the security posture of various data formats. We will then build tracers that may apply to many of these features, and support intricate, yet meaningful tracing of parsers for different data formats.

6.2. Improving the PDF Error Ontology

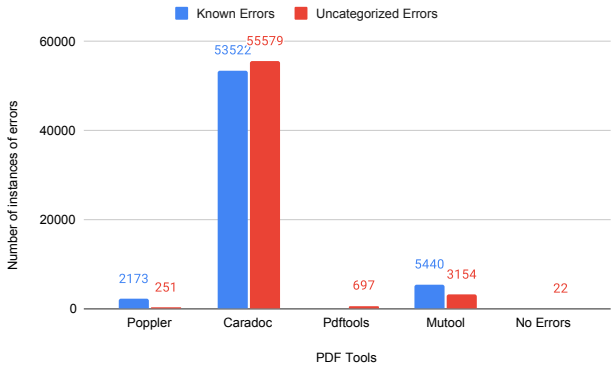


Figure 4: Comparison of uncategorized and categorized errors using the Error Ontology regular expressions

We ran the regular expressions provided with the PDF error ontology on errors emitted by various PDF tools on live hunt PDF files. We found that the uncategorized errors—errors not identified with any regular expression—were high in number. Figure 4 shows our findings.

Caradoc produces high numbers of errors in each file—the graph shows instances of errors, not the number of files. At the end of our live hunt, we saw that the number of uncategorized Caradoc errors was higher than the categorized errors. For Poppler pdfinfo, we found that close to 10% of the errors were uncategorized. Similarly, around 35% of the errors produced by Mutool were unaccounted for.

Caradoc follows the strictest set of rules among the PDF tools we use in our study. Mutool and Poppler ignore Xref table errors and reconstruct the Xref table—following a more permissive approach to parsing the PDF file. Most practical PDF tools focus on providing some output to the user—failing only when every other option fails.

This finding demonstrates the need for more research into the commonly seen error cases in real-world and malicious datasets. In future work, we will also explore if some of these unrecognized error cases are specific to malicious files—cases that are not accounted for in clean datasets. We also believe that techniques such as Pareto analysis may be helpful to minimize the crucial set of errors we need to consider to gain extensive coverage over the errors produced by various PDF tools.

6.3. Building a format-aware lightweight scanner: Identifying Malicious Files

At the beginning of this investigation, we set out with a research question to understand how we can build simple intrusion detection tools using insight from these format-aware tools. The heatmaps we produce in this paper are effective tools for visualizing co-occurrences between the errors produced while parsing PDFs and malice. In addition, we also see that bpftrace logs of malicious and benign files deviate in specific behaviors, such as gaps between offsets (also known as cavities in Polyglot files), and the number of memory events, as seen in Table 3.

However, more research is needed to construct first-pass filters for an IDS to detect malformed files early to conduct a deeper analysis of potentially malicious files. We believe that with more statistical analysis of bpftrace logs and comparing parser errors with malicious behavior, such filters may be feasible in future work.

6.4. Beating the system

Adversaries often use techniques to detect sandbox environments and instrumented operating systems to decide whether to execute a payload [4], [28]. These evasive, "environment-sensitive" approaches study the characteristics of the environment and the configurations to decide whether to execute. PolyDoc relies on PDF parsers that find structural and syntactic issues in PDF files.

We run these PDF files inside Docker containers with nsjail since our dataset contained malicious files. PolyDoc is not executing code from PDF files, so the malware is not meant to be executed. Our study aimed to identify if malware writers use malformations in PDF files to inject exploits. Implementing sandbox detection while crafting the malicious PDFs may help evade some of the engines used in the PolySwarm network. However, to evade the PolyDoc system, attackers must use well-formed PDF files to execute exploits on specific PDF viewers. Attackers must identify countermeasures that make the PDF files syntactically well-formed.

7. Conclusions

We find that in this first large-scale study of real-world malicious PDF files, there is an association between the level of malformations present in the PDF and whether a file is drawn from the malicious set as compared to a validated benign set. In particular, malicious PDF files seem to have significantly more malformations than "normal" PDF files.

Further work can explore *why* this association is present. We hypothesize that it may reflect either common attacker tooling approaches (i.e., the bare minimum structure needed to templatize a PDF file that holds a specific exploit code or malicious payload) or be due to a minimum floor of malformed complexity necessary to exploit particular vulnerabilities or logic weaknesses in extant PDF parsers. In other words, a certain level of incorrect structure is required to drive the target application code into a vulnerable state, and such code is proportionally greater than the level of malformation that extant parsers tolerate in normal benign files.

