covdiff - A Code-Coverage Difference Snapshot Analysis & Visualization Tool

Adam Niezabitowski

December 15, 2025

1 Introduction

1.1 Summary

This report presents *covdiff*, a tool for extraction, analysis, and visualization of temporal coverage snapshots. The primary aim of the tool is to aid security researchers in the development of Fuzzing strategies, and to allow for fast triage of high-impact execution paths.

1.2 Motivations

Fuzzing is a process in which code is bombarded with random inputs derived from mutated reference samples, in the hope of finding an input that causes a bug, which may then lead to the discovery of a vulnerability [1]. Fuzzing is coverage guided, i.e. inputs which unlock new code paths are propagated with the hope of finding further paths.

It has been shown that an increase in coverage alone is a sufficient metric to measure fuzzer effectiveness [2]. However, how much new coverage is discovered is determined by the fuzzer setup (such as custom mutators, dictionaries, etc). Hence the challenge in modern fuzzing is not determining if the current setup is effective; it is in designing a setup such that the amount of new coverage discovered is maximized.

This report proposes a means of addressing this challenge by allowing the researcher to visualize the difference of two coverage snapshots produced by the fuzzer. Through this, they can answer some critical questions:

1. What caused the increased in coverage?
2. How did my current setup allow for this coverage to be discovered?
3. How can I adapt my setup to ensure that other, similar paths may be discovered?

2 Tools & Technologies Used

covdiff is divided into two distinct parts - data analysis, which involves processing the raw data from the coverage generator and binaries - and visualization, which involved displaying the analyzed data in an intuitive, useful way.

For data analysis, given the requirement for rapid prototyping and quick iteration, Python was primarily used. Java was also used for extraction of binary data, as this is the primary scripting language for the binary analysis engine used (Ghidra [3]).

For data storage, SQLite was used. Due to the high volume of data (tens of thousands of basic-blocks, edges, etc), this allowed for a high-performance manner in which to store and manipulate the data. After analysis, the data is exported into JSON format for ease of use in the visualization components.

Finally, for visualization, Javascript was used with the React framework [4]due to its rich visual framework support. Third party libraries used consist of *d3.js* [5] and *Cytoscape.js* [6] for visualization, and *flexlayout-react* [7] for the UI layout. *Electron* [8] was also used to build an optional native application to wrap the web-view.

3 Dataset & Analysis

There are two primary sources of data:

1. The coverage files themselves - containing the edges and basic-blocks that were covered at the first snapshot (coverage A) and the second snapshot (coverage B). Note that A is a subset of B. The current version of *covdiff* only supports coverage generated by *TinyInst* [9], with in edge coverage mode.

2. The binary files for which the coverage was collected for.

3.1 Data Extraction

Data is extracted from both the raw coverage files and the binaries.

For the coverage files, files A and B are passed into coverage_parser.py, which parses the textual format and outputs a SQLite database containing the basic blocks and edges associated with each coverage.

For the binary files, *Ghidra* is used to run analysis on each of the binaries. Then, it is invoked in headless mode with the *ExtractCFGToSQLite.java* script to extract & relate function, basic-block, and edge information. This is then stored a separate, master SQLite database. This database is used for containing binary information for all modules, and can span multiple analyses of different fuzzing campaigns.

3.2 Data Analysis

Analysis is then performed on the extracted datasets via *coverage_ analyzer.py*. The analysis identifies newly covered code blocks between snapshots A and B and attributes them to specific execution points.

Coverage Processing: The program joins coverage data with binary static analysis, mapping module IDs to binaries via SHA256 hashes and resolving return addresses to their containing basic blocks.

New Block Identification: Computes which blocks are in A, in B, or newly covered (present in B but not A).

Frontier Analysis: Identifies *frontier edges* that connect previously covered (A) blocks to new blocks, classifying them as *strong* (reachable only via A blocks) or *weak* (reachable via multiple paths) [10]. The program then computes reachability from each frontier block to determine which new blocks it can reach.

Attribution & Scoring: Attributes new coverage to frontier targets, distinguishing uniquely attributed blocks from shared ones. Finally, aggregates these results into function-level and callsite-level "unlock scores" that quantify how much new coverage each code location enables.

3.3 Data Export

Analysis results are exported to JSON format via *coverage_ exporter.py* for visualization. The script queries both databases to extract module, function, and block-level data, including coverage status, attribution scores, and frontier classifications. Data is organized hierarchically (modules → functions → blocks → edges) to enable interactive visualization of coverage differences.

4 Novelty

While there are existing tools to compare coverage on a detailed level (see *Lighthouse* [11] and *Gcov*) [12], there are no such tools for visualizing and large-scale "big picture" coverage changes, or for causality analysis.

5 Encoding Channels & Idioms

The analyzed data is then displayed in an interactive, reorganize-able dashboard. The primary features include:

- **Treemap Views** – three separate treemap views are utilized for modules, functions, and basic blocks.
- **Call Graph View** – the main view of the dashboard; the graph allows researchers to discover connections between functions and quickly distinguish important calls.
- **Detail View** – shows more detailed information for the currently selected item. It offers textual information and various charts specific to the selected item.

