Towards Path-Aware Coverage-Guided Fuzzing

Giacomo Priamo*, Daniele Cono D'Elia*, Mathias Payer†, and Leonardo Querzoni*
*Sapienza University of Rome, Rome, Italy
†EPFL, Lausanne, Switzerland
{g.priamo, delia, querzoni}@diag.uniroma1.it, mathias.payer@nebelwelt.net

***Abstract*—Automated fuzz testing is now standard practice, yet key blind spots persist. Coverage-guided fuzzers typically rely on edge coverage as a lightweight proxy for program behavior. However, this metric captures path variations only weakly: it cannot differentiate executions that follow distinct control-flow paths but traverse the same edges—causing many path-dependent bugs to go undetected. Path awareness would offer a richer coverage view but has been considered too costly for fuzzing.**

We introduce a lightweight method for tracking intra-procedural execution paths, enabling efficient path-aware feedback. This enhances the fuzzer's ability to detect subtle bugs, even in well-tested software. To counter the resulting seed explosion, we evaluate two strategies—*culling* and *opportunistic* path-aware fuzzing—that balance precision and throughput. Our findings show that path-aware fuzzing, when properly guided, uncovers more bugs and reveals untapped potential in fuzzing research.

***Index Terms*—Fuzzing, Coverage feedback, Path profiling**

I. Introduction

Fuzzing has surged in popularity due to its effectiveness in discovering bugs in software [1]. Generating random inputs for a program under test is a simple, yet surprisingly effective way to trigger crashes and expose bugs. Coverage-guided fuzzing refines this approach by incorporating a feedback mechanism to guide input generation. Rather than starting from scratch for each input, it mutates previously *interesting* inputs—those that trigger novel execution behaviors [1], [2]. This feedback-driven strategy surpasses purely random input generation by focusing mutations on test cases that cover new behaviors.

The most widely adopted feedback metric is *edge coverage*, which records the control-flow edges exercised during execution. While edge coverage is both efficient and effective [2], it captures control-flow paths in a lossy form. Notably, it cannot distinguish between test cases that traverse the same edges but along different paths [3]. This limitation hinders the discovery of bugs that depend on specific path-related conditions [4].

To overcome this limitation, one could consider not just which edges are taken but also the *order* in which they are traversed, thereby distinguishing complete execution paths. Tracking statement ordering is known as *flow-sensitivity* in program analysis [5] or path-sensitivity in testing [6] (without considering path conditions). Here, we prefer the term *path-aware fuzzing* to exclusively refer to the path exploration.

Literature considered path-aware fuzzing impractical due to the anticipated high costs for instrumentation, storage, and runtime [6], [7]; only PathAFL [7] tried tracking partial whole-program paths, using aggressive pruning and partial instrumentation. Modern fuzzers thus rely on the coarse, inaccurate [3] approximation from edge coverage to deal with paths.

Our approach. This paper shows that it is feasible to distinguish and track execution paths with manageable overhead. Our key insight is that instead of tracking whole-program paths [8], which leads to path explosion [7], we focus on *intra-procedural path profiles* [3] (i.e., paths within individual functions). By adapting an efficient path encoding [3], we track per-function paths with costs almost comparable to edge coverage. A fuzzer can use this mechanism to estimate behavioral differences between executions based on function-level path coverage. When we replace edge coverage with our path-aware feedback, we discover numerous bugs missed by edge coverage-guided fuzzing—even in well-tested software.

However, this finer-grained feedback increases the number of interesting test cases that the fuzzer has to manage, resulting in *seed explosion* [9] effects (also called *queue explosion* [4]). Many of these test cases are redundant or unhelpful, slowing down the fuzzer. To explore the benefits of path-aware fuzzing while mitigating this overhead, we investigate two strategies:

(a) **Culling**: This method periodically culls the queue of interesting test cases while preserving total edge coverage. By trimming redundant test cases, it enables the fuzzer to explore new execution paths more effectively, improving throughput and occasionally revisiting previous paths. This strategy surpasses both the path-aware and edge-guided baselines.

(b) **Opportunistic**: This method starts with a queue generated using edge coverage as feedback and switches to the path-aware feedback for the remainder of the fuzzing time. This hybrid strategy benefits from the effectiveness of edge-based exploration and the sensitivity of path-aware fuzzing, revealing bugs that neither the two baselines nor *culling* uncover.

The evaluation of our AFL++-based implementation on the UniFuzz suite [10] shows that the increased visibility of path-aware fuzzing uncovers a large number of bugs missed with edge coverage. Our best configuration finds 10.1% more bugs overall, and 27.5% that are unique to our approach. We responsibly disclosed 11 zero-day vulnerabilities, 5 of which were missed by traditional fuzzers. These findings indicate that path-aware sensitivity, when guided by *exploration biasing* methods, can open new directions for fuzzing research.

This paper proposes the following contributions:

- A practical, fuzzer-friendly instrumentation technique for recognizing intra-procedural execution paths, making path-aware fuzzing viable.
- Two exploration biasing strategies, *culling* and *opportunistic*, to mitigate seed explosion and broaden the effectiveness of path-aware fuzzing.

- An open-source LLVM-based implementation of our techniques: https://github.com/Sap4Sec/path-aware-fuzzing.
- An extensive evaluation on the UNIFUZZ benchmarks with AFL++, showing our methods reveal a broader set of bugs than both edge coverage-guided fuzzing and PathAFL, and improve internal performance metrics.

