# PolyTracker: Whole-Input Dynamic Information Flow Tracing


Evan Sultanik*
Trail of Bits
Philadelphia, PA, USA
evan.sultanik@trailofbits.com

Marek Surovič*
Trail of Bits
Brno, Czechia
marek.surovic@trailofbits.com

Henrik Brodin*
Trail of Bits
Stockholm, Sweden
henrik.brodin@trailofbits.com

Kelly Kaoudis*
Trail of Bits
Boulder, CO, USA
kelly.kaoudis@trailofbits.com

Facundo Tuesca*
Trail of Bits
Amsterdam, Netherlands
facundo.tuesca@trailofbits.com

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

Lisa Overall*‡
Trail of Bits
Washington, DC, USA
lmovera@super.org

Joseph Sweeney*
Trail of Bits
New York, NY, USA
joe.sweeney@trailofbits.com

Bradford Larsen*§
Trail of Bits
Groton, MA, USA
brad@bradfordlarsen.com



## Abstract

We present PolyTracker, a whole-program, whole-input dynamic information flow tracing (DIFT) framework. Given an LLVM compatible codebase or a binary that has been lifted to LLVM intermediate representation (IR), PolyTracker compiles it, adding static instrumentation. The instrumented program will run normally with modest performance overhead, but will additionally output a runtime trace artifact in the co-designed TDAG (Tainted Directed Acyclic Graph) format. TDAGs can be post-processed for a variety of analyses, including tracing every input byte through program execution. TDAGs can be generated either by running the program over a corpus of inputs or by employing a randomized input generator such as a fuzzer. PolyTracker traces (TDAGs) are useful not only for very localized, targeted dynamic program analysis as with smaller-scale DIFT: TDAGs are primarily intended for whole-program runtime exploration and bug finding, granular information-flow diffing between program runs, and comparisons of implementations of the same input specification without any need to emulate and instrument the entire running environment. For user-friendliness and reproducibility, the software repository provides a number of examples of PolyTracker-instrumented builds of popular open-source software projects. We also provide an analysis library and REPL written in Python that are designed to assist users with operating over TDAGs.



## CCS Concepts

• **Security and privacy** → *Software reverse engineering*; • **Software and its engineering** → **Dynamic analysis**; *Software testing and debugging*.

## Keywords

Dynamic information flow tracing, DIFT, dynamic taint analysis, DTA, universal taint analysis, file formats, data formats

**ACM Reference Format:**
Evan Sultanik, Marek Surovič, Henrik Brodin, Kelly Kaoudis, Facundo Tuesca, Carson Harmon, Lisa Overall, Joseph Sweeney, and Bradford Larsen. 2024. PolyTracker: Whole-Input Dynamic Information Flow Tracing. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '24), September 16–20, 2024, Vienna, Austria.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3650212.3685313


## 1 Introduction

Developers regularly and unintentionally deviate from format specifications, resulting in programs that differ subtly in their input handling [22]. For example, ambiguity in the JSON RFC [10] and standard [13] about how to handle dictionaries with duplicate keys has resulted in some implementations that choose the first duplicate, others that choose the last, and yet others that silently reject all duplicates. There is a rich history of exploitation of such divergent behavior [2]. When a potentially malicious program input is known but its runtime effects are not yet fully understood, dynamic information flow tracing (DIFT) enables data flow examination at the level of individual loads and stores without having to step through the program in a debugger.

Unfortunately, most DIFT tools prioritize use cases that only require tracing a limited set of data flows (*e.g.*, the path of a small known secret value) with selective target instrumentation, or require analysis to happen simultaneously with data collection. These limitations preclude most analyses that could detect input validation bugs or parser differentials. As is true with custom program instrumentation techniques more generally, selective DIFT instrumentation is primarily useful to highly specialized reverse engineers who are already deeply familiar with their target codebase.