Acknowledgments

We thank the anonymous reviewers for their time and effort. Their suggestions immensely improved the quality of the paper and the arguments we made. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001119C0074. 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).

References

[1] Adobe Security Bulletin. Integer Overflow or Wraparound; Out-of-bounds Read; Stack-based Buffer Overflow in Adobe Reader. https://helpx.adobe.com/security/products/acrobat/apsb23-01.html, 2023.

[2] Ange Albertini. Abusing file formats; or, corkami, the novella. *PoC or GTFO 0x07*, 2015.

[3] Prashant Anantharaman. *Protecting Systems from Exploits Using Language-Theoretic Security*. PhD thesis, Dartmouth College, 2022.

[4] Davide Balzarotti, Marco Cova, Christoph Karlberger, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Efficient Detection of Split Personalities in Malware. In *NDSS*, 2010.

[5] Alexandre Blonce, Eric Filiol, and Laurent Frayssignes. Portable Document Format (PDF) Security Analysis and Malware Threats. In *Presentations of Europe BlackHat Conference*. Citeseer, 2008.

[6] Sergey Bratus. SafeDocs: Restoring Trust in Electronic Documents. https://www.darpa.mil/news-events/2018-08-09, 2018.

[7] Sergey Bratus, Meredith L Patterson, and Dan Hirsch. From shotgun parsers to more secure stacks. *Shmoocon, Nov*, 2013.

[8] Bryan Cantrill, Michael W Shapiro, Adam H Leventhal, et al. Dynamic Instrumentation of Production Systems. In *USENIX Annual Technical Conference, General Track*, pages 15–28, 2004.

[9] Curtis Carmony, Xunchao Hu, Heng Yin, Abhishek Vasisht Bhaskar, and Mu Zhang. Extract Me If You Can: Abusing PDF Parsers in Malware Detectors. In *NDSS*, 2016.

[10] Chia Yuan Cho, Domagoj Babić, Pongsin Poosankam, Kevin Zhijie Chen, Edward XueJun Wu, and Dawn Song. MACE: Model-inference-Assisted Concolic Exploration for Protocol and Vulnerability Discovery. In *20th USENIX Security Symposium (USENIX Security 11)*, 2011.

[11] Manuel Costa, Miguel Castro, Lidong Zhou, Lintao Zhang, and Marcus Peinado. Bouncer: Securing Software by Blocking Bad Input. *SIGOPS Oper. Syst. Rev.*, 41(6) page 117–130, oct 2007. DOI 10.1145/1323293.1294274.

[12] Manuel Costa, Jon Crowcroft, Miguel Castro, Antony Rowstron, Lidong Zhou, Lintao Zhang, and Paul Barham. Vigilante: End-to-End Containment of Internet Worms. *SIGOPS Oper. Syst. Rev.*, 39(5) page 133–147, oct 2005. DOI 10.1145/1095809.1095824.

[13] Alex Eckelberry. New exploit blows by fully patched Windows XP systems. https://techtalk.gfi.com/new-exploit-blows-by-fully-patched-windows-xp-systems/, 2005.

[14] Guillaume Endignoux, Olivier Levillain, and Jean-Yves Migeon. Caradoc: A Pragmatic Approach to PDF Parsing and Validation. In *IEEE Security and Privacy Workshops (SPW)*, pages 126–139, 2016. DOI 10.1109/SPW.2016.39.

[15] Simon Garfinkel. Govdocs1 — (nearly) 1 million freely-redistributable files. https://digitalcorpora.org/corpora/files/, 2012.

[16] Timothy Garnett. *Dynamic optimization if IA-32 applications under DynamoRIO*. PhD thesis, Massachusetts Institute of Technology, 2003.

[17] Xinyang Ge, Weidong Cui, and Trent Jaeger. Griffin: Guarding Control Flows using Intel Processor Trace. *ACM SIGPLAN Notices*, 52(4) pages 585–598, 2017. DOI 10.1145/3093336.3037716.

[18] C Guarnieri, Mark Schloesser, J Bremer, and A Tanasi. Cuckoo sandbox-open source automated malware analysis. *Black Hat USA*, 2013.

[19] Carson Harmon, Bradford Larsen, and Evan A Sultanik. Toward Automated Grammar Extraction via Semantic Labeling of Parser Implementations. In *IEEE Security and Privacy Workshops (SPW)*, pages 276–283. IEEE, 2020. DOI 10.1109/SPW50608.2020.00061.

[20] Andrew Henderson, Aravind Prakash, Lok Kwong Yan, Xunchao Hu, Xujiewen Wang, Rundong Zhou, and Heng Yin. Make It Work, Make It Right, Make It Fast: Building a Platform-Neutral Whole-System Dynamic Binary Analysis Platform. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, page 248–258, New York, NY, USA, 2014. Association for Computing Machinery. DOI 10.1145/2610384.2610407.

