BLIND SPOTS: AUTOMATICALLY DETECTING IGNORED PROGRAM INPUTS

HENRIK BRODIN, EVAN SULTANIK, AND MAREK SUROVIČ

Abstract. A *blind spot* is any input to a program that can be arbitrarily mutated without affecting the program's output. Blind spots can be used for steganography or to embed malware payloads. If blind spots overlap file format keywords, they indicate parsing bugs that can lead to differentials. This paper formalizes the operational semantics of blind spots, leading to a technique that automatically detects blind spots based on dynamic information flow tracking. An efficient implementation is introduced an evaluated against a corpus of over a thousand diverse PDFs. There are zero false-positive blind spot classifications and the missed detection rate is bounded above by 11%. On average, at least 5% of each PDF file is completely ignored by the parser. Our results show promise that this technique is an efficient automated means to detect parser bugs and differentials. Nothing in the technique is tied to PDF in general, so it can be immediately applied to other notoriously difficult-to-parse formats like ELF, X.509, and XML.

1. Introduction

We define a *blind spot* as any input to a program that can be arbitrarily mutated without affecting the program's output. For example, the Office Open XML (OOXML) [1] format used by document processing tools like Microsoft Office is based upon the ZIP archive file format. Every valid OOXML file is *also* a valid ZIP that can be extracted using ZIP utilities. Therefore, arbitrary files can be added to an OOXML file as if it were a regular ZIP archive. Any such extraneous files are ignored by the document processor and will therefore be blind spots during processing.

Blind spots are dangerous. For example, blind spots can be exploited for steganography and embedding malware payloads. They can also be indicative of parser differentials, for instance, if two parsers exhibit *different* blind spots for the same input. But they can also be useful: Blind spots can potentially be excluded as candidates for mutation when generating fuzz testing inputs, similar to the Angora fuzzer's branch coverage maximization strategy [6].

Blind spots are a generalization of the concept of *file cavities* [3]: unused spaces in a file format that are created due to the structure of the surrounding data. However, unlike cavities, blind spots may be dependent on the program itself and its execution environment. For example, the image content of a JPEG file will be a blind spot to a parser that only reads its EXIF metadata. Likewise, the EXIF metadata will be a blind spot to a program that converts JPEGs to another image format like BMP that does not support embedded EXIF metadata.

Blind spots are a type of a more general class of data sources we call *quantum inputs*: Bytes that can be *almost* arbitrarily mutated without affecting a program's output. For example, a source code comment is a form of quantum input to a compiler, since the bytes within the comment can be arbitrarily changed without affecting the behavior of the compiler *as long as* the bytes do not contain the comment delimiter. We only consider blind spots in this paper.

Since they are associated with *both* program input *and* the program itself, blind spots can be indicators of parsing bugs. Parsers—particularly hand-generated ones—will often accept a superset of the grammar for which they were designed. This manifests as a parser that accepts some inputs that are technically invalid according to the file format specification. Sometimes this is intentional, in order to maximize compatibility with files generated by other, incorrectly implemented software, or to attempt to repair malformed documents. For example, we discovered that an optimization in MμPDF[1] will sometimes only check the leading "e" in the `endobj` token; it will eagerly accept `eXXXXX` and the PDF will still be parsed correctly, despite the fact that the PDF standard prescribes the existence of the full token. Such lexical permissiveness can lead

[1] `https://mupdf.com/`

to parser differentials and so-called *file format schizophrenia* [2]: when two implementations of a file format interpret the same input file differently.

This paper formalizes the concept of program input blind spots and proposes a novel technique based on dynamic taint analysis to detect them. Our definition clarifies the difference between blind spots and file cavities: Blind spots are the set of input bytes whose data flow never influences either control flow that leads to an output or an output itself. First, we provide a formal definition of blind spots in Section 2. This entails mapping input bytes to the output bytes they influence. Next, in Section 3 we describe the program instrumentation that is able to efficiently capture the data necessary to create this mapping. We then describe how the mapping is constructed from the instrumentation output, as well as how blind spots are inferred from the mapping. We demonstrate how this technique can be used to automatically detect lexical permissiveness and parser bugs. Finally, in Section 4 we provide empirical evidence that these blind spots are correct, and an analysis of blind spots in specific file formats and programs.

2. Definitions and Formalization

In order to formalize the concept of input blind spots and explicitly describe our dynamic taint analysis approach, we propose an extension to Schwartz, Avgerinos, and Brumley's SimpIL operational semantics for dynamic taint propagation [20]. This section describes that extension and uses it to formally define blind spots.

2.1. An Extension to SimpIL.

The original conception of dataflow analysis in SimpIL only tracks whether a given variable or memory cell is tainted, not *from whence* it is tainted. In order to detect blind spots, we need to additionally track exactly which input bytes influence output. For example, consider the pseudocode in Algorithm 1. The variable a is tainted by program input on line 2. Moreover, the value of a can indirectly cause hard-coded data ($d = 5$) to be written to output on line 7 by virtue of the conditional on line 5. Therefore, the first byte of the file *cannot* be a blind spot, since its mutation *can* affect output—despite the fact that the value of the first byte is never written to output. Even if the SimpIL taint policy (*i.e.*, the rules by which taints are propagated) is sufficient to detect that tainted inputs affected the output of the program, it is *insufficient* to detect *which* inputs were responsible. Therefore, we need to extend SimpIL to additionally track the provenance of a taint so that we can map a complete data flow from inputs to outputs.

Algorithm 1 Tainted Control Flow

```
1: procedure TaintedControlFlow
2:     a ← ReadInput(1)          ▷ a is tainted by the 1st byte
3:     b ← ReadInput(1)          ▷ b is tainted by the 2nd byte
4:     c ← a + b
5:     if c ≥ 42 then
6:         d ← 5
7:         WriteOutput(d)
8:     end if
9: end procedure
```

SimpIL uses meta-syntactic variables to represent an execution context. Δ is a mapping of variable names to their values and μ is a mapping from memory addresses to their values. τ_Δ and τ_μ map variable names and memory addresses to booleans ($\mathbf{T}|\mathbf{F}$) defining whether or not that value is tainted in the current execution context.

In order to track taint provenance, we introduce the concept of a *taint label*: a unique identifier for each instance of a tainted variable or memory address in an execution context. We define two types of taint labels: *canonical* and *union*. A canonical taint is the result of a variable or memory address being assigned directly from a program input. A union taint is the result of the combination of two previously tainted values (*e.g.*, the result of two tainted variables being operands in a binary operation).