II. Background and Motivation

This section covers the fundamental concepts of coverage-guided fuzzing and draws attention to a connected limitation.

A. Coverage-guided Fuzzing

Fuzzing techniques are popular in software testing and security research thanks to their effectiveness in discovering bugs [1]. The most basic implementation of a fuzzer is a system that generates random inputs and feeds them as test cases to the program under test during repeated executions.

Grey-box fuzzers [2], [11] enhance this baseline by employing lightweight instrumentation to track coarse-grained information about program execution. These fuzzers are the most prevalent choice among all fuzzer types due to their effectiveness. Well-known grey-box fuzzers like AFL [2] track code coverage and use it as a feedback mechanism to drive the exploration of the program under test. More specifically, they keep track of what control flow edges are taken during the execution of the test cases (*edge coverage*). This strategy significantly improves a fuzzer's bug finding abilities [12].

Coverage-guided fuzzers instrument the code of the program under test to collect coverage-related facts and update a *coverage map* that profiles test case execution. With edge coverage, a map entry describes one or more control flow edges. This information drives the subsequent mutation work of previous inputs, retaining in the fuzzer's *queue* those test cases that brought about coverage novelty, with the objective of generating new inputs that either reach unexplored code regions, or make the program crash. A common refinement keeps track of how many times (*hit count*) a test case exercises a coverage element, with a normalization step (power-of-two buckets) to limit the number of inputs retained in the queue.

B. The Limitation of Edge Coverage

Despite the effectiveness in detecting bugs it entails, the use of edge coverage presents one limitation in capturing program control flows, which relates to its coarse-grained summary of execution. Specifically, it only provides a loose [3] approximation of the control-flow paths that program execution traverses.

Edge coverage may miss relevant paths able to set the specific internal conditions required to trigger certain bugs [4], [7] or to unlock further code regions. This would occur because traversing the same edges from different paths is not considered a novelty by an edge coverage-based fuzzer.

Motivating example: We use the code in Figure 1 as a running example to show how a path-aware coverage feedback can assist a fuzzer in effectively exposing a bug that manifests only when certain path-dependent conditions are met.

Function `foo` contains a heap-overflow bug on line 19 that can trigger only when the execution passes through line 9

```
 1  // arr is an heap array of size N=54
 2  void foo(char *input, int *arr) {
 3      int j;
 4
 5      unsigned len = strlen(input);
 6      if (len-2 > N || len < 3) return;
 7
 8      if (len % 4 == 0 && len > 39) {
 9          j = 3; // rare to reach
10      } else {
11          j = -2;
12      }
13
14      char c = input[0];
15
16      if (c == 'h') {
17          // buffer overflow if reached
18          // via `rare' block and len > 50
19          arr[len+j] = 7;
20      } else {
21          j = abs(j);
22          arr[j] = 0;
23      }
24  }
```

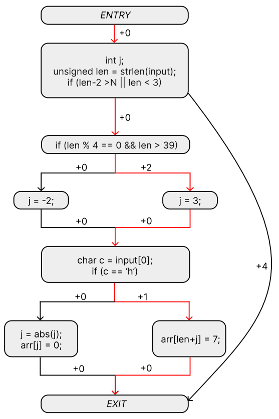

Fig. 1. **Left**: Motivating example for a path-aware fuzzer. **Right**: Control flow graph for `foo`, augmented with machinery for path identification. Edges with non-zero increments are instrumented to compute the path ID (initially set to 0). Traversing the path in red is necessary for triggering the bug.

and the size of the string `input` is greater than 50. To do that, the program state must satisfy two conditions: (i) the fuzzer-controlled input string `input` should be longer than 50 characters and start with an '`h`'; and (ii) execution must pass through line 9 to set the value for variable `j` such that it can overflow the bounds of the array `arr` on line 19.

While condition (i) can be met by byte mutations or branch solving techniques, condition (ii) requires a precise program state configuration to be met, which is set by execution flowing from line 9 to line 19 by taking the edges for lines 8→9 and 16→19. A classic fuzzer instead would most likely first reach line 19 from line 11 (10→11, 16→19) due to the hard condition to be met on line 8. Then, if it reaches line 21 from line 9 (8→9, 20→21), and only at a later point it generates a test case that reaches line 19 from line 9 as required for (ii), it will not retain this test case because all the individual edges composing this path were observed in previous executions[1].

A fuzzer able to discriminate control-flow paths would instead retain the test case because the path through line 9 to line 19 presents itself as a novelty to the fuzzer compared to previous executions. It then becomes only a matter of subsequent byte mutations to derive from such test case an input that satisfies also condition (i) and thus triggers the bug.

C. Towards a Finer-grained Feedback

The issue above could be tackled by refining the edge-coverage feedback for the fuzzer. Instead of considering only which edges are hit, one could also track the **order** in which they are activated. This corresponds to tracing execution paths.

The straightforward effect of adopting a more sensitive feedback is augmenting the visibility of the fuzzer over the program under test. More pervasive program testing with a path-aware feedback increases the probability of traversing and exploiting paths that set the required bug-triggering conditions.

[1]This holds unless the test case also satisfies condition (i) at the same time, therefore immediately triggering the bug; which we however deem unlikely.