[21] Galen Hunt and Doug Brubacher. Detours: Binary Interception of Win32 Functions. In *3rd USENIX Windows NT Symposium*, 1999.

[22] ISO 32000-2:2020. Document management - Portable document format - Part 2: PDF 2.0 2020.12, 2020.

[23] Suman Jana and Vitaly Shmatikov. Abusing File Processing in Malware Detectors for Fun and Profit. In *2012 IEEE Symposium on Security and Privacy*, pages 80–94, 2012. DOI 10.1109/SP.2012.15.

[24] Luke Koch, Sean Oesch, Amul Chaulagain, Mary Adkisson, Samantha Erwin, and Brian Weber. Toward the Detection of Polyglot Files. In *Proceedings of the 15th Workshop on Cyber Security Experimentation and Test*, pages 120–128, 2022.

[25] Pavel Laskov and Nedim Šrndić. Static Detection of Malicious JavaScript-Bearing PDF Documents. In *Proceedings of the 27th Annual Computer Security Applications Conference*, ACSAC '11, page 373–382, New York, NY, USA, 2011. Association for Computing Machinery. DOI 10.1145/2076732.2076785.

[26] Samuel Laurén, Sampsa Rauti, and Ville Leppänen. A Survey on Application Sandboxing Techniques. In *Proceedings of the 18th International Conference on Computer Systems and Technologies*, CompSysTech'17, page 141–148, New York, NY, USA, 2017. Association for Computing Machinery. DOI 10.1145/3134302.3134312.

[27] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic Reverse Engineering of Data Structures from Binary Execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[28] Martina Lindorfer, Clemens Kolbitsch, and Paolo Milani Comparetti. Detecting environment-sensitive malware. In *Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo Park, CA, USA, September 20-21, 2011. Proceedings 14*, pages 338–357. Springer, 2011.

[29] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. *SIGPLAN Not.*, 40(6) page 190–200, jun 2005. DOI 10.1145/1064978.1065034.

[30] Davide Maiorca and Battista Biggio. Digital Investigation of PDF Files: Unveiling Traces of Embedded Malware. *IEEE Security & Privacy*, 17(1) pages 63–71, 2019. DOI 10.1109/MSEC.2018.2875879.

[31] Davide Maiorca, Igino Corona, and Giorgio Giacinto. Looking at the Bag is Not Enough to Find the Bomb: An Evasion of Structural Methods for Malicious PDF Files Detection. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, page 119–130, New York, NY, USA, 2013. Association for Computing Machinery. DOI 10.1145/2484313.2484327.

[32] Davide Maiorca, Giorgio Giacinto, and Igino Corona. A Pattern Recognition System for Malicious PDF Files Detection. In *Machine Learning and Data Mining in Pattern Recognition: 8th International Conference, MLDM 2012, Berlin, Germany, July 13-20, 2012. Proceedings 8*, pages 510–524. Springer, 2012. DOI 10.1007/978-3-642-31537-4_40.

[33] Zhengyang Mao, Zhiyang Fang, Meijin Li, and Yang Fan. EvadeRL: Evading PDF Malware Classifiers with Deep Reinforcement Learning. *Security and Communication Networks*, v.2022, 2022. DOI 10.1155/2022/7218800.

[34] Joshua Mason, Sam Small, Fabian Monrose, and Greg MacManus. English Shellcode. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, page 524–533, New York, NY, USA, 2009. Association for Computing Machinery. DOI 10.1145/1653662.1653725.

[35] Ryan McDonald and Joakim Nivre. Characterizing the errors of data-driven dependency parsers. *Google Inc.*, 2007.

[36] Mike Bremford. BFO PDF Library 2.27.2. https://bfo.com/blog/2022/12/05/bfo_pdf_library_2_27_2_introducing_the_arlington_model/, 2022.

[37] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar^2: A Multi-Target Orchestration Platform. In *Proceedings of the Workshop on Binary Analysis Research (BAR) (Colocated NDSS Symposium)*, volume 18, pages 1–11, 2018.

[38] Nexor. Preventing document based malware from devastating your business. https://www.nexor.com/resources/white-papers/preventing-document-based-malware-from-devastating-your-business/, 2017.

[39] Frédéric Raynal, Guillaume Delugré, and Damien Aumaitre. Malicious origami in PDF. *Journal in computer virology*, 6(4) pages 289–315, 2010.

[40] Alastair Robertson. bpftrace: High-level tracing language for Linux eBPF. https://github.com/iovisor/bpftrace, 2019.