We extend the SimpIL notation with three new mappings to represent taint labels and track provenance[2]:

[2]The SimpIL semantics are already replete with Greek letters. We have chosen ε from the Greek word for "labels" ($\varepsilon\pi\iota\gamma\rho\alpha\phi\varepsilon\varsigma$), κ from the word for "canonical" ($\kappa\alpha\nu o\nu\iota\kappa o\varsigma$), and γ from the word for "parent" ($\gamma o\nu\varepsilon\upsilon\varsigma$).

$$\frac{\overbrace{v \text{ is input from } src \quad \varepsilon' = \varepsilon[v \leftarrow |\varepsilon| + 1] \quad \kappa' = \kappa[\varepsilon'[v] \leftarrow src]}^{\text{SimpIL notation for the computation performed by the Input operation}}}{\underbrace{\mu, \Delta, \varepsilon, \kappa, \gamma \rightsquigarrow \mu, \Delta, \varepsilon', \kappa', \gamma}_{\substack{\text{SimpIL notation for the} \\ \text{updated execution state after the} \\ \text{operation}}} \qquad \underbrace{\mu, \Delta \vdash \text{get\_input}(src) \Downarrow v}_{\substack{\text{SimpIL notation for the evaluation of} \\ \text{expression get\_input}(src) \text{ to value } v \text{ in} \\ \text{context } \mu, \Delta}}} \textsc{Input}$$

$$\frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \Diamond_b v_2 \quad \varepsilon' = \varepsilon[v' \leftarrow |\varepsilon| + 1] \quad \gamma' = \gamma\big[\varepsilon'[v'] \leftarrow \langle \varepsilon[v_1], \varepsilon[v_2] \rangle\big]}{\mu, \Delta, \varepsilon, \kappa, \gamma \rightsquigarrow \mu, \Delta, \varepsilon', \kappa, \gamma' \quad \mu, \Delta \vdash e_1 \Diamond_b e_2 \Downarrow v'} \textsc{Binop}$$

Figure 1. SimpIL operational semantics for reading input and executing binary operators (see Figure 1 of [20]) updated to include data flow provenance tracking. When a value v is read from an input source src, we create a new, unique canonical taint label for v and set its taint source info in κ to src. When a binary operator $\Diamond_b$ is applied to expressions $e_1 = v_1$ and $e_2 = v_2$ resulting in the value v', we create a new, unique union taint label for v' and set its parents to be the taint labels associated with values v_1 and v_2.

(1) $\varepsilon : \Delta \cup \mu \to \mathbb{N}$ that maps variable names and memory addresses to unique taint labels;
(2) $\kappa : \mathbb{N} \to \{\text{the set of all taint sources}\}$ that maps canonical taint labels to the information about their source (*i.e.*, filename and byte offset); and
(3) $\gamma : \mathbb{N} \to \mathbb{N} \times \mathbb{N}$ that maps union taint labels to their parents.

SimpIL treats these mappings more like programmatic hashmaps than set theoretic functions. As such, SimpIL uses the notation "$\kappa[\ell]$" for the value of taint label ℓ in mapping κ. For brevity, we shall continue this theme by using the notation $\ell \in \kappa$ to represent the fact that ℓ is a key in the mapping κ, and $\ell \notin \kappa$ to mean that ℓ is not a key in κ.

The zero taint label is reserved to represent untainted variables and memory. An untainted variable v will always lack source info and descend from the zero label:

$$\tau_\Delta[v] = \begin{cases} \mathbf{F} & \varepsilon[v] \notin \kappa \wedge \gamma[\varepsilon[v]] = \langle 0, 0 \rangle, \\ \mathbf{T} & \varepsilon[v] \in \kappa \vee (\gamma[\varepsilon[v]] = \langle i, j \rangle \wedge i + j > 0). \end{cases}$$

The SimpIL *taint policy* (see Table III from [20]) and semantics are modified to update these mappings on every taint status change. For example, the updated semantics for reading from input and for executing binary operations are given in Figure 1.

2.2. **Mapping Taint Sources to Sinks.** These mappings allow us to track the entire provenance of a taint in any execution context. The γ mapping implicitly creates a directed acyclic graph (DAG) of labels, representing the dataflow through the program: The program inputs that affect a tainted variable or memory address are its taint label's topmost ancestors in the γ DAG. A recursive function $\psi : \mathbb{N} \to 2^{\{\text{the set of all taint sources}\}}$ can map taint labels to all of their ancestral sources:

$$\psi[\ell] = \begin{cases} \{\kappa[\ell]\} & \ell \in \kappa, \\ \bigcup_{p \in \gamma[\ell]} \psi[p] & \gamma[\ell] \neq \langle 0, 0 \rangle, \\ \emptyset & \text{otherwise.} \end{cases}$$

When we observe that the program writes to output, we use the ψ mapping to record which inputs, if any, tainted the output. This can be accomplished by enumerating the canonical ancestors of the output labels by traversing the γ DAG. This allows us to construct a complete mapping of taint sources to sinks. Note that any taint sources without associated sinks can be arbitrarily mutated without affecting the output.

$$\frac{\overbrace{\mu, \Delta \vdash e \Downarrow v}^{\substack{\text{the conditional expression} \\ e \text{ evaluates to } v}} \quad \overbrace{\varepsilon' = \varepsilon[u \leftarrow |\varepsilon| + \varepsilon[u] : \forall u \in \varepsilon]}^{\substack{\text{create a new taint label} \\ \text{for every existing label}}} \quad \overbrace{\gamma' = \gamma[\varepsilon'[u] \leftarrow \langle \varepsilon[u], \varepsilon[v] \rangle : \forall u \in \varepsilon]}^{\text{union every existing label with the taints of } v}}{\mu, \Delta, \varepsilon, \kappa, \gamma \rightsquigarrow \mu, \Delta, \varepsilon', \kappa, \gamma'} \textsc{PreCond}$$

FIGURE 2. Updated SIMPIL operational semantics to enforce the taint policy that every input that affects control-flow will be unioned with all labels created in the branch they influence. The PRECOND rule is executed before every conditional rule (TCOND and FCOND in Figure 1 of [20]).