These paths may be the only ones that satisfy the conditions, or they may do so more readily than others.

Fuzzing with a path-aware feedback may pave the way to discovering bugs that a classic fuzzer would struggle to find. Occasionally, it may even facilitate the fuzzer in meeting specific program conditions required for reaching new code regions. Section V-A and V-C study these effects, respectively.

To build such a fuzzer, two technical challenges should be addressed: (i) collecting path information may be expensive and (ii) more sensitive feedbacks are prone to queue explosion effects [4], [9]. For challenge (i), as a matter of fact, tracking paths has been considered infeasible for fuzzing [6], [7]. Our paper challenges this belief, showing a tenable solution.

As for challenge (ii), queue explosion effects can harm fuzzing efficiency. The increased visibility of a path-aware feedback comes at a cost: a high diversity of encountered paths can make the fuzzer maintain a queue of overly abundant test cases. A too large queue can prevent the fuzzer from effectively utilizing all these test cases for generating new inputs. With a limited time budget for testing, the fuzzer may end up stressing only a small fraction of the witnessed program states sufficiently, missing bugs tied to conditions set by those test cases that received insufficient mutation time.

III. Methodology and Design

Our core idea for building a path-aware fuzzer is that targeting *acyclic paths* within the control flow graph (CFG) of individual functions greatly reduces the number of paths to distinguish. This renders path tracking tenable, and the baseline path-aware fuzzer already proves capable of uncovering bugs the edge-based counterpart overlooks. However, the improved visibility negatively affects the general fuzzing throughput.

To tame this queue explosion setback, we vary the exploration strategy of the fuzzer with two biasing methods, seeking to obtain every time further new bugs compared to the other (edge and path-aware) configurations, thus proving that path-awareness is a promising new direction for fuzzing research.

A. Path Profiles as Feedback for Fuzzing

The order in which individual CFG edges are traversed can be traced in many ways. Previous endeavors attempted to reason on the whole program, thus collecting inter-procedural path information [7]. Yet, the enormous number of paths that a program can subsume represents a significant setback, as it makes their complete enumeration untenable [6], [7].

We therefore explore the tracking of path information in a **per-function** fashion, focusing on accurately identifying the internal states of each function and, for the sake of scalability, truncating paths that involve function calls.

To both handle the collection of this intra-procedural path feedback and discriminate any two program executions for novelty, we avail ourselves of the notion of path profiles.

In performance profiling literature, a *path profile* determines how many times each path in a routine executes [3]. A path profile subsumes the more common basic block and edge profiling information, which are cheap to collect but typically inaccurate if used as a proxy for predicting paths [3]. Path profiles proved themselves valuable in software performance measuring and tuning [13], [14] and in debugging [15].

a) A feedback for fuzzing: Path profiles represent an interesting opportunity to refine the feedback mechanism of a code coverage-guided fuzzer. A more sensitive feedback can help it explore the code of the program under test more pervasively [9]. Here, a path-aware fuzzer could more effectively explore under-tested code regions harboring bugs that manifest only when specific execution paths shape the program state as required for triggering—or do so more easily than other paths.

Nonetheless, this integration is not straightforward, as a fuzzer must be efficient in order to be effective. Therefore, the benefits of the complexity introduced when switching to a more refined feedback must outweigh its costs.

Loops represent a threat in this sense, as they give rise to a potentially unbounded number of intra-procedural paths. Accurate reasoning on cyclic paths is deemed an intractable task in programming language research when limited spatial and temporal performance overheads are allowed [13]. Thus, the literature [3], [16] resorts to the notion of acyclic paths.

Acyclic paths can be conveniently enumerated by transforming a function's CFG into a Directed Acyclic Graph (DAG) and taking special provisions for back edges (we return to this in Section IV). The space requirements for their accurate enumeration and identification are reasonable, and a path profiling tool only needs to maintain a single word-sized state for differentiating paths [3]. Although acyclic-path profiling implies a partial loss of information about the program under analysis [13], it stands as a good trade-off for efficiency.

We find this pragmatic way of collecting path information a suitable candidate for fuzzing. It allows us to identify and instrument the acyclic paths within a function in a cost-effective way during the compilation of the program under test, replacing the edge coverage feedback instrumentation.

We employ this gathered intelligence to track at run-time the paths traversed during test case execution, informing the fuzzer by issuing a coverage map update every time an acyclic path reaches its end (as opposed to when traversing each edge, if the traditional feedback is used). Such a path terminates when execution reaches the function epilogue or a loop back edge.

b) Mode of operation: Our path-aware feedback operates similarly to an edge coverage-based feedback, but with a key distinction: the fuzzer considers each *new traversed path* as a novelty, rather than only any new covered edge.

Our design instruments CFG edges with probes for identifying the intra-procedural acyclic path that execution is currently traversing. The signals from the probes are aggregated at path termination, informing the fuzzer of the identity of the traversed path by triggering a coverage map update.

Figure 1 visually shows the effects of this mechanism applied to our motivating example from the previous section (we defer to Section IV a detailed discussion of it, as how to track path profiles represents an implementation choice). We instrument CFG edges to apply predetermined increments to a path identifier (ID) integer variable, which is initially set

TABLE I
UNIFUZZ SUBJECTS STATISTICS: QUEUE ITEMS AFTER 24-HOUR FUZZING.