[41] Karthik Selvaraj and Nino Fred Gutierrez. The Rise of PDF Malware. *Symantec Security Response*, 2010.

[42] Andrea Shepard. PDF Error Ontology trees for SAFEDOCS. https://gitlab.special-circumstanc.es/andrea/error-ontology, 2020.

[43] Charles Smutz and Angelos Stavrou. Malicious PDF Detection Using Metadata and Structural Features. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 239–248, New York, NY, USA, 2012. Association for Computing Machinery. DOI 10.1145/2420950.2420987.

[44] Yingbo Song, Michael E. Locasto, Angelos Stavrou, Angelos D. Keromytis, and Salvatore J. Stolfo. On the Infeasibility of Modeling Polymorphic Shellcode. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, page 541–551, New York, NY, USA, 2007. Association for Computing Machinery. DOI 10.1145/1315245.1315312.

[45] Didier Stevens. PDF Parser V 0.7.8. https://blog.didierstevens.com/programs/pdf-tools/, 2023.

[46] Robert Swiecki. google/nsjail: A light-weight process isolation tool, making use of Linux namespaces and seccomp-bpf syscall filters. https://github.com/google/nsjail.

[47] Ludwig Thomeczek, Andreas Attenberger, Johannes Kolb, Vaclav Matousek, and Juergen Mottok. Measuring Safety Critical Latency Sources using Linux Kernel eBPF Tracing. In *ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems*, pages 1–8, 2019.

[48] Trail of Bits. PolyFile: A pure Python implementation of libmagic. https://github.com/trailofbits/polyfile, 2020.

[49] Trail of Bits. PolyTracker: An LLVM-based instrumentation tool for universal taint tracking, dataflow analysis, and tracking. https://github.com/trailofbits/polytracker, 2020.

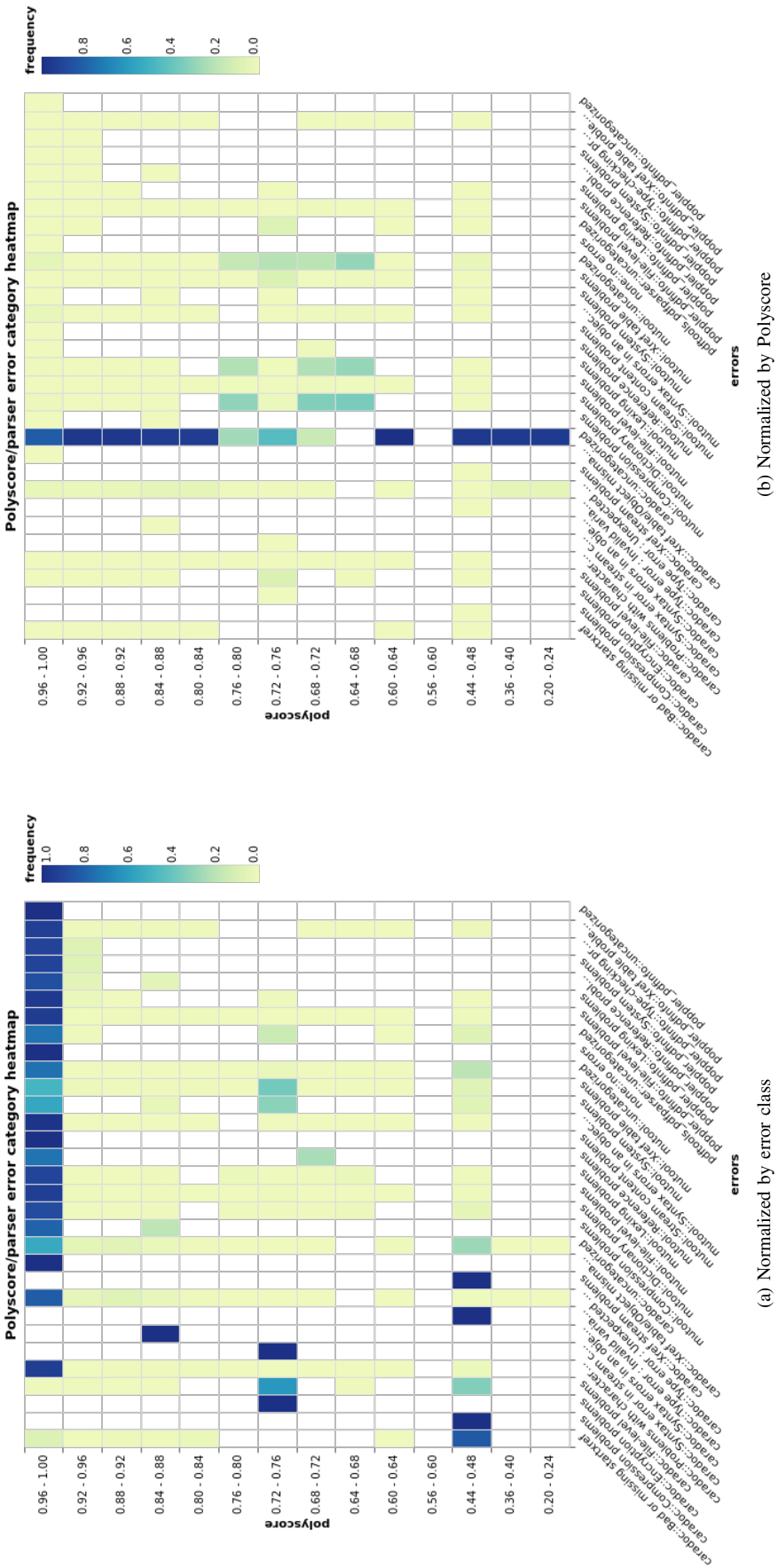
[50] Mark Tullsen, William Harris, and Peter Wyatt. Strengthening Weak Links in the PDF Trust Chain. In *IEEE Security and Privacy Workshops (SPW)*, pages 152–167. IEEE, 2022. DOI 10.1109/SPW54247.2022.9833889.