*Authors are listed by rough commit count at time of publication.
†Current affiliation: Block, Inc. (Square)
‡Current affiliation: Institute for Defense Analyses, Center for Computing Sciences
§Current affiliation: Praetorian



*ISSTA '24, September 16–20, 2024, Vienna, Austria*



ACM ISBN 979-8-4007-0612-7/24/09
https://doi.org/10.1145/3650212.3685313

PolyTracker is a DIFT framework for creating and analyzing granular whole-program execution traces for *entire program inputs*. PolyTracker records every input byte, how it flows through the program, how it is combined with other inputs, how it affects control-flow, and where it is output. Our primary design goal is to enable engineers of a variety of backgrounds to discover and deeply understand the ground truth bridging a potentially ambiguous input format specification and its implementations. To accomplish this, we require the ability to holistically explore the flow of control and data through an entire parser, or potentially even the entire application, if it handles inputs "shotgun" style [22]. PolyTracker achieves this with minimal runtime overhead, thanks to our novel, co-designed Tainted Directed Acyclic Graph (TDAG) runtime trace format. Post-processing the TDAG permits every program output byte to be mapped back to recorded data flows and, ultimately, the input bytes that influenced them. Analysis of PolyTracker's TDAG traces has been shown useful for variability bug triage [17], automatically detecting parser bugs that can lead to differentials [6], and automated reverse engineering [16].

## 2 Background

*Universal taint analysis* [21]—simultaneously tracking all input bytes and program data flows—is a challenging technique to implement. One must trace not only taintedness of each intermediate program value, but also calculate full *provenance* for every byte in memory. Most DIFT tools can trace just a few data flows at once, and have high overhead since they perform data flow tracing and analysis simultaneously. These are perhaps best leveraged in fuzzing coverage checks, or via inclusion of a concolic solver in an expert's targeted fuzzing workflow [7, 14] — use cases that do not generally require simultaneously tracking all data flow progress and provenance relations.

Exploring full program execution over real-world input requires the ability to determine provenance of arbitrary intermediate values in memory to first *find* interesting control and data flow sections, and to discover relevant input bytes. The selective code instrumentation needed to use many DIFT tools requires deep program analysis knowledge, and frequently also custom built tooling extensions. Thus, current DIFT tooling is less commonly leveraged than dynamic tools and sanitizers that only require instrumenting the target with appropriate compiler passes.

To enable *post hoc* trace analysis and comparison, we also need to efficiently store all taint labels and the relationships between them. While PolyTracker is inspired by Angora's [7] DIFT feature, PolyTracker yields all provenance relationships for every label, where Angora does not. Tools such as DataFlowSanitizer (DFSan) [8], Taintgrind [20], and libdft [19] hold all taint labels in memory at once; for example, the current DFSan version can track at most eight data flows. DFSan originally supported tracking up to $2^{16}$ taint labels, but this was still only sufficient to track several hundred input bytes simultaneously. In 2021, it was significantly refactored for memory layout compatibility with other LLVM sanitizers [9]; this reduced its maximum label count to eight. This is acceptable for targeted data flow analysis during fuzzing input generation, but precludes universal taint analysis.

PANDA [12] enables the user to track *whole-system* data flow, yielding more information than our use case requires. PANDA extends QEMU [3], which adds base runtime overhead of about 15% [18]. PANDA's authors report its "record and replay" (its most similar component) adds another 4x slowdown. DECAF++ [11], another QEMU-based tool, reduces the QEMU slowdown *and* decouples data flow tracking from analysis, but lacks a documented trace format and analysis library for granular trace comparison.

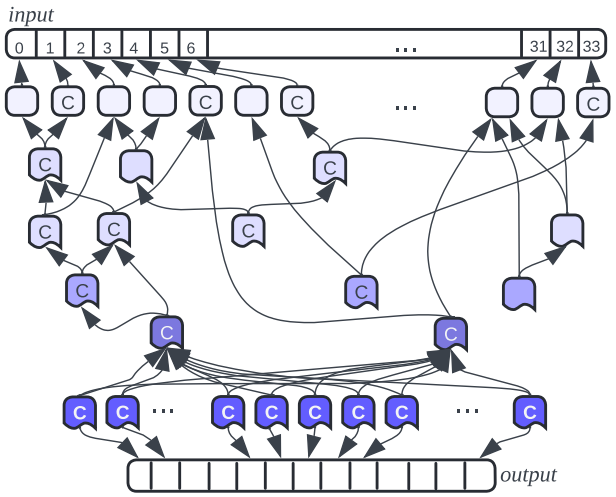


**Figure 1: A partial example TDAG program trace.**

## 3 Design

PolyTracker enables engineers to reason about the runtime progression of every input byte through code within a userland program trace, and across such traces. We designed the *tainted directed acyclic graph* (TDAG) format to efficiently store and represent the abundance of labels and relationships generated.