Benchmark	Type	Functions	Queue (edge)	Queue (path)
cflow	C	310	1 120	3 237
exiv2	C++	2 644	2 071	2 153
ffmpeg	C	37 218	2 619	14 648
flvmeta	C	907	458	566
gdk	C	435	2 969	10 203
imginfo	C	577	1 647	4 342
infotocap	C	3 007	3 538	191 297
jhead	C	40	364	1 013
jq	C	409	1 910	11 044
lame	C/C++	482	2 151	69 590
mp3gain	C	71	1 592	5 877
mp42aac	C++	2 505	3 042	4 053
mujs	C	742	4 806	7 589
nm-new	C	3 259	3 310	15 419
objdump	C	6 241	3 854	7 303
pdftotext	C/C++	4 607	4 868	10 022
sqlite3	C	1 194	10 112	21 516
tiffsplit	C	782	1 096	19 538

to zero. The sum of the increments along each acyclic path gives us a path ID unique for the CFG. In the figure, the path capable of triggering the bug is highlighted in red and sees two increment operations, which give it the ID value `3' to distinguish it among the five acyclic paths for the CFG.

The fuzzer now gains the ability to discriminate what it cannot with an edge-based feedback: if a new test case traverses a crucial path to reach a bug, it is considered novel even if all its edges were individually visited before. Subsequent mutations can now leverage this test case to trigger the bug.

B. Wielding the Feedback for Increased Efficiency

With the path-aware feedback we propose, the benefits in terms of efficacy are clear in our experiments: this **baseline** integration of a path-aware feedback already provides valuable and interesting results, identifying several bugs in well-tested software that the edge coverage-based counterpart failed to expose (Section V-A). Moreover, it already tackles challenges deemed insurmountable in previous works [6], [7].

Nevertheless, as we discussed, a fuzzer employing a more sensitive feedback risks producing an overly rich queue that hinders its overall efficiency. To give an idea about the relevance of this phenomenon, Table I reports statistics from fuzzing for 24 hours our evaluation programs from the UNIFUZZ [10] benchmarking platform. The size increase compared to the edge feedback assumes different proportions depending on the program, ranging from moderate to very high.

To address this setback, we propose and evaluate two simple methods of immediate applicability that **bias** the exploration work of a path-aware fuzzer. That is, for all code, the fuzzer always reasons about paths, but we manipulate the selection of test cases (and associated paths) it can pick for mutation.

The **culling** method periodically reduces the queue size, preserving its edge coverage. It is a reactive method that lets the path-aware fuzzer—which lacks time to fully explore its queue—revisit its exploration choices. The **opportunistic** method capitalizes on a wealth of code coverage attained by a more coarse-grained fuzzer: it looks for bugs in potentially under-explored code, working with a smaller queue than the one a fully path-aware fuzzer would have built in that time.

Figure 2 depicts the operation of the two exploration-biasing strategies and their expected impact on the fuzzer's queue size.

1) Culling: We devise a dynamic mechanism that, after suspending the fuzzer's execution at set time intervals, which we term *culling rounds*, runs a heuristic to significantly reduce the size of its queue and then resumes its execution.

The rationale is to offer the fuzzer a fresh start, allowing it to make new prioritization choices, some of which may differ from those made during previous rounds, helping it break out of local minima. With a limited temporal budget, the fuzzer would hardly have enough time to examine and sufficiently mutate those queues. Therefore, an effective prioritization may substantially boost the fuzzer's bug finding abilities.

Reducing the queue size, and thus path diversity, ultimately improves the fuzzer's bug finding performance, as the test cases in it can now receive more, and hopefully sufficient, mutation time; meanwhile, it does not prevent the fuzzer from re-considering previously-discarded paths as they reoccur.

As one possible *culling criterion*, we choose to retain a set of test cases capable of exercising all the edges encountered across prior executions. This procedure provides at least the same visibility as an edge coverage-guided fuzzer (avoiding regression in a traditional code coverage sense) and successfully keeps the dimension of the queue at bay. Additionally, as the path-aware fuzzer sees as equally novel any two paths (i.e., regardless of the traversed blocks or edges), the procedure eases the prioritization of any new code reached in the previous round. Such code will now have a fresh, higher opportunity to be scheduled, whereas many inputs (and paths) covering previously known code will have been trimmed by then.

The procedure above is program-agnostic and does not require any specific insight or analysis of the code before or during execution. These are desirable properties as they help safeguarding the efficiency of the fuzzer, granting it the freedom to decide which program parts it should focus its exploration on. A downside of the culling strategy is that the fuzzer runs the risks of doing repeated work every time the queue is reduced, as in the subsequent exploration it may "regenerate" some of the discarded paths all over again. Nevertheless, the evidence we gathered from our experiments suggests that the benefits outmatch this potential loss: the fuzzer seems capable of catching up quickly, and this optimization gives a considerable boost to the bug finding abilities of the baseline path profiling-based fuzzer, managing to **expose more bugs** than the edge-based feedback does (Section V-A).

The curious reader may wonder why we do not cull the queue to cover all paths rather than all edges. In reality, many fuzzers (especially AFL-derived ones) already maintain a *favored corpus* of test cases entailing this coverage and skip non-favored test cases from scheduling with a very high probability. Experiments we conducted using path identity as culling criterion confirmed the benefits of using edges here.