[51] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security & Privacy*, 5(2) pages 32–39, 2007. DOI 10.1109/MSP.2007.45.

[52] Peter Wyatt. Work in Progress: Demystifying PDF Through a Machine-Readable Definition. *IEEE Security and Privacy Workshops (SPW)*, 2021.

[53] Meng Xu and Taesoo Kim. PLATPAL: Detecting Malicious Documents with Platform Diversity. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 271–287, USA, 2017. USENIX Association.

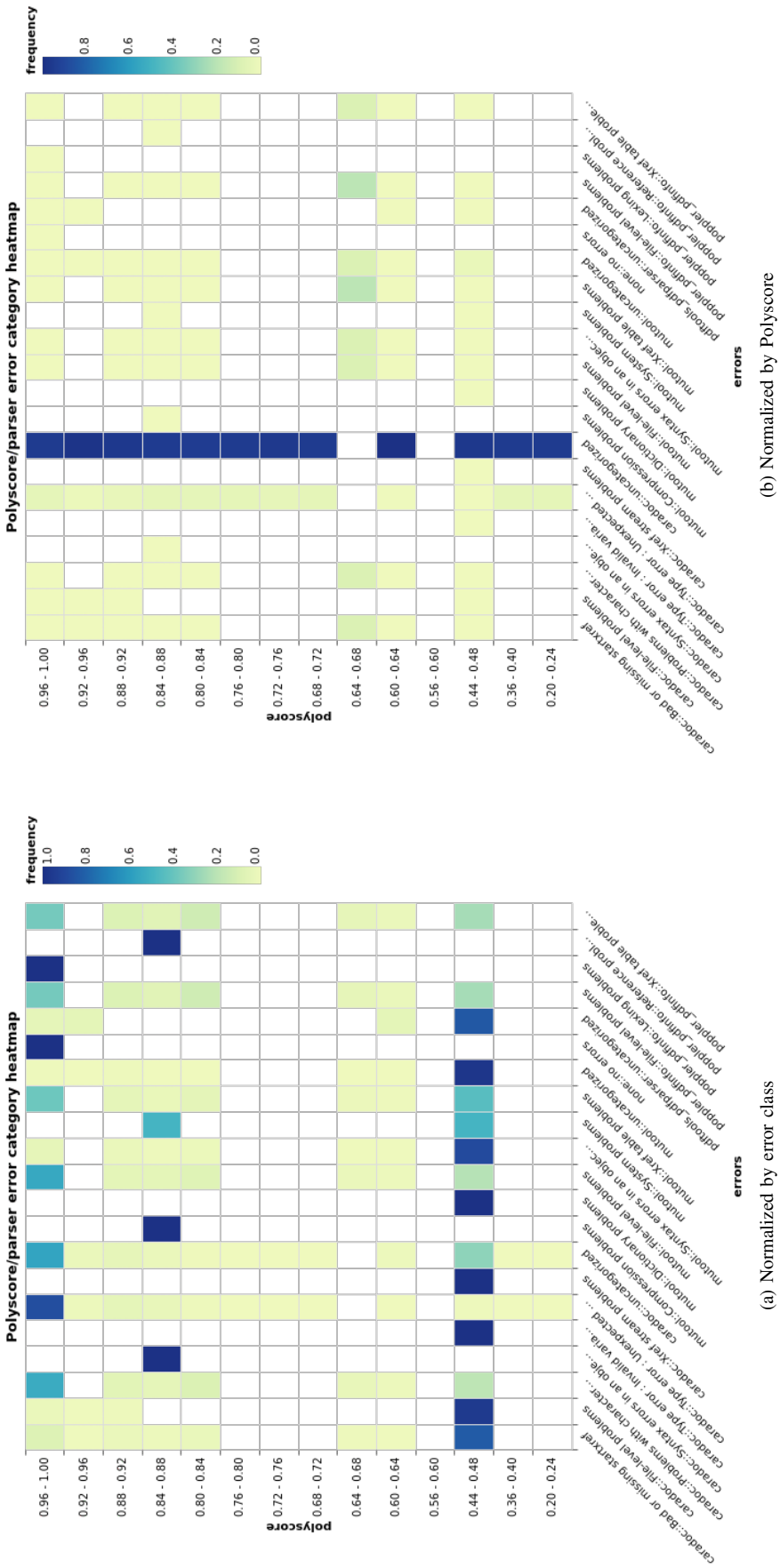
[54] Carter Yagemann, Salmin Sultana, Li Chen, and Wenke Lee. *Barnum*: Detecting Document Malware via Control Flow Anomalies in Hardware Traces. In *Information Security: 22nd International Conference, ISC 2019, New York City, NY, USA, September 16–18, 2019, Proceedings 22*, pages 341–359. Springer, 2019. DOI 10.1007/978-3-030-30215-3_17.