### 3.1 The Tainted Directed Acyclic Graph (TDAG)

The TDAG data structure is a directed acyclic graph of each taint label recorded at execution time and the provenance relationships between them. The TDAG is comprised of distinct node types (Figure 1). The increase in color saturation from graph top to bottom represents accumulating taint. *Source* nodes (the first level of labels) each reference an input byte offset. Every other label type has two ancestor references: the nodes whose taints were combined during execution. A *union node* represents a combination of two non-memory-adjacent labels. A *range node* represents a combination of two or more memory-adjacent taint labels, such as the bytes of an array. *Sink nodes* comprise the final label level before bytes are output. All nodes can be tagged as having affected control flow, depicted in the figure with '`C`'. We also record the call stack context where each control-flow-affecting node occurs. The full graph of relationships and flows between any source and sink can be reconstructed from the TDAG after program execution.

### 3.2 File Format

Our trace file format stores key aspects of the TDAG in a packed, abstracted fashion. Each TDAG file includes the following sections: a header with the magic bytes '`TDAG`'; the main labels section; sources; sinks; a source index; a compilation-mangled function name record; and a nested control flow log. We show the file representation for the

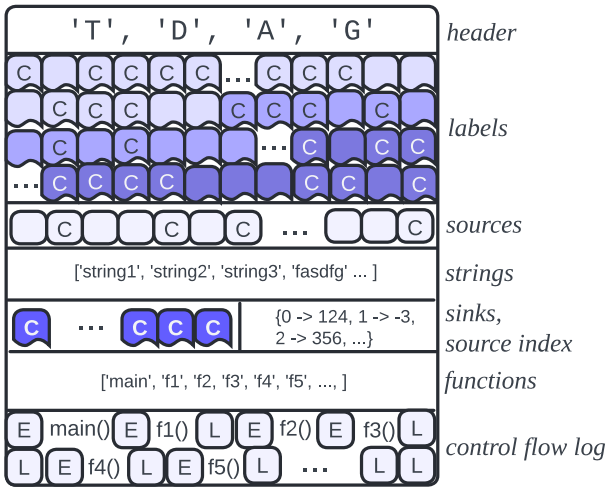


**Figure 2: Our example trace written in the TDAG file format.**

example TDAG from Figure 1 in Figure 2. At runtime, as the instrumented binary executes, PolyTracker records from shadow memory to the TDAG file's labels section. After instrumented-program exit, PolyTracker writes all other sections to the file. While most sections consist of records written without separating bytes, the control flow log differs. This section nests function call records similarly to stack frames. For every function recorded in the functions section that control flow entered, we include enter (`E`) and leave (`L`) markers so that later analyses can reconstruct the call stack.

## 4 Implementation and Usage

To enable comparative TDAG analysis, we split the DIFT tasks of recording and analysis into PolyTracker's *write* and *read* sides.

### 4.1 Write Side

PolyTracker's *write side*, shown in Figure 3, instruments software and creates TDAGs. We provide a base Dockerfile in the PolyTracker repository for ease of use. In step (1) of Figure 3, the user obtains a target codebase and builds it in a container extending our Dockerfile. PolyTracker leverages Blight [25] to record GLLVM [24] building the desired target. GLLVM is an LLVM derivative that creates whole-program bitcodes. Blight wraps build tools to record each build step. Then, in step (2) `polytracker instrument-targets` runs a series of LLVM passes to: (a) extract the whole-program bitcode from the binary; (b) place control flow logging instrumentation; (c) optimize the bitcode with `-O3`; (d) instrument the optimized bitcode for data flow logging; and (e) lower the instrumented bitcode to an executable. Steps 2a–2e are each available as `polytracker` subcommands for debugging. GLLVM includes all referenced symbols, including from dependencies that can be statically linked, in the whole-program bitcode. Direct dependencies that the user does not want to instrument need not be statically linked and can be ignorelisted instead. Syscalls, which leave userspace bounds, are out of scope. We also produce an un-instrumented binary so users can test our instrumentation is transparent. In step (3), our instrumentation placed during the data flow logging LLVM pass caches intermediate runtime data flow to shadow memory, and updates the TDAG labels section whenever a new combination of labels and