2) Opportunistic: With this strategy, we start the fuzzing campaign with the coarse-grained edge feedback to achieve

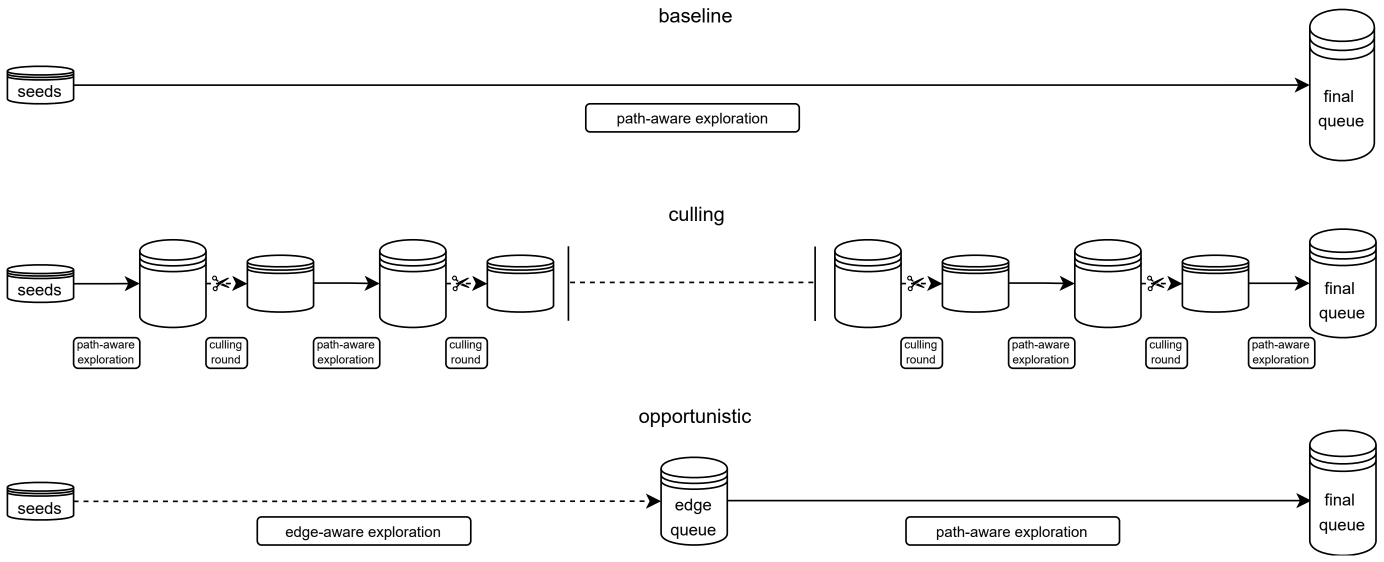

Fig. 2. Comparison of the three techniques we study for path-aware fuzzing (baseline, culling, opportunistic), with visual cues on relative queue sizes.

sufficient code coverage, and then transition to our path-aware feedback for the remaining exploration.

The rationale behind it is that while exploring program state more pervasively eases bug discovery, a more sensitive feedback reduces the fuzzer's opportunities to progressively reach new program parts within the time budget. Although a higher sensitivity may occasionally help meet difficult branch conditions, total code coverage is expected to be lower than, or at best similar to, what the edge-based feedback achieves [4].

Therefore, we propose to opportunistically rely on the work of another fuzzer to reach a larger portion of code faster, and see whether our fuzzer can discover overlooked bugs even without the wealth of alternative paths to bugs that the use of path-awareness from the start would have provided it with.

We note that a queue collected under the guidance of edge coverage represents a feasible outcome for a (very biased) path-aware exploration. As such a queue is smaller (Table I), the path-aware fuzzer is not immediately harmed by queue explosion effects when using it, and can thus build on the work of others for probing more code regions than it would normally be able to within the budget. Our method allocates a fraction of the total fuzzing budget to the work of the less sensitive feedback before switching to the path-aware one.

An insight behind this method's design is that edge coverage-guided fuzzers tend to unlock less and less new coverage over time [17] due to factors independent of the coverage feedback, such as hard-to-solve branches and incomplete harnesses. Therefore, the efficacy of a fuzzer technique can show by exposing bugs in this wealth of already reached code. From a practical perspective, the opportunity is interesting, as saturated queues from edge coverage-guided fuzzers are easily available [18]. As Section V-A will show, this flavor of path-aware fuzzer can find bugs missed by the other edge and path-aware fuzzer configurations we discussed until now.

IV. Implementation

We devise our techniques as a set of analysis and transformation passes (~1800 C++ LOC) for the intermediate representation of the LLVM compiler [19]. We work atop the state-of-the-art AFL++ [20] fuzzer (v. 4.07a), as it combines dozens of incremental research efforts and constantly delivers superb performance in community benchmarking initiatives [21]. We plug our path profiling machinery in the fuzzer's compilation process for the program under test, as an alternative option to the edge coverage feedback. This enables a seamless integration that requires no external intervention in the building phase of programs, producing a binary ready to use for fuzzing.

Ball-Larus algorithm: To construct path profiles, we rely on the efficient Ball-Larus algorithm [3], which profoundly influenced compiler and performance profiling literature [8], [22]. The algorithm offers an accurate and spatially optimal encoding, with provisions to minimize the number of instrumentation probes to place on CFG edges. We refer the interested reader to the original paper [3] for its more advanced technicalities, providing hereafter an operational description of it and how we adapt it to work synergistically with a fuzzer.