(a) Normalized by error class

(b) Normalized by Polyscore

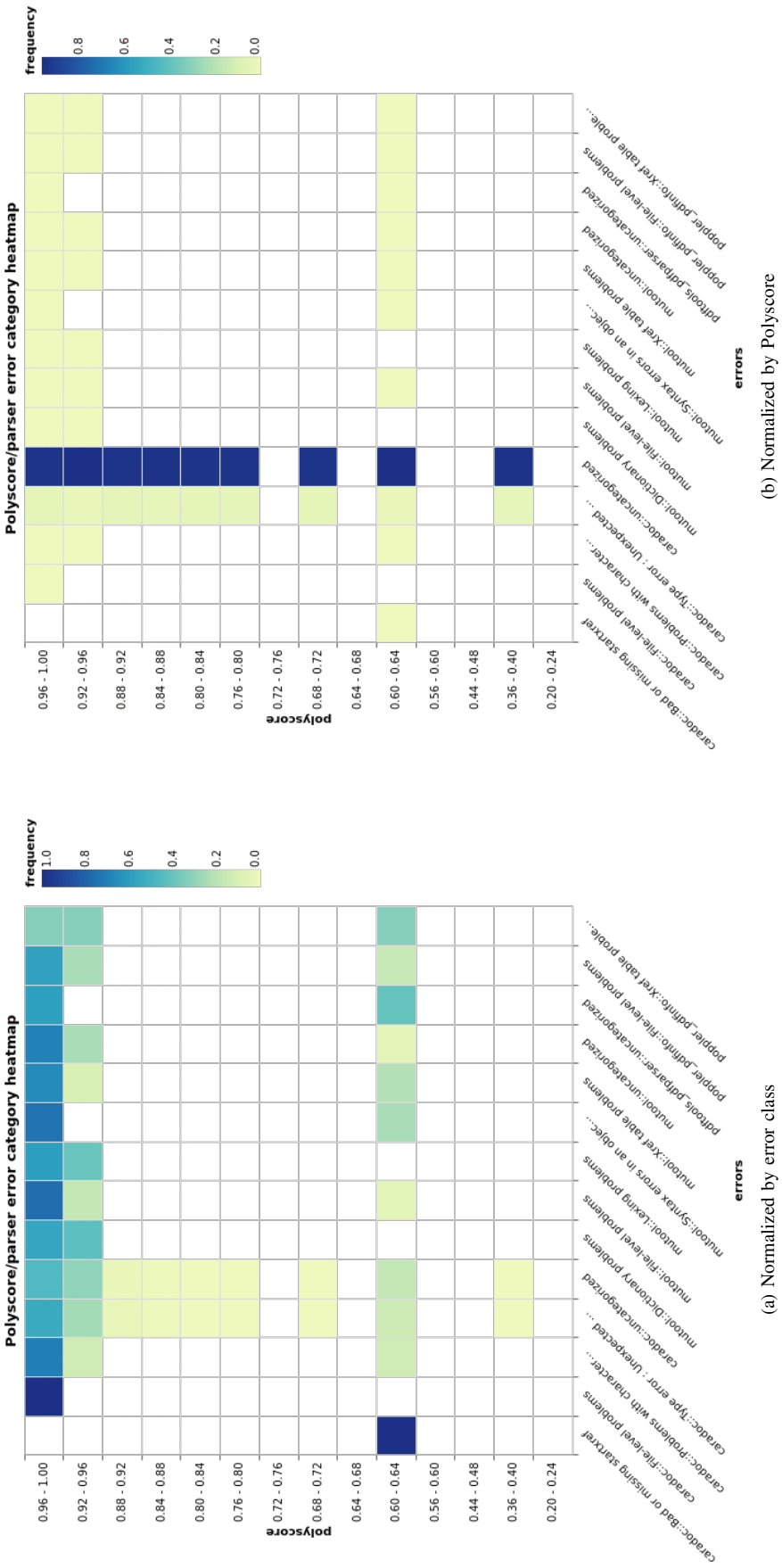
Figure 5: Heatmaps showing files labeled “Trojan” by Polyswarm.