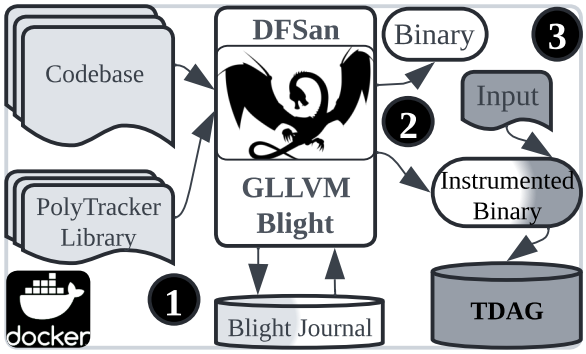


**Figure 3: The write-side implementation of PolyTracker.**

their relationships is produced. Shadow memory is a small section of process memory that we reserve to cache intermediate label values during program tracing. Through provenance relations, each label is essentially a pointer to the LLVM bitcode level propagation of all its ancestor labels.

While we originally based our data flow label recording on pre-refactor[1] DFSan [8], the label recording PolyTracker component significantly diverged from DFSan as we developed the TDAG format. Our instrumentation caches labels in shadow memory as unsigned 32-bit integers. TDAG label section entries have a corresponding fixed size so that later analyses can look up any label by offset. The label resulting from any runtime taint range or union operation is written to shadow memory for use in any further taint propagation after we write the label's ancestors to the TDAG file. All other data recorded originates from instrumentation placed via our other LLVM passes, and is written to the TDAG once program execution concludes.

### 4.2 Read Side

Reading a TDAG file requires identifying the boundaries between and decoding elements from its sections. We provide example utilities as `polytracker` subcommands using our read side library: `info`, `mapping`, `cavities`, and `forest`. The entire TDAG "forest" can be loaded into memory as several `polytracker` commands do, or the library user can work with just a particular label or set of provenance relations of interest. Label lookup followed by tracing the relationships that pertain to that label are two possible operations using the Python read side library, as are basic trace diffing and working with TDAG sections like the control flow log.

## 5 Evaluation and Use Cases

### 5.1 Current Use Cases

Any LLVM-compatible program can in theory be explored via analyzing TDAG tainted data and control flow traces, though to date we have targeted C and C++ software. Here, we highlight current PolyTracker uses for detecting bugs and security vulnerabilities.

*5.1.1 The Format Analysis Workbench and Grammar Inference.* The Format Analysis Workbench (FAW) [15] enables users to compare

[1]We consider the current DFSan to be a completely different LLVM-based tool from pre-refactor DFSan and from PolyTracker, since its design goals and implementation substantially differ.

the behavior of parsers of the same file format. Each integrated parser is instrumented with PolyTracker for parser *grammar inference*. Grammar inference [16] can assist in highlighting features that cause differences in parser output, and enables producing test inputs that result in output differentials. Such differentials are useful for finding vulnerabilities and bugs.

*5.1.2 PDF Parsers and Blind Spots.* We leveraged TDAGs to explore the internals of popular PDF parsers in order to find and compare their *blind spots*. A blind spot is comprised of one or more bytes of program input that can be arbitrarily mutated without affecting program output [6]. Blind spots are almost always indicative of an implementation bug, an error in a file format specification, or both. Blind spots can be leveraged to create polyglots (files that combine more than one format, often maliciously). Our blind spots trace analysis code is also currently integrated into the FAW. Several Dockerfiles in `examples/` can be leveraged to reproduce this work.