The algorithm identifies acyclic paths in a function through a numeric value: one of $\{0 \dots n-1\}$, where n is the number of acyclic paths in the function. As we anticipated, the path ID value is given by the sum of *increment values* that the algorithm places as edge labels in the CFG of a function. The run-time instrumentation adds these constant values to a per-function, word-sized state variable. A profiling client can consume this variable to identify a path once the execution completes its traversal. An acyclic path ends at the function epilogue or at a back edge: those points receive instrumentation to consume the path ID and reset the state variable to zero.

Using the Ball-Larus algorithm to profile test case execution presents interesting advantages for fuzzers. First, it can distinguish paths with minimal temporal and spatial overheads: it maintains just a word-sized state, and its operations are initializations and constant increments to such variable. Additionally, unlike edge coverage tracking that instruments and issues an event (for a fuzzer, a coverage map update) at every edge, only a fraction of the CFG edges here require instrumentation.

Integration: In practice, using path profiles as coverage feedback for a fuzzer requires: (1) running the Ball-Larus algorithm over the code of the program under test as detailed above and (2) make the fuzzer consume the produced path IDs, which translates to correctly handling coverage map updates.

We adapt a publicly available implementation of the Ball-Larus algorithm [23]. Path enumeration and increment value definition are compile-time tasks; this is ideal, as we avoid placing their burden on the fuzzer's operation. To ensure accuracy and ease performance and compatibility, we perform our analyses and add our instrumentation right after the compiler completes the execution of all optimization passes on the intermediate representation (LLVM IR) of the code. As a result, the compiler is in no way restricted from optimizing program code (or, as we will see, enabling sanitizers) in its middle-end, and its back-end can generate efficient code for our instrumentation merged within the optimized program.

Path tracking occurs at run-time by issuing a coverage map update upon encountering loop back edges and function returns, availing itself of the instrumentation introduced during the compilation phase. In this way, paths can be traced with costs almost comparable to the best-performing edge coverage tracking flavor available to date for fuzzers (`pcguard`), as we show more in detail in Section V-B and in Appendix A.

Concerning the fuzzer, we use AFL++'s configurable fixed-size bitmap, set to 2^{18} entries in the evaluation to match typical L2 cache sizes as in common practices [24]. To index the map, we combine the path ID and the identifier of the associated function as in: $(\texttt{path\_id} \oplus \texttt{function}) \,\%\, \texttt{map\_size}$.

With this modification and the above instrumentation for run-time intra-procedural acyclic path identification, we make AFL++ path-aware, without affecting its other components and the many optimizations they enable for fuzzing.

We believe that these changes, with the appropriate adaptations, may also be applied to other coverage-guided fuzzers that rely on source code instrumentation and coverage maps.

Culling: We devise the culling method through a driver that orchestrates fuzzer executions (~180 LOC). The driver takes as input a time budget, the number and duration of culling rounds, and the initial seeds. Once a culling round other than the last completes, the driver invokes a culling procedure to prune the queue, then starts a new fuzzer instance seeded with the culled queue. For fairness, the driver tracks the running time of each culling action and subtracts the accumulated time from the duration allowed for the last round; therefore, culling costs are accounted for in the fuzzing budget.

To build a minimal edge coverage-preserving queue of test cases, we rely on a fast approximation that fuzzers employ for the expensive set cover problem [25]: the favored corpus construction. In our tests, this was more efficient than using the `afl-cmin` queue minimization tool, for equivalent results.

Opportunistic: This method means to provide a queue collected with the edge coverage feedback as the starting point for the path-aware fuzzer. For fairness and efficiency, we argue that such a queue needs some (light) pre-processing.

First, to avoid biasing the results, we remove crashing inputs found by the less sensitive fuzzer to ensure that our more sensitive fuzzer relies solely on its capabilities. Second, we argue that a queue from many hours of fuzzing may be "overburdened" with many test cases sharing very similar activated edges. We thus trim this queue, seeding the path-aware stage with a smaller queue that preserves all the exercised edges. Although this may hamper the path-aware fuzzer by limiting path diversity, it serves our rigorous goal of evaluating the fuzzer's ability to discover new bugs without initial path feedback.

V. Evaluation

To evaluate the benefits of our path-aware feedback for fuzzing, we aim to address the following research questions:

RQ1: Can our path-aware feedback expose bugs that an edge coverage-based solution overlooks?

RQ2: What is the runtime cost of tracking execution paths?

RQ3: How does path-awareness affect code coverage?

The main materials from our evaluation are available as part of the artifacts we share for this paper on the linked repository.

Environment: We conduct our experiments on a server machine featuring two AMD EPYC CPUs (2.25 GHz), 1 TB RAM, and Ubuntu 20.04.5 LTS. Each fuzzer runs in a dedicated Docker container on a single CPU core. LLVM version 12 is our compiler choice for our evaluation.

Fuzzer configurations: We center our comparisons around the capabilities of edge coverage-guided fuzzing, as fuzzing evaluation best practices [26] suggest that a new fuzzing method should be compared against the existing implementation of a fuzzer that shares the highest number of similarities to the proposed one. For a fair comparison, we take the most performant embodiment available for edge coverage, `pcguard`, which is also the default configuration of AFL++.