5.1 Treemap Channels & Idioms

5.1.1 Encoding

- **Area:** entity size (bytes/block count)
- **Color:** status (red/orange/gray), coverage gradient (gray→orange→red), or frontier gradient (blue→purple)
- **Containment:** hierarchical relationships (module→function→basic block)

5.1.2 Interaction

- Click selection for drill-down, category filtering, collapsible legend

5.2 Call Graph Channels & Idioms

5.2.1 Encoding

- **Node size:** new basic block count
- **Node color:** coverage percentage or frontier count (dual-mode gradients)
- **Node shape:** circle (direct calls) vs. diamond (indirect calls)
- **Border style:** solid/dashed for frontier strength
- **Edge style:** solid/dashed, black/grey for direct vs transitive edges
- **Layout:** force-directed positioning

5.2.2 Interaction

- Distance-based edge filtering, minimum size/BB filters, show/hide unchanged functions

5.3 Detail View Channels & Idioms

5.3.1 Encoding

- **Horizontal stacked bars:** part-to-whole composition with categorical colors
- **Color-coded text:** status and frontier type indicators
- **Two-column layout:** textual details (left), compositional charts (right)

5.3.2 Charts

- New BB composition, frontier composition, function/block coverage distributions

6 Critical Analysis

While *covdiff* certainly succeeds in aiding security researchers in answering the questions outlined in Section 1.2, there are numerous improvements that can still be made, particularly relating to the ease-of-use of the application:

- The analysis and visualization components are highly disconnected. Users have to first manually parse, analyze & export the data, and only then import it into the visualization dashboard. A better solution would be to have the entire process integrated into the dashboard component.

- There is no fuzzer integration. Users must manually extract coverage information from fuzzers.
- covdiff only supports coverage with edge information - specifically, data generated by *TinyInst*. This severely limits potential applications where different coverage engines are used.
- There is no detailed basic-block information. While basic block information is exported, due to performance and library support limitations, this is not visualized (e.g. internal function edges).

Nevertheless, *covdiff* hopes to be a valuable tool for the security research community. At the time of writing, you may find the code & working demo on my GitHub. See Appendix A for a screenshot of the dashboard.

References

[1] Barton P. Miller, Lars Fredriksen, and Bryan So. "An empirical study of the reliability of UNIX utilities". In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279. URL: https://doi.org/10.1145/96267.96279.

[2] Marcel Böhme and Brandon Falk. "Fuzzing: on the exponential cost of vulnerability discovery". In: *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2020. Virtual Event, USA: Association for Computing Machinery, 2020, pp. 713–724. ISBN: 9781450370431. DOI: 10.1145/3368089.3409729. URL: https://doi.org/10.1145/3368089.3409729.

[3] National Security Agency. *Ghidra*. 2025. URL: https://github.com/NationalSecurityAgency/ghidra.

[4] Facebook. *React*. 2025. URL: https://github.com/facebook/react.

[5] D3 contributors. *D3.js*. 2025. URL: https://github.com/d3/d3.

[6] The Cytoscape Consortium. *Cytoscape.js*. 2025. URL: https://github.com/cytoscape/cytoscape.js.

[7] caplin. *FlexLayout (flexlayout-react)*. 2015. URL: https://github.com/caplin/FlexLayout.

[8] Electron contributors. *Electron*. 2025. URL: https://github.com/electron/electron.