2.3. **A Definition of Program Input Blind Spots.** Let Ω be the set of taint labels written to output during execution. Then a blind spot is the set of all potential program inputs that did not affect the output:

$$\overbrace{\left\{\langle \text{source info} \rangle : (\forall \ell \in \kappa : \kappa[\ell] \neq \langle \text{source info} \rangle)\right\}}^{\substack{\text{all } \textit{potential} \text{ input that is } \textit{not} \text{ consumed by the program} \\ (\textit{e.g.}, \text{ a portion of an input file that is never read})}} \bigcup \underbrace{\left\{\kappa[\ell] : \ell \in \kappa \wedge (\forall \ell' \in \Omega : \ell \notin \psi[\ell'])\right\}}_{\text{the sources of all canonical taints that did not affect output}}. \tag{1}$$

As we mentioned above, SIMPIL includes a *taint policy* specifying the rules by which taints are propagated. The taint policy will affect our definition of blind spots. For example, let us again consider the pseudocode in Algorithm 1. The variable a is tainted by program input on line 2, which we can now specify in our SIMPIL extension as $\kappa[\varepsilon[a]] \neq \emptyset$. Recall that the value of a can indirectly cause hard-coded data ($d = 5$) to be written to output on line 7. Therefore, a cannot be a blind spot, since its mutation can affect output. However, the semantics by which the taint policy propagates taint through conditionals will affect whether our extension of SIMPIL will consider the output to be tainted by a, because the value of a itself is never written to output.

We resolve this discrepancy by enforcing the following constraint on blind spot taint policies: The taint labels of every variable and memory address that affect the program's control-flow will be unioned with all labels created in the branch they influence. In other words, in addition to the definition of blind spots in Equation (1), a blind spot cannot influence control flow that leads to a program output. Updated operational semantics for the conditional operator that implement this policy are given in Figure 2.

A trace of Algorithm 1 showing the iterative updates to the execution context is given in Table 1. It demonstrates how the blind spot taint policy propagates taints from variables in the path condition—variables that have affected control flow leading to the current state (*e.g.*, variable c on line 5)—to variables that would otherwise not be tainted (*e.g.*, variable d). This reduces false-positive blind spot classifications, since it captures tainted variables that indirectly cause output.

3. TRACKING TAINTS

Dynamic Information Flow Tracking (DIFT), also known as Dynamic Taint Analysis (DTA), is a technique in which the flow of information through a program is modeled and tracked at runtime. DIFT is a challenging problem; many modern approaches suffer from high implementation overhead, low accuracy, and/or low fidelity [5]. *Universal taint analysis* is a form of DIFT that can track all input bytes throughout the execution of a program, mapping inputs to outputs [24].

Thus far we have developed formal semantics for blind spots and discovered some necessary taint propagation policies to detect them. The next step is to automatically instrument a parser to extract the data flow information necessary to classify input byte regions as blind spots. We gather this data flow information by performing universal taint analysis.

3.1. **Related Work.** There are significant challenges when performing universal taint analysis on real-world software. When the cyclomatic complexity of a program is large, the amount of new taint labels generated

Line	Statement	Δ	τ_Δ	ε	κ	γ
1	start	{}	{}	{}	{}	{}
2	$a \leftarrow$ ReadInput(1)	$\{a \to 40\}$	$\{a \to \mathbf{T}\}$	$\{a \to 1\}$	$\{1 \to \langle 1^{\text{st}} \text{ byte of input}\rangle\}$	{}
3	$b \leftarrow$ ReadInput(1)	$\{ a \to 40, b \to 12 \}$	$\{ a \to \mathbf{T}, b \to \mathbf{T} \}$	$\{ a \to 1, b \to 2 \}$	$\{ 1 \to \langle 1^{\text{st}} \text{ byte of input}\rangle, 2 \to \langle 2^{\text{nd}} \text{ byte of input}\rangle \}$	{}
4	$c \leftarrow a + b$	$\{ a \to 40, b \to 12, c \to 52 \}$	$\{ a \to \mathbf{T}, b \to \mathbf{T}, c \to \mathbf{T} \}$	$\{ a \to 1, b \to 2, c \to 3 \}$	$\{ 1 \to \langle 1^{\text{st}} \text{ byte of input}\rangle, 2 \to \langle 2^{\text{nd}} \text{ byte of input}\rangle \}$	$\{3 \to \langle 1, 2\rangle\}$
5	**if** $c \geq 42$ **then**	$\{ a \to 40, b \to 12, c \to 52 \}$	$\{ a \to \mathbf{T}, b \to \mathbf{T}, c \to \mathbf{T} \}$	$\{ a \to 1, b \to 2, c \to 3 \}$	$\{ 1 \to \langle 1^{\text{st}} \text{ byte of input}\rangle, 2 \to \langle 2^{\text{nd}} \text{ byte of input}\rangle \}$	$\{3 \to \langle 1, 2\rangle\}$
6	$d \leftarrow 5$	$\{ a \to 40, b \to 12, c \to 52, d \to 5 \}$	$\{ a \to \mathbf{T}, b \to \mathbf{T}, c \to \mathbf{T}, d \to \mathbf{T} \}$	$\{ a \to 1, b \to 2, c \to 3, d \to 4 \}$	$\{ 1 \to \langle 1^{\text{st}} \text{ byte of input}\rangle, 2 \to \langle 2^{\text{nd}} \text{ byte of input}\rangle \}$	$\{ 3 \to \langle 1, 2\rangle, 4 \to \langle 3, 0\rangle \}$

Table 1. Execution context trace for Algorithm 1. Note on line 6 that, despite being assigned a constant value of 5, the d variable (taint label 4) is in fact tainted by variable c (taint label 3). This is because the path condition to line 6 depends on c from the conditional branch on line 5. Therefore, neither of the first two bytes of input are blind spots.

from even a small input is enormous. For example, many document formats such as PDF use compression to keep file sizes small. When tracking data flow through a PDF parser, there will be a significant number of taint unions as the data are decompressed. This phenomenon is known as *taint explosion*, which generally occurs when a function performs a large number of combinatorial operations on input data.

There are several existing projects that achieve universal taint tracking, using various methods. Two of the best maintained and easiest to use are AUTOGRAM [14] and TaintGrind [16]. However, the former is limited to analysis within the Java virtual machine and the latter suffers from unacceptable runtime overhead when tracking as few as several bytes at a time. For example, we ran `mutool`, a utility in the MμPDF project[3], using TaintGrind over a corpus of medium sized PDFs, and in every case the tool had to be halted after over twenty-four hours of execution for operations that would normally complete in milliseconds without instrumentation.

There are also existing tools for performing dynamic program analysis via QEMU [4], such as PANDA [11] and DECAF(++) [8], both of which have taint tracking extensions. However, being an emulation framework rather than virtualization, QEMU incurs a runtime overhead of about 15% just to execute a binary, not including any instrumentation [15]. After adding the program instrumentation necessary to enable fuzz testing, QEMU was observed to have over three times the runtime overhead of equivalent compile-time instrumentation [17].