(a) Normalized by error class

(b) Normalized by Polyscore

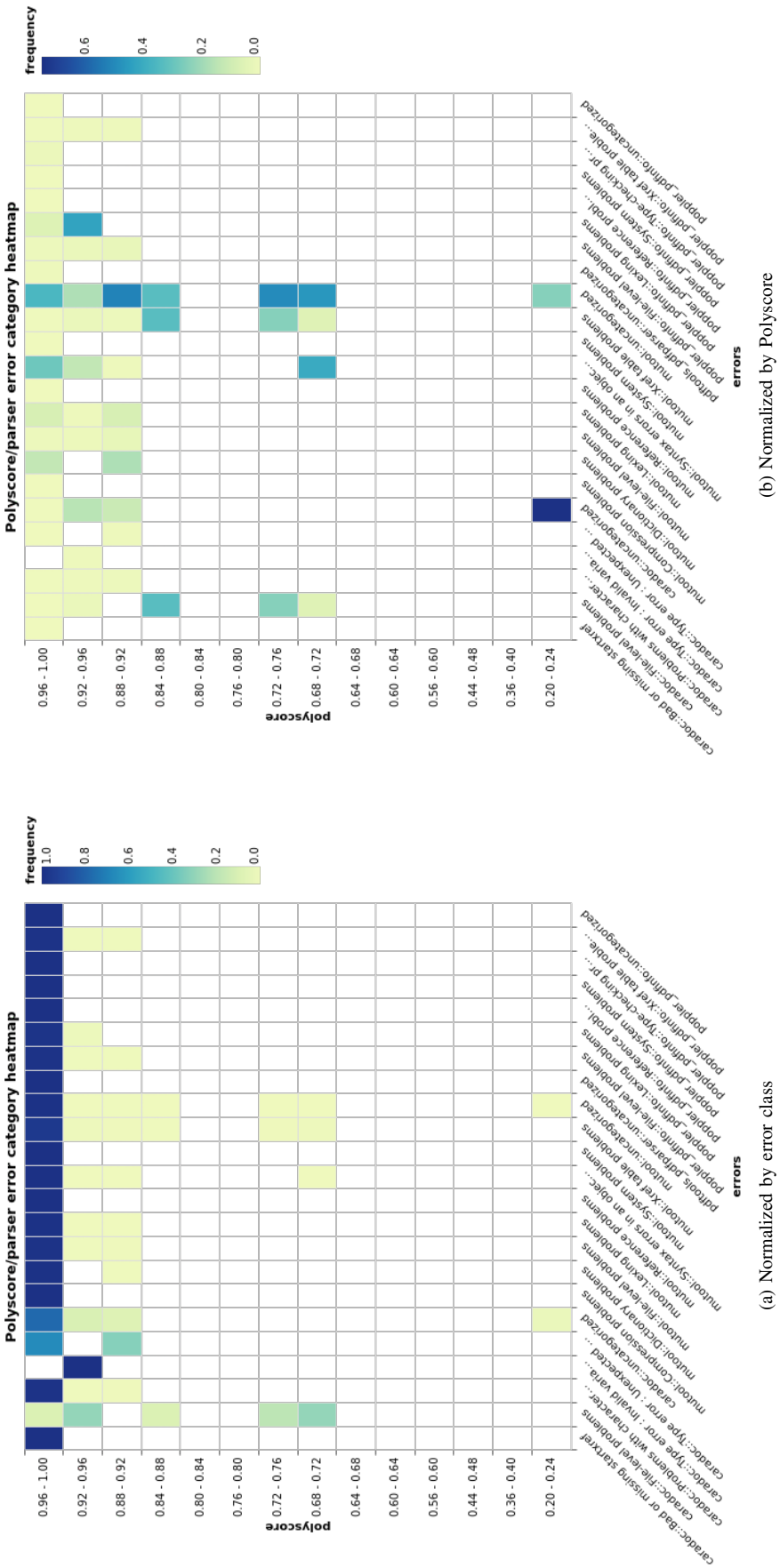
Figure 6: Heatmaps showing files labeled "Mass Mailer" by Polyswarm.