*5.1.3 The Acropalypse.* We leveraged PolyTracker's blind spot detection to demonstrate the usefulness of universal DIFT in parser *trace* differentials. We instrumented a version of the image parser libpng with PolyTracker, then processed images that were cropped using software vulnerable to CVE-2023-21036 [1] with our instrumented binary [5]. Diffing the resulting TDAGs using PolyTracker's blind spot detection (provenance relationship tracing) functionality showed that the "cropped" image ended in a blind spot at the point it had been cropped, revealing the bug resulting in the CVE. `Dockerfile-acropalypse.demo` can be used to reproduce this work.

*5.1.4 NITF Parser Build Differentials.* The National Imagery Transmission Format (NITF) is an imagery packaging file format with a reference implementation called Nitro [23]. Instrumenting Nitro with PolyTracker, we discovered Nitro suffers from variability bugs: errors that seem to disappear or change when the program is compiled for debugging. We developed a technique to automatically compare patterns in *control-affecting data flow* between Nitro variant builds. Control-affecting data flow is a program representation derived from the TDAG. Our differential analysis exposed bugs and undefined behavior in Nitro [17], such as the result of omitting a specification defined field value from the implementation `enum` [4]. `analysis/ubet/Dockerfile.nitro` and scripts in `analysis/ubet` can be used to reproduce this work.

## 5.2 Evaluation

Our setup used the base Docker container from `master a90f9dd`. Our build containers ran on a Debian Bullseye VM with 16 GiB RAM, 8 AMD vCPUs, 320 GiB disk. In Table 1, we show Docker-measured seconds to instrument each whole-program bitcode versus approx. (`ls -lb`) size. While one can instrument multiple targets from the same codebase at once with PolyTracker, we separately instrumented e.g., poppler `pdftops` and `pdftotext` for clearer measurement. We used the same Blight journal for targets from the same codebase. Table 2 shows tracing runtime and memory overhead for happy path test inputs averaged over ten runs. We used the shell builtin `time` to obtain userland execution times and separately used GNU `memusage` for peak heap and stack usage. Table 3 follows the same methodology as Table 2, for inputs that exercise parsing and security corner cases. Input and software complexity contribute to data flow tracing overhead. Averaged over corner cases, tracing increased peak heap usage by 18.6% and peak stack usage by 51.27%. Averaged across all software test runs, tracing increased peak heap usage by 21% and peak stack usage by 55.3%. Inputs are available on request.

**Table 1: Instrumentation time vs journal and bitcode size.**

| Codebase | Journal | In .bc | Out .bc | Inst. time |
|---|---|---|---|---|
| libpng | 0.09 MiB | 1.65 MiB | 4 MiB | 30.8 sec |
| daedalus pdf | 15.88 MiB | 4.51 MiB | 16.53 MiB | 36.5 sec |
| file | 0.10 MiB | 0.99 MiB | 2.02 MiB | 12.5 sec |
| ffmpeg | 1.74 MiB | 33.54 MiB | 81.89 MiB | 159.9 sec |
| libjpeg | 0.38 MiB | 1.53 MiB | 3.65 MiB | 26.7 sec |
| mupdf | 2.38 MiB | 21.43 MiB | 83.98 MiB | 193.5 sec |
| nitro | 1.73 MiB | 4.63 MiB | 16.30 MiB | 29.3 sec |
| openjpeg | 0.25 MiB | 0.88 MiB | 3.70 MiB | 53.1 sec |
| poppler pdftops | 0.89 MiB | 9.92 MiB | 36.56 MiB | 288.4 sec |
| poppler pdftotext | 0.89 MiB | 8.98 MiB | 32.60 MiB | 250.9 sec |
| qpdf | 3.04 MiB | 13.60 MiB | 49.10 MiB | 513.9 sec |
| xpdf pdfinfo | 0.68 MiB | 4.41 MiB | 18.16 MiB | 163.5 sec |
| xpdf pdftops | 0.68 MiB | 5.66 MiB | 24.34 MiB | 231.1 sec |
| xpdf pdftotext | 0.68 MiB | 4.68 MiB | 19.79 MiB | 175.4 sec |