While PathAFL is the only prior work to model paths, our approach is comparable with it only in a limited sense. PathAFL traces portions of whole-program paths [8] through partial instrumentation, aggressive pruning, and coarse-grained identifiers—all devised atop AFL [2]. It also modifies other components of the fuzzer, including seed selection, hash functions and power scheduling. Our approach traces all traversed intra-procedural acyclic paths, which embody a different path abstraction, and only replaces the fuzzer's coverage feedback. For these reasons, we believe the only ground to compare the two approaches is by looking at bug counts.

We test the following fuzzer configurations ("fuzzers"):

- **`path`**: the baseline approach using intra-procedural paths as coverage feedback (Section III-A);
- **`cull`**: the `path` fuzzer augmented with our culling-based biasing method (Section III-B1);

TABLE II
UNIQUE BUGS (AND UNIQUE CRASHES BETWEEN PARENTHESES) FOUND BY EACH FUZZER CUMULATIVELY ACROSS THE 10 RUNS, FOLLOWED BY PAIRWISE COMPARISONS FOR COMMON (SET INTERSECTIONS) AND DIFFERENT (SET SUBTRACTIONS) BUGS.

Benchmark	path	pcguard	cull	opp	path ∩ pcguard	cull ∩ pcguard	opp ∩ pcguard	opp ∩ cull	path \ pcguard	pcguard \ path	cull \ pcguard	pcguard \ cull	opp \ pcguard	pcguard \ opp	opp \ cull	cull \ opp
cflow	2 (21)	2 (18)	3 (28)	2 (18)	1 (15)	2 (18)	1 (15)	2 (18)	1 (6)	1 (3)	1 (10)	0 (0)	1 (3)	1 (3)	0 (0)	1 (10)
exiv2	8 (24)	8 (58)	6 (16)	8 (28)	7 (16)	6 (14)	7 (21)	6 (10)	1 (8)	1 (42)	0 (2)	2 (44)	1 (7)	1 (37)	2 (18)	0 (6)
ffmpeg	2 (2)	3 (3)	1 (1)	0 (0)	2 (2)	1 (1)	0 (0)	0 (0)	0 (0)	1 (1)	0 (0)	2 (2)	0 (0)	3 (3)	0 (0)	1 (1)
flvmeta	2 (11)	2 (9)	2 (13)	2 (6)	2 (9)	2 (9)	2 (6)	2 (6)	0 (2)	0 (0)	0 (4)	0 (0)	0 (0)	0 (3)	0 (0)	0 (7)
gdk	8 (58)	7 (42)	11 (90)	9 (70)	7 (35)	7 (37)	7 (40)	9 (63)	1 (23)	0 (7)	4 (53)	0 (5)	2 (30)	0 (2)	0 (7)	2 (27)
imginfo	2 (2)	2 (2)	2 (3)	3 (4)	2 (2)	2 (2)	2 (2)	2 (3)	0 (0)	0 (0)	0 (1)	0 (0)	1 (2)	0 (0)	1 (1)	0 (0)
infotocap	2 (2)	5 (5)	3 (3)	5 (5)	2 (2)	3 (3)	5 (5)	3 (3)	0 (0)	3 (3)	0 (0)	2 (2)	0 (0)	0 (0)	2 (2)	0 (0)
jhead	6 (12)	6 (11)	6 (13)	6 (13)	6 (11)	6 (11)	6 (10)	6 (13)	0 (1)	0 (0)	0 (2)	0 (0)	0 (3)	0 (1)	0 (0)	0 (0)
jq	1 (3)	1 (3)	1 (3)	1 (3)	1 (3)	1 (3)	1 (3)	1 (3)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
lame	4 (6)	4 (6)	5 (7)	4 (6)	4 (4)	4 (4)	4 (3)	4 (5)	0 (2)	0 (2)	1 (3)	0 (2)	0 (3)	0 (3)	0 (1)	1 (2)
mp3gain	3 (9)	4 (8)	3 (12)	3 (10)	2 (5)	1 (2)	2 (5)	2 (3)	1 (4)	2 (3)	2 (10)	3 (6)	1 (5)	2 (3)	1 (7)	1 (9)
mp42aac	7 (20)	8 (21)	8 (27)	6 (15)	6 (16)	7 (18)	6 (14)	6 (15)	1 (4)	2 (5)	1 (9)	1 (3)	0 (1)	2 (7)	0 (0)	2 (12)
mujs	3 (4)	4 (4)	4 (5)	1 (1)	2 (2)	3 (3)	1 (1)	1 (1)	1 (2)	2 (2)	1 (2)	1 (1)	0 (0)	3 (3)	0 (0)	3 (4)
nm-new	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)	0 (0)
objdump	9 (38)	8 (34)	12 (62)	11 (44)	8 (22)	8 (28)	7 (24)	10 (34)	1 (16)	0 (12)	4 (34)	0 (6)	4 (20)	1 (10)	1 (10)	2 (28)
pdftotext	8 (11)	10 (19)	18 (36)	10 (13)	4 (7)	7 (10)	7 (8)	9 (10)	4 (4)	6 (12)	11 (26)	3 (9)	3 (5)	3 (11)	1 (3)	9 (26)
sqlite3	5 (6)	9 (16)	7 (8)	7 (8)	3 (4)	6 (6)	4 (4)	4 (5)	2 (2)	6 (12)	1 (2)	3 (10)	3 (4)	5 (12)	3 (3)	3 (3)
tiffsplit	5 (34)	6 (30)	6 (55)	5 (39)	4 (24)	5 (26)	5 (22)	5 (37)	1 (10)	2 (6)	1 (29)	1 (4)	0 (17)	1 (8)	0 (2)	1 (18)
TOTAL	**77** (263)	**89** (289)	**98** (382)	**83** (283)	**63** (179)	**71** (195)	**67** (183)	**72** (229)	**14** (84)	**26** (110)	**27** (187)	**18** (94)	**16** (100)	**22** (106)	**11** (54)	**26** (153)

- **opp**: the path fuzzer augmented with our opportunistic biasing method (Section III-B2);
- **pcguard**: the default embodiment of edge coverage-guided fuzzing in AFL++.

For all these fuzzers, we enable AFL++'s *cmplog* instrumentation during program compilation to aid them with difficult branches using input-to-state correspondence [27], and the ASAN [28] sanitizer to expose silent memory safety bugs.

Given the richness of the feedback that path-aware fuzzers provide, we set a budget of 48 hours for each fuzzing run, so as to grant enough time (as in, e.g., [9], [29]–[31]) to the fuzzer to leverage the increased visibility over the program under test. We perform 10 runs per experiment. An important aspect for the cull fuzzer is the duration of each fuzzing round: we set it empirically to 6 hours, as this value yielded the best results in a preliminary sensitivity study we conducted[2]. For the opp fuzzer, we split the 48 hours evenly, using at each run a different queue from a 24-hour run with pcguard (we take the early test cases retained in the 48-hour pcguard runs).

Benchmarks and seeds: As we do not introduce new mutation or branch solving techniques, but solely change how a fuzzer differentiates executions, to estimate bug finding capabilities we seek for subjects with multiple bugs reachable by existing fuzzing techniques. Thus, under identical fuzzing budgets on such subjects, variations in retrieved bugs can be reasonably attributed to differences in feedback mechanisms.

Our choice falls on the UNIFUZZ [10] suite, which provides assorted benchmarks for different tasks and input formats. By construction, its programs hold the properties of comprehensiveness (for the different functionalities and vulnerability types they present) and practicality (i.e., at least one bug per subject should be found in a reasonable amount of time), as its authors explain in [10]. The suite has seen extensive use in the literature and comes with seeds that ease reproducibility.

[2]The main takeaway was that runs of 3 or 6 hours yielded similar results for most subjects, with 6 hours performing slightly better in a few cases. As expected, longer durations (e.g., 12 hours) proved detrimental.

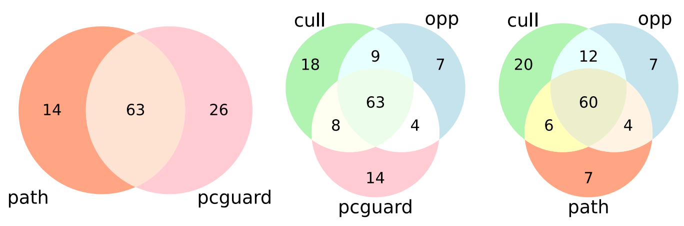

Fig. 3. Venn diagrams (set logical relations) for unique bugs found by the four fuzzers across all benchmarks.

We test the fuzzers on 18 of the 20 UNIFUZZ subjects (Table I). We excluded wav2svf as it does not compile with LLVM and patching its code may threaten soundness, whereas for tcpdump we found several hundreds of unique crashes that make it an outlier in the collection, which would have biased the global comparison and impeded manual bug triaging.

A. RQ1: Bug Finding Capabilities

As the main goal of fuzzing is detecting bugs [32], and our methods aim to contribute to enhancing this task, we first study the crashes exposed by the fuzzers in the tested subjects.

Bug triaging: For a rigorous evaluation, we manually deduplicate the crashes exposed by the fuzzers to estimate their bug finding capabilities. As a preliminary step, we employ standard stack trace hashing-based clustering to identify unique crashes (as per common practices [32], we consider the top 5 frames), and then proceed with manual bug analysis. Table II lists how many **unique** bugs (alongside unique crashes, in brackets) each fuzzer found in our test programs over all the 10 fuzzing trials, and pairwise comparisons between fuzzers that differentiate bugs by their identity. For space reasons, we plot columns only for the most relevant pairs of fuzzers. Appendix B then reports the data points for the median runs.

Next, to allow a more comprehensive, high-level view of the performance of the approaches, we provide the cumulative **inclusion relations** in Figure 3, which show more clearly the differences in unique bugs detected by each fuzzer, as these may be overlooked if only focusing on bug counts.

Baseline approach: In support of our claim that a **baseline fuzzer** that uses path profiles as fuzzing feedback (Section III-A) can already prove its worth, we remark that the path fuzzer discloses, out of a total of 77 bugs, a considerable amount of 14 bugs (18.2%) that the edge coverage-based pcguard fuzzer misses, and only 12 fewer bugs (-13.5%) than pcguard (89) in absolute numbers.

We use a heap overflow detected in cflow as an example of bug uniquely found by path. This bug, an out-of-bounds access to structure token_stack[curs] in parser.c (line 302), occurred when the value