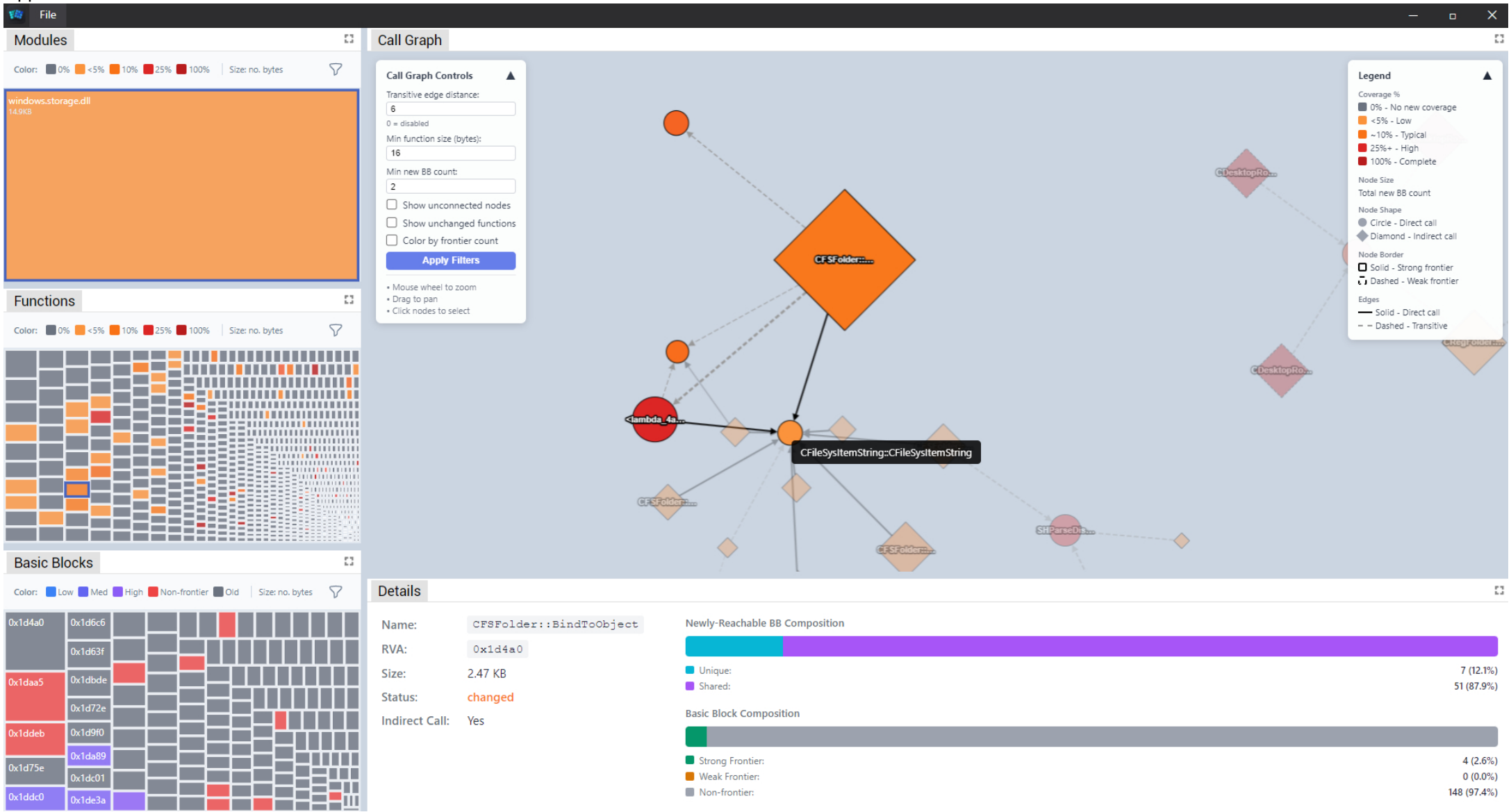
[9] Google Project Zero. *TinyInst*. 2020. URL: https://github.com/googleprojectzero/TinyInst.

[10] Stefan Nagy and Matthew Hicks. "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing". In: *2019 IEEE Symposium on Security and Privacy (SP)*. 2019, pp. 787–802. DOI: 10.1109/SP.2019.00069.

[11] gaasedelen. *Lighthouse*. 2017. URL: https://github.com/gaasedelen/lighthouse.

[12] GNU Project. *gcov*. 2021. URL: https://gcc.gnu.org/onlinedocs/gcc/Gcov.html.

Appendix A


File
Modules
Color: 0% <5% 10% 25% 100%
Size: no. bytes
windows.storage.dll
14.9KB
Functions
Color: 0% <5% 10% 25% 100%
Size: no. bytes
Basic Blocks
Color: Low Med High Non-frontier Old
Size: no. bytes
0x1d4a0
0x1d6c6
0x1d63f
0x1daa5
0x1dbde
0x1d72e
0x1ddeb
0x1d9f0
0x1da89
0x1d75e
0x1dc01
0x1ddc0
0x1de3a
Call Graph
Call Graph Controls
Transitive edge distance:
6
0 = disabled
Min function size (bytes):
16
Min new BB count:
2
Show unconnected nodes
Show unchanged functions
Color by frontier count
Apply Filters
• Mouse wheel to zoom
• Drag to pan
• Click nodes to select
Legend
Coverage %
0% - No new coverage
<5% - Low
~10% - Typical
25%+ - High
100% - Complete
Node Size
Total new BB count
Node Shape
Circle - Direct call
Diamond - Indirect call
Node Border
Solid - Strong frontier
Dashed - Weak frontier
Edges
Solid - Direct call
Dashed - Transitive
CFSFolder::...
<lambda_4a...
CFileSysItemString::CFileSysItemString
CFSFolder::...
CFSFolder::...
SHParseDis...
CDesktopRo...
CDesktopRo...
Details
Name: CFSFolder::BindToObject
RVA: 0x1d4a0
Size: 2.47 KB
Status: changed
Indirect Call: Yes
Newly-Reachable BB Composition
Unique: 7 (12.1%)
Shared: 51 (87.9%)
Basic Block Composition
Strong Frontier: 4 (2.6%)
Weak Frontier: 0 (0.0%)
Non-frontier: 148 (97.4%)