(a) Normalized by error class

(b) Normalized by Polyscore

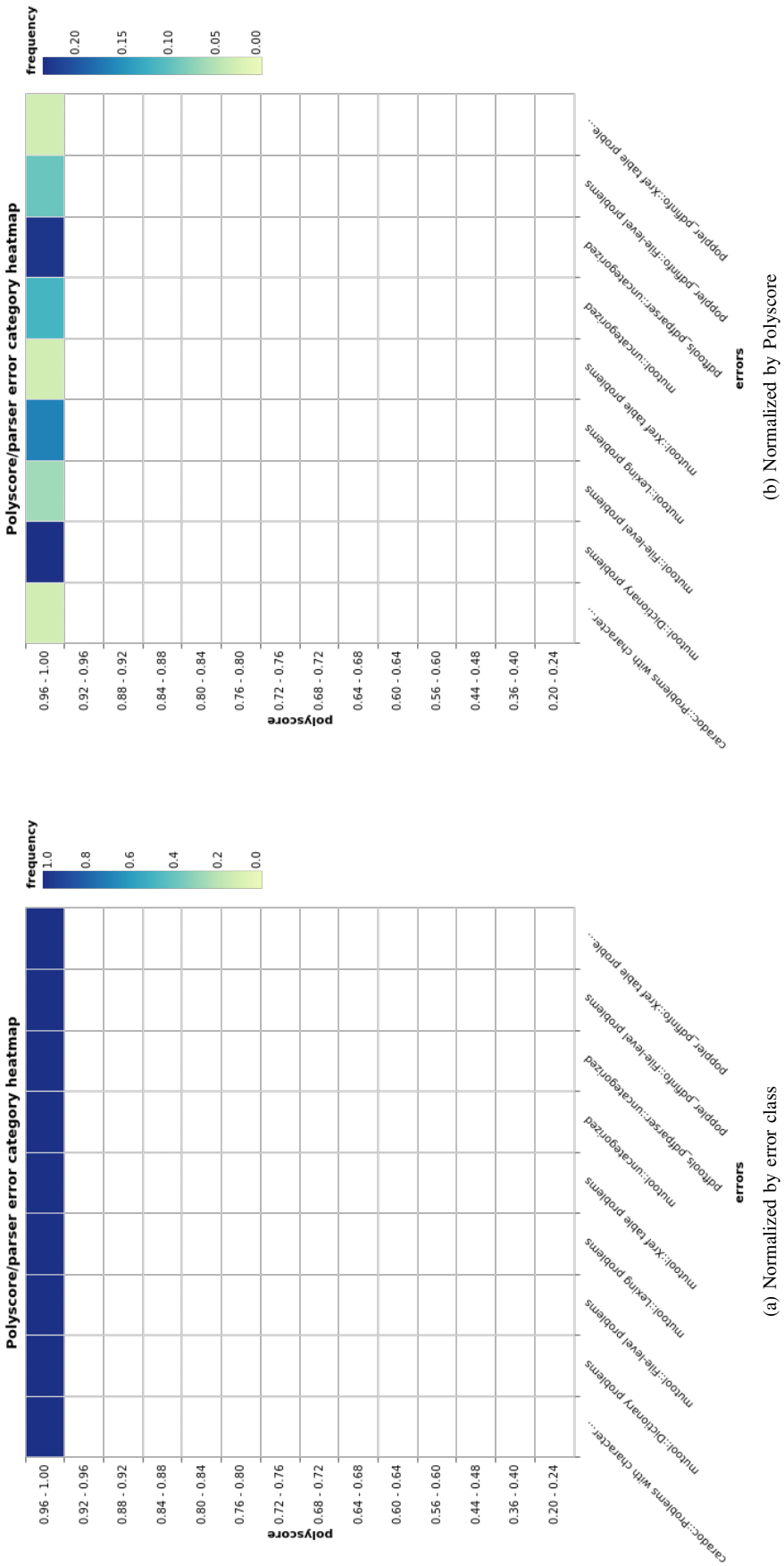
Figure 7: Heatmaps showing files labeled "Security Analysis Tool" by Polyswarm.



(a) Normalized by error class

(b) Normalized by Polyscore

Figure 8: Heatmaps showing files labeled "Virus" by Polyswarm.



(a) Normalized by error class

(b) Normalized by Polyscore

Figure 9: Heatmaps showing files labeled "Cryptominer" by Polyswarm.