**Table 2: Userland runtime and peak memory overhead on happy path inputs.**

| Utility | Input | Runtime | Heap | Stack |
|---|---|---|---|---|
| pngtest | fabio_color_256.png | 101x | 1.23x | 3.31x |
| parser-test | test.pdf | 3x | 1x | 25.8x |
| file | fabio_color_256.png | 150x | 1x | 1x |
| ffmpeg | 1MB-MP4.mp4 | 98x | 3.17x | 7x |
| djpeg | testimg.jpg | 80x | 2.62x | 1.53x |
| mutool info | test.pdf | 2x | 1.13x | 25.43x |
| show_nitf++ | U_0001A.NTF | 20x | 13.16x | 52.5x |
| opj_decompress | sample1.jp2 | 82.65x | 1.78x | 8.8x |
| poppler pdftops | test.pdf | 30x | 1x | 2.14x |
| poppler pdftotext | test.pdf | 2x | 1x | 2.14x |
| qpdf | test.pdf | 30x | 1.1x | 1x |
| xpdf pdfinfo | test.pdf | 10x | 1x | 3.1x |
| xpdf pdftops | test.pdf | 3x | 1x | 1x |
| xpdf pdftotext | test.pdf | 15x | 1x | 3.48x |

**Table 3: Userland runtime and peak memory overhead on corner case inputs.**

| Utility | Input | Runtime | Heap | Stack |
|---|---|---|---|---|
| pngtest | re3eot.png | 149.38x | 1.21x | 3.31x |
| parser-test | pocorgtfo07.pdf | 10x | 1x | 23.8x |
| file | no-magic-nor-end.pdf | 150x | 1x | 1x |
| ffmpeg | AVI+ZIP.avi | 4x | 5.15x | 30.68x |
| djpeg | cve-2023-2804.jpg | 80x | 3.89x | 2.63x |
| mutool info | pocorgtfo07.pdf | 14x | 1.13x | 46.06x |
| show_nitf++ | U_2001E.NTF | 20x | - | - |
| opj_decompress | 2k_wild_lossless.jp2 | 15.63x | 1x | 8.8x |
| poppler pdftops | pocorgtfo07.pdf | 36.79x | 1x | 1.1x |
| poppler pdftotext | pocorgtfo07.pdf | 44.85x | 1x | 1.7x |
| qpdf | pocorgtfo07.pdf | 210.28x | .03x | 1.06x |
| xpdf pdfinfo | pocorgtfo07.pdf | 18.57x | 1x | 1.58x |
| xpdf pdftops | pocorgtfo07.pdf | 42.67x | 1.26x | 1x |
| xpdf pdftotext | pocorgtfo07.pdf | 40.68x | 1.1x | 2.2x |

## 6 Tool Availability

PolyTracker is publicly available from https://github.com/trailofbits/polytracker. We also provide a Python API manual at https://bit.ly/3A5VIBU. Finally, we link a (longer, as presented to Purdue's CERIAS Security Seminar) video demonstration of PolyTracker's capabilities https://youtu.be/RuA0eerGqEE, and a shorter walkthrough video https://youtu.be/O4JtycJPN3k.

## Acknowledgments

This research was supported in part by the DARPA SafeDocs program as a subcontractor to Galois under HR0011-19-C-0073.

## References


[1] Simon Aarons and David Buchanan. 2023. *CVE-2023-21036*. Available from MITRE and the NIST National Vulnerability Database. https://nvd.nist.gov/vuln/detail/CVE-2023-21036.

[2] Ange Albertini. 2015. Abusing file formats; or, Corkami, the Novella. *The International Journal of Proof of Concept or GTFO* 0x07, 6 (March 2015), 18–41.

[3] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual USENIX Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[4] Henrik Brodin. 2023. *#528: Missing support for YCbCr601*. https://github.com/mdaus/nitro/issues/528.

[5] Henrik Brodin. 2023. *How to avoid the aCropalypse*. https://blog.trailofbits.com/2023/03/30/acropalypse-polytracker-blind-spots/.