Symbolic execution engines like Triton [19] and SymCC [18] have also been used for data flow analysis. Symbolic execution could be extended to detect blind spots, *e.g.*, by making all input bytes symbolic and observing all data that is written. The input bytes associated with any symbolic data that is either written or included in the path condition during a write *is not* a blind spot. However, since each branch of a conditional affected by program input must be enumerated, symbolic execution is very likely to be intractable for large inputs.

DRTaint [24] is a recently published tool that can also perform universal taint tracking. It adds a minimal amount of runtime instrumentation to create runtime artifacts that can be post-processed to extract any data flow. The authors do not quantify the exact overhead of DRTaint, but Figure 5 from their paper suggests at least a 60x slowdown compared to the uninstrumented program. It is also unclear whether this instrumentation was sufficient to reconstruct all data flows. This is consistent with earlier techniques such as Dytan [7] that reported a 50x slowdown when tracking as few as 64 taint labels.

3.2. **PolyTracker.** This section introduces PolyTracker [21], a novel LLVM-based dynamic analysis tool for extracting ground truth information from programs. It is open-source and available at `https://github.com/trailofbits/polytracker`.

[3] https://mupdf.com/

PolyTracker automatically adds instrumentation to a program such that, when the program is executed, it produces runtime artifacts that can be analyzed to track the data flows of all input bytes. It is an extension of the LLVM DataFlowSanitizer (DFSan) [9], a generalized dynamic data flow analysis instrumentation tool. PolyTracker has previously been used to label the semantic purpose of functions in a parser [13].

Originally, DFSan only supported tracking at most 2^{16} taint labels at a time[4]. This restriction was acceptable for DFSan's primary use case at the time: data flow analysis for fuzz testing. However, this restriction was insufficient for our goal of tracking the taints of every input byte simultaneously. By replacing DFSan's dense matrix representation of taint unions with a DAG (denoted as the γ mapping in our semantics above), we are able to simultaneously track 2^{31} taints—an increase of several orders of magnitude. This proved sufficient to detect blind spots in all programs and inputs on which we have experimented.

4. Empirical Analysis

The previous sections introduced a method for identifying the blind spots of a program input. How accurate is this blind spot classifier? Since we do not have pre-labeled ground truth for the blind spots of an input, we need to develop statistical estimates for the confusion matrix of our classifier.

We focus our evaluation on the PDF file format, for several reasons.

(1) The PDF file format is old and complex, has had many revisions, and enjoys numerous independent implementations. This has led to differentials that necessitate lexical permissiveness for interoperability [23].
(2) PDF is a container format that allows embedding of other formats like JPEG, providing more opportunity for blind spots.
(3) The GovDocs corpus [12] provides thousands of real-world PDFs generated by a diversity of software.

We instrumented the popular MμPDF utility using PolyTracker to detect blind spots. Next, we ran the instrumented utility on 1,087 PDFs sampled from the GovDocs corpus to render the PDFs to PostScript. The PDFs totaled over 622 MB and averaged 572 KB each. The largest file was 9.6 MB. We discovered a total of 33.5 MB of blind spots, averaging 30.8 KB per file, some files having zero, and one file having 1.07 MB of blind spots.

PostScript, a vector image format, was chosen rather than a raster format like JPEG or PNG because the relative lack of compression would help prevent explosion of taint union labels and thereby reduce runtime. Runtime of the instrumented program was less than one minute for each PDF. We would expect to get similar results when rendering to JPEG or PNG, however, since blind spots are dependent on execution, there could be some discrepancies. For example, since fonts can be embedded in both PDF and PostScript but cannot be embedded in JPEG or PNG, one might expect to see more blind spots related to fonts if one were to have rendered to a different file format.

4.1. Classification Error.

We validate blind spots by iteratively mutating each classified blind spot byte in the input file and re-running the original, uninstrumented program again. See Figure 3 for a notional example of this mutation process. If the PostScript output produced from the mutated input is different from the output of the unmodified file, then our classified blind spot is incorrect (Type I error), since any mutation inside a blind spot should, by definition, not affect program output.

We also sample bytes from *outside* of our classified blind spots and mutate them, similarly. If a byte outside a blind spot can be arbitrarily mutated, it is likely a missed detection (Type II error). However, it is not tractable to mutate and verify all possible combinations of input bytes, since this would amount to testing the power set of all bytes, running in $\Theta(2^n)$ time. It might be the case that a byte outside a blind spot is in fact a quantum byte that can be almost arbitrarily mutated, but has some undetected data dependency on another byte. Therefore, our reported Type I error rate is a tight bound on the actual Type I error, but our Type II error rate is a loose upper bound on the true Type II error.

We mutated all 33.5 million blind spot bytes classified in the corpus, and all mutated blind spots produced identical output to the original file for a 0% false-positive rate. Of the bytes *not* classified as blind spots, 89% did affect the output. Therefore, the false-negative rate is bounded above by 11%. A significant number

[4]Over the course of 2021, DFSan underwent a significant refactor in order to make its memory layout compatible with other LLVM sanitizers [10]; this refactor reduced the effective number of taints it could track to $2^3 = 8$, which is still sufficient for most fuzzing use cases, but not for detecting blind spots.

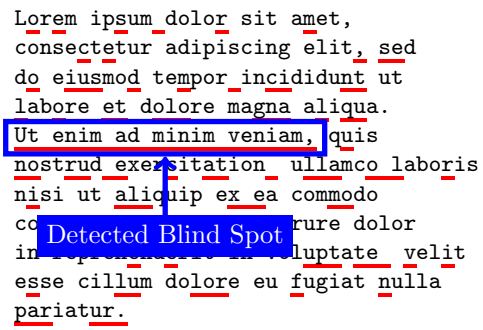


FIGURE 3. Notional example of validating detected blind spots. Underlined bytes are iteratively mutated and re-parsed. Bytes outside of blind spots are randomly selected for mutation, but all bytes within a blind spot are mutated. If a byte *inside* a blind spot is mutated and produces a different result, then that blind spot classification was a false-positive (Type I error). If a byte *outside* a blind spot can be arbitrarily mutated, then it is a missed detection (Type II error).

BYTES	# Blind Spots	Total Frequency
\r	561451	6745095
\n	52175	4996436
\x20	47752	17967270
\x00	9504	7658663
\x11	5232	3325230
\x08	3930	3611559
\x06	3629	3104109
\x09	3572	2981341
%	3145	3041586
\x12	2932	2847588
#	2859	3399908
\x05	2772	2873564
\x02	2640	3007837
\x0F	2607	2944161
\x14	2594	3132850
\xF0	2351	2930162
a	2348	4959732
!	2271	3172472
\xA3	2108	2783202
4	2074	4848767
\x13	2024	2788459

TABLE 2. The twenty most common blind spot prefixes of length at most seven bytes.

of these missed detections are likely quantum bytes: Pieces of data that can be mutated *almost* arbitrarily, but have some data dependency that can affect output that our random mutations did not exercise.

4.2. **Blind Spot Content.** What is the content of PDF blind spots?

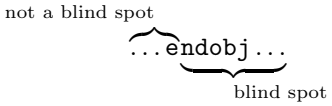
In the GovDocs PDF corpus, we detect 63,194 unique blind spot prefixes of length at most seven bytes, and 338,943 unique byte sequences of length at most seven that precede blind spots.

Consider the bytes that occur at the start of a blind spot; the most common of these are listed in Table 2. They are all one byte long, meaning that there is a diversity of content at the start of blind spots. The most common byte sequences that *precede* a blind spot, listed in Table 3, are more interesting: most are multi-byte, and they comprise many PDF tokens like `endobj`. This means that bytes following certain tokens are often or always ignored by the parser.

Bytes	# Blind Spots	Total Frequency
`n`	300057	5811224
`\ n`	299785	629959
`00000\ n`	299621	555279
`f`	236173	3738217
`\ f`	235736	552315
`65535\ f`	205564	470855
`m`	66860	4242668
`dstream`	66588	189282
`00001\ f`	26343	44612
`\r`	12839	6745095
`endobj\r`	11001	527199
`e`	7056	7495303
`be`	6419	41673
`Adobe`	6418	11545
`\x00\x0EAdobe`	6417	9142
`\x02`	5533	3007837
`\x08`	4185	3611559
`\x00\x02`	3787	82034
`00000\ f`	3770	6838
`\x01\x00\x02`	3707	29366

Table 3. The twenty most common byte sequences of length at most seven preceding a blind spot.

Now let us consider the unique suffix/prefix pairs that occur adjacent to the start of a blind spot. There are 1,029,129 unique pairs of these byte sequences. If we sort them by frequency, the pair



is in the top 0.01% of such pairs. "`endobj`" is a PDF token used to delimit the end of an object in the document model. The fact that this token is split across a blind spot boundary so frequently is indicative of, at best, intentional lexical permissiveness on the part of the parser, and, at worst, a bug. Other interesting blind spot contexts within the top hundredth of the first percentile include the entire `endobj` token, if preceded by a carriage return. Any whitespace after the `stream` token is ignored. The `obj` token is completely ignored if preceded by a space. Similarly, the PDF dictionary delimiters << and >> are frequently skipped, *e.g.*, at the beginning of a PDF object. This simple contextual blind spot frequency analysis can discover parsing errors and differentials.

4.3. **Blind Spot Context.** How frequent are blind spots, and where do they occur in PDFs?

Figure 4 plots the number of blind spots in the PDF corpus as a function of file size. This suggests that the number of blind spot bytes in a typical PDF is constant. Note, however, that blind spots in PDFs can be arbitrarily large, since the PDF format permits the inclusion of arbitrary binary blobs that do not have to be connected to the document object model (DOM) [23].

Figure 5 plots a histogram of the contiguous size of blind spot regions in the corpus. The majority of blind spots are small, but a nontrivial number of blind spots are over 1 KB. The average blind spot is 42 bytes long with a standard deviation of 1.72 KB.

Figure 6 is a histogram of the normalized position of blind spot bytes in their files: the blind spot's byte offset divided by the file size. In our experiments with the MμPDF renderer translating to PostScript, the majority of PDF blind spots are at the beginning and ends of the files.

Figure 7 combines the two previous figures by comparing the mean contiguous blind spot size to the normalized position in the PDF. Despite the most blind spot bytes being at the beginning and ends of the PDFs, the *longest* blind spots tend to be in the first 10–20% of the file, but not immediately at the beginning.

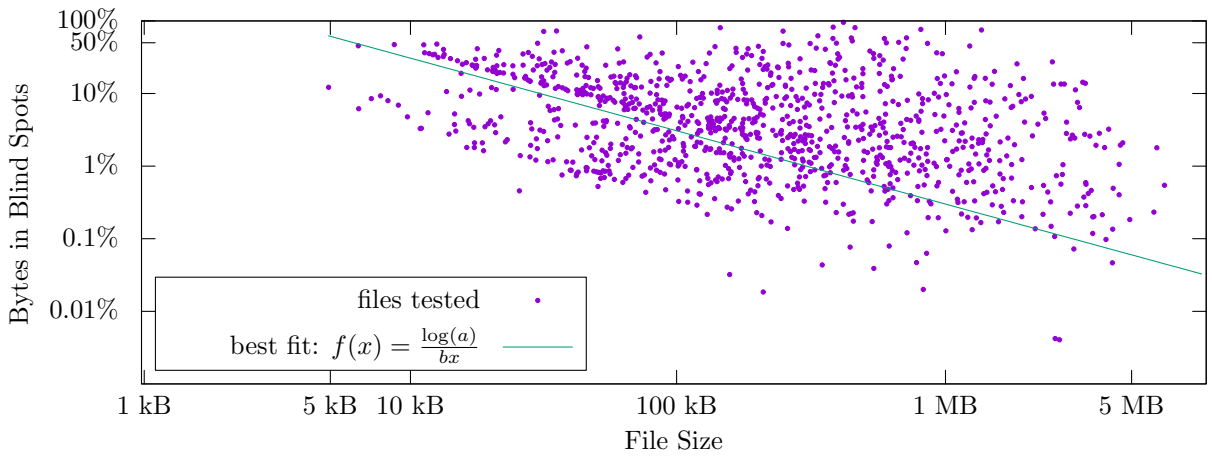


FIGURE 4. Proportion of PDF bytes that are in blind blind spots as a function of file size.

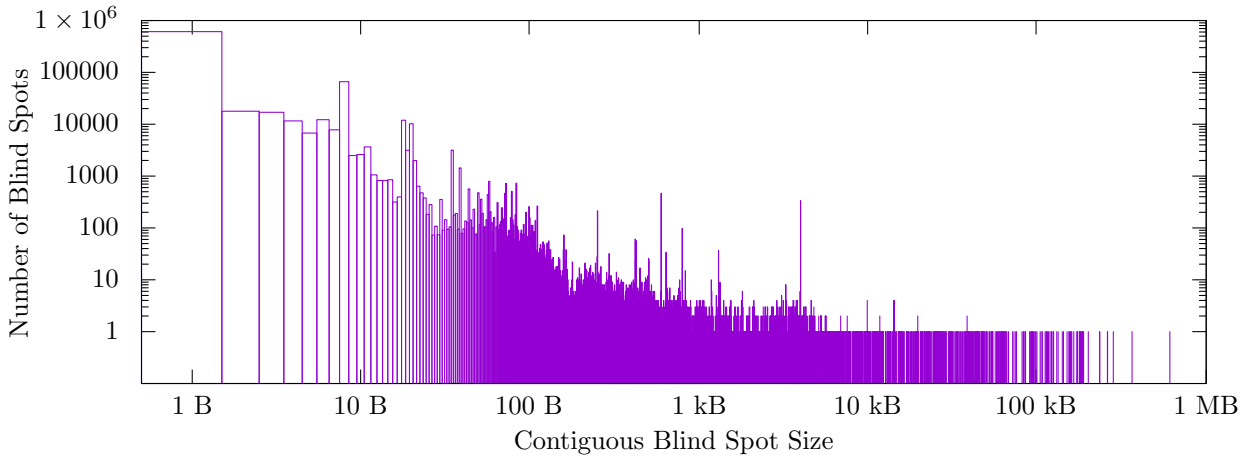


FIGURE 5. Histogram of the sizes of contiguous PDF blind spots. The majority of blind spots are single bytes, but a nontrivial number of blind spots are over 1 kilobyte. The average blind spot is 42 bytes long with a standard deviation of 1.72 kilobytes.

For each input PDF, we generate a parse tree using PolyTracker's sister tool, PolyFile[5] [21]. Each byte in the input PDF corresponds to one or more *parse tree derivations*: unique paths through the PDF parse tree. A byte could have more than one derivation, for instance, if the input file is a *polyglot*—a file that is valid in two or more formats. PDFs are particularly easy to turn into polyglots, and many legitimate PDF generators exploit this fact. For example, it is common to produce valid PDFs that are *also* valid ZIP archives that, when extracted, contain additional files related to the document. Therefore, a byte might have one derivation in the PDF parse tree and have a different but completely valid derivation in the ZIP parse tree. An example of a parse tree derivation is given in Figure 8.

For each unique parse tree derivation, we count the number of blind spot bytes that occur in that derivation. The most frequent derivation containing blind spots is `application/pdf`, the root of the PDF parse tree. This means that the majority of PDF blind spots occur in portions of the file that have no semantic purpose. Blind spot locality might be explained by the fact that PDF parsers are resilient to both leading and trailing

[5]Open-source and available at `https://github.com/trailofbits/polyfile`.

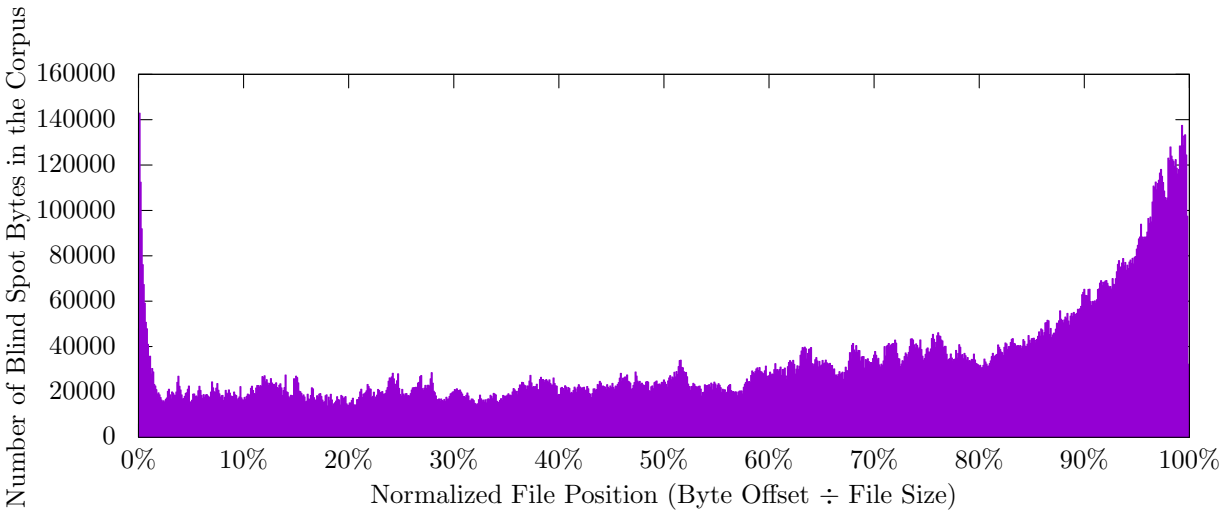


FIGURE 6. Normalized position of blind spots in PDF files. With the MμPDF renderer, blind spots are most frequent at the beginning and end of PDF files.

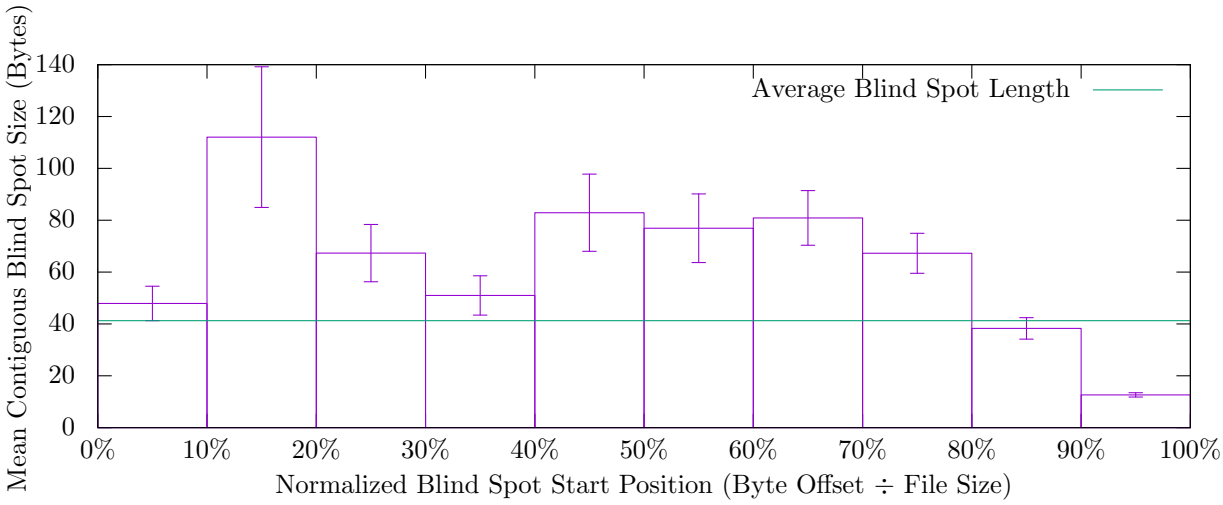


FIGURE 7. Contiguous blind spot size as a function of its position in the PDF file. Error bars correspond to the standard deviation of blind spot sizes in that portion of the file. The longest blind spots tend to be in the second tenth of the file.

application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFList → PSInt.

FIGURE 8. The parse tree derivation of a byte representing an integer in a list of lists that is a value in a PDF dictionary in a PDF object.

garbage bytes before and after the PDF file. Also, as we saw above, the majority of naturally occurring blind spot bytes are at the beginning and end of the file.

The frequency of every unique parse tree derivation is presented in Figure 9. Blind spots overwhelmingly occur in a small number of derivations. However, the long tail demonstrates that blind spots can and do occur in many diverse derivations.

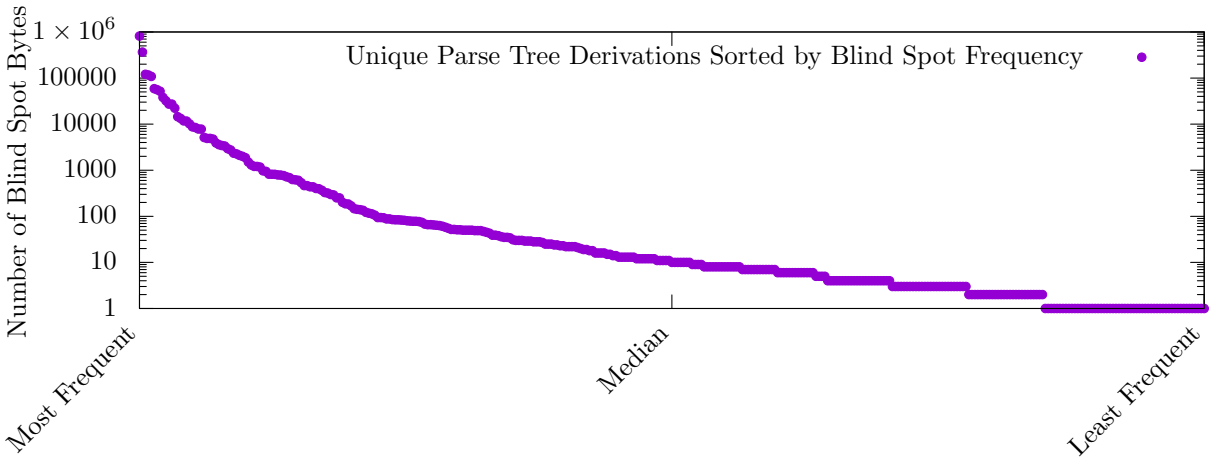


FIGURE 9. Each point is a *parse tree derivation*—a unique path through a PDF parse tree—whose y-axis value is the number of times a blind spot occurred in that derivation. Blind spots overwhelmingly occur in a small number of derivations, yet there is a long tail demonstrating that blind spots can and do occur in many diverse derivations.

The most frequent parse tree derivations for PDF blind spots are given in Table 4. All but the most frequent derivation descend from the PDFObject node. This is unsurprising, since PDF objects can contain streams of arbitrary binary data. PDF objects also do not need to be connected to the root of the PDF document object model, nor do they need to be used in any way for rendering.

The second most frequent derivation for blind spots are in PDF dictionaries. PDF dictionaries can contain arbitrary key/value pairs which are often used for metadata that is not necessary for rendering (*e.g.*, timestamps). Dictionaries can and often do contain redundant information. For example, the length of a PDF object stream can either be specified as a key/value pair in the preceding object dictionary or implicitly defined by the location of a required termination token. If both are specified, then they must agree. However, if a length is specified in the dictionary which *does not* agree with the position of the termination token, then most parsers will ignore the specified length and defer to the token position [22], making the dictionary entry a blind spot.

5. CONCLUSIONS

This paper defined the concept of blind spots: inputs to a program that can be arbitrarily mutated without affecting the program's output. Operational semantics for blind spots were formalized by extending SIMPIL [20]. An efficient implementation capable of automatically detecting blind spots, PolyTracker, was introduced. It works by adding instrumentation for performing dynamic information flow tracking (DIFT) to a program.

The technique was evaluated by detecting blind spots in the popular MμPDF parser over a corpus of over a thousand diverse PDFs [12]. There were zero false-positive blind spot classifications, and the missed detection rate was bounded above by 11%. On average, at least 5% of each PDF file was completely ignored by the parser; blind spots that could be repurposed for steganography or embedding malware payloads.

Future work includes extending the approach to detect quantum bytes: inputs that can *almost* arbitrarily be mutated without affecting output, like source code comments. The current implementation injects its DIFT instrumentation at the LLVM/IR level. Therefore, it is limited to programs that can be compiled using LLVM, or binaries that can be lifted to LLVM/IR. It would be useful to apply the technique to runtime instrumentation that could be applied to a black-box binary.

Our results show promise that this technique could be an efficient automated means to detect parser bugs and differentials. Nothing in the technique is tied to PDF in general, so it can be immediately applied to other notoriously difficult-to-parse formats like ELF, X.509, and XML.

Derivation	# Bytes
application/pdf	815891
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Key	365771
application/pdf → PDFObject → PSBytes → image/jpeg	120803
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFObjRef	116538
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFLiteral	108573
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PSInt	58864
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFObjRef	55920
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PSInt	51693
application/pdf → PDFObject → PDFList → PDFObjRef	37982
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFLiteral	32221
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFDictionary → KeyValuePair → Key	27220
application/pdf → PDFObject → FlateDecode → DecodedStream → PSBytes → application/octet-stream	27210
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PSFloat	22254
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFList → PSInt	14489
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PSBytes → text/plain	13300
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PSFloat	11781
application/pdf → PDFObject → FlateDecode	11634
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFDictionary → KeyValuePair → Value → PDFLiteral	10148
application/pdf → PDFObject → PSBytes → image/jp2	8738
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFDictionary → KeyValuePair → Key	8445
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFDictionary → KeyValuePair → Value → PSFloat	7849
application/pdf → PDFObject → PSBytes → application/octet-stream	7800
application/pdf → PDFObject → PSBytes → text/plain	5131
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PSBytes → text/plain	4901
application/pdf → PDFObject → PDFDeciphered → image/jpeg	4900
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFDictionary → KeyValuePair → Value → PSInt	4741
application/pdf → PDFObject → LZWDecode → DecodedStream → PSBytes → application/octet-stream	3914
application/pdf → PDFObject → PDFDictionary → KeyValuePair → Value → PDFList → PDFDictionary → KeyValuePair → Value → PDFLiteral	3583

Table 4. The PDF parse tree derivations containing the most blind spot bytes. These primarily descend from PDFObject.

Acknowledgment

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) SafeDocs program as a subcontractor to Galois under HR0011-19-C-0073. Many thanks to Michael Brown, Trent Brunson, Filipe Casal, Peter Goodman, Kelly Kaoudis, Bill Harris, Nichole Schimanski, Mark Tullsen, Walt Woods, Peter Wyatt, and Sergey Bratus for their invaluable feedback on the approach and tooling. Special thanks to Carson Harmon, the original creator of PolyTracker, whose ideas and discussions germinated this research.

References

1. ISO/IEC 29500-1:2016, *Information technology—Document description and processing languages—Office Open XML File Formats—Part 1: Fundamentals and Markup Language Reference*, Standard, International Organization for Standardization, Geneva, CH, November 2016.
2. Ange Albertini, *Abusing file formats; or, Corkami, the novella*, The International Journal of Proof of Concept or GTFO **0x07** (2015), no. 6, 18–41.

3. Ange Albertini, Thai Duong, Shay Gueron, Stefan Kölbl, Atul Luykx, and Sophie Schmieg, *How to abuse and fix authenticated encryption without key commitment*, Proceedings of the 31st USENIX Security Symposium (Boston, MA), USENIX Association, August 2022.
4. Fabrice Bellard, *QEMU, a fast and portable dynamic translator*, Proceedings of the Annual USENIX Technical Conference (USA), ATEC '05, USENIX Association, 2005, p. 41.
5. Christopher Brant, Prakash Shrestha, Benjamin Mixon-Baca, Kejun Chen, Said Varlioglu, Nelly Elsayed, Yier Jin, Jedidiah Crandall, and Daniela Oliveira, *Challenges and opportunities for practical and effective dynamic information flow tracking*, ACM Computing Surveys **55** (2021), no. 1.
6. Peng Chen and Hao Chen, *Angora: Efficient fuzzing by principled search*, Proceedings of the IEEE Symposium on Security and Privacy, 2018, pp. 711–725.
7. James Clause, Wanchun Li, and Alessandro Orso, *Dytan: A generic dynamic taint analysis framework*, Proceedings of the 2007 International Symposium on Software Testing and Analysis (New York, NY, USA), ISSTA '07, Association for Computing Machinery, 2007, pp. 196–206.
8. Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin, *DECAF++: Elastic Whole-System dynamic taint analysis*, 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019) (Chaoyang District, Beijing), USENIX Association, September 2019, pp. 31–45.
9. *DataFlowSanitizer*, `https://clang.llvm.org/docs/DataFlowSanitizer.html`, Accessed: 2020-07-26.
10. *[DFSan] change shadow and origin memory layouts to match MSan.*, LLVM commit `45f6d5522f8d`, `https://reviews.llvm.org/D104896?id=354633`, Accessed: 2020-07-26.
11. Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan, *Repeatable reverse engineering with PANDA*, Proceedings of the 5th Program Protection and Reverse Engineering Workshop (New York, NY, USA), PPREW-5, Association for Computing Machinery, 2015.
12. Simson Garfinkel, Paul Farrell, Vassil Roussev, and George Dinolt, *Bringing science to digital forensics with standardized forensic corpora*, Digital Investigation **6** (2009), S2–S11, Proceedings of the Ninth Annual DFRWS Conference.
13. Carson Harmon, Bradford Larsen, and Evan Sultanik, *Toward automated grammar extraction via semantic labeling of parser implementations*, Proceedings of the Sixth Workshop on Language-Theoretic Security (LangSec), IEEE Symposium on Security and Privacy, 2021.
14. Matthias Höschele and Andreas Zeller, *Mining input grammars from dynamic taints*, Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (New York, NY, USA), ASE 2016, Association for Computing Machinery, 2016, pp. 720–725.
15. Ahmed Karaman, *Measuring QEMU emulation efficiency*, `https://ahmedkrmn.github.io/TCG-Continuous-Benchmarking/Measuring-QEMU-Emulation-Efficiency/`, August 2020, Accessed: 2020-07-26.
16. W. M. Khoo, *Taintgrind: a Valgrind taint analysis tool*, `https://github.com/wmkhoo/taintgrind`, Accessed: March 2, 2020.
17. Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks, *Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing*, 30th USENIX Security Symposium (USENIX Security 21), USENIX Association, August 2021, pp. 1683–1700.
18. Sebastian Poeplau and Aurélien Francillon, *Symbolic execution with SymCC: Don't interpret, compile!*, 29th USENIX Security Symposium (USENIX Security 20), USENIX Association, August 2020, pp. 181–198.
19. Florent Saudel and Jonathan Salwan, *Triton: A dynamic symbolic execution framework*, Symposium sur la sécurité des technologies de l'information et des communications (Rennes, France), SSTIC, June 2015, pp. 31–54.
20. Edward J. Schwartz, Thanassis Avgerinos, and David Brumley, *All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)*, Proceedings of the IEEE Symposium on Security and Privacy, 2010, pp. 317–331.
21. Evan Sultanik, Brad Larsen, and Carson Harmon, *Two new tools that tame the treachery of files*, `https://blog.trailofbits.com/2019/11/01/two-new-tools-that-tame-the-treachery-of-files/`, November 1, 2019, Accessed: January 12, 2020.
22. Julia Wolf, *OMG WTF PDF—PDF ambiguity and obfuscation*, Proceedings of TROOPERS (Heidelberg, Germany), March 2011.
23. Peter Wyatt, *Work in progress: Demystifying PDF through a machine-readable definition*, Proceedings of the Seventh Workshop on Language-Theoretic Security (LangSec), IEEE Symposium on Security and Privacy, 2021.
24. Pan Yang, Fei Kang, Yuntian Zhao, and Hui Shu, *DRTaint: A dynamic taint analysis framework supporting correlation analysis between data regions*, Journal of Physics: Conference Series **1856** (2021), no. 1, 012013.

Trail of Bits, Stockholm, Sweden
Email address: `henrik.brodin@trailofbits.com`

Trail of Bits, Philadelphia, USA
Email address: `evan.sultanik@trailofbits.com`

Trail of Bits, Brno, Czech Republic
Email address: `marek.surovic@trailofbits.com`