[6] Henrik Brodin, Evan A. Sultanik, and Marek Surovič. 2023. Blind Spots: Identifying Exploitable Program Inputs. In *Proceedings of the Eighth Workshop on Language-Theoretic Security at IEEE Security and Privacy*.

[7] Peng Chen and Hao Chen. 2018. Angora: Efficient Fuzzing by Principled Search. In *Proceedings of the IEEE Symposium on Security and Privacy*. 711–725. https://doi.org/10.1109/SP.2018.00046

[8] LLVM Contributors. 2020. DataFlowSanitizer. https://clang.llvm.org/docs/DataFlowSanitizer.html. Accessed: 2020-07-26.

[9] LLVM Contributors. 2021. [DFSan] Change shadow and origin memory layouts to match MSan. LLVM commit `45f6d5522f8d`, https://reviews.llvm.org/D104896?id=354633. Accessed: 2020-07-26.

[10] Douglas Crockford. 2006. The application/json Media Type for JavaScript Object Notation (JSON). RFC 4627. https://doi.org/10.17487/RFC4627

[11] Ali Davanian, Zhenxiao Qi, Yu Qu, and Heng Yin. 2019. DECAF++: Elastic Whole-System Dynamic Taint Analysis. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*. USENIX Association, Chaoyang District, Beijing, 31–45.

[12] Brendan Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. 2015. Repeatable reverse engineering with PANDA. In *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*. 1–11.

[13] Standard ECMA-404. 2017. *The JSON Data Interchange Format*. ECMA International. https://ecma-international.org/publications-and-standards/standards/ecma-404/.

[14] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. afl++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*.

[15] Galois. 2023. *Format Analysis Workbench (FAW)*. https://github.com/galoisinc/faw.

[16] Carson Harmon, Bradford Larsen, and Evan A. Sultanik. 2020. Toward Automated Grammar Extraction via Semantic Labeling of Parser Implementations. In *Proceedings of the Sixth Workshop on Language-Theoretic Security at IEEE Security and Privacy*.

[17] Kelly Kaoudis, Henrik Brodin, and Evan A. Sultanik. 2023. Automatically Detecting Variability Bugs Through Hybrid Control and Data Flow Analysis. In *Proceedings of the Eighth Workshop on Language-Theoretic Security at IEEE Security and Privacy*.

[18] Ahmed Karaman. 2020. Measuring QEMU Emulation Efficiency. https://ahmedkrmn.github.io/TCG-Continuous-Benchmarking/Measuring-QEMU-Emulation-Efficiency/. Accessed: 2020-07-26.

[19] Vasileios P Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D Keromytis. 2012. libdft: Practical dynamic data flow tracking for commodity systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 121–132.

[20] W. M. Khoo. 2020. Taintgrind: a Valgrind taint analysis tool. https://github.com/wmkhoo/taintgrind/releases/tag/v3.14.0.

[21] Junhyoung Kim, TaeGuen Kim, and Eul Gyu Im. 2014. Survey of dynamic taint analysis. In *2014 4th IEEE International Conference on Network Infrastructure and Digital Content*. 269–272. https://doi.org/10.1109/ICNIDC.2014.7000307

[22] Falcon Momot, Sergey Bratus, Sven M Hallberg, and Meredith L Patterson. 2016. The seven turrets of babel: A taxonomy of langsec errors and how to expunge them. In *2016 IEEE Cybersecurity Development (SecDev)*. IEEE, 45–52.

[23] J. Daniel Smith. 2022. *NITRO*. Maxar. NITRO (NITFio, "R" is a ligature for "Fi") is a full-fledged, extensible library solution for reading and writing the National Imagery Transmission Format (NITF), a U.S. DoD standard format. Available from https://github.com/mdaus/nitro/releases/tag/NITRO-2.11.2,.

[24] SRI International. 2022. *WLLVM: Whole Program LLVM in Go*. https://github.com/SRI-CSL/gllvm/releases/tag/v1.3.1.

[25] William Woodruff. 2023. *Blight: A framework for instrumenting build tools*. Trail of Bits. https://github.com/trailofbits/blight/releases/tag/v0.0.53.


Received 2024-07-05; accepted 2024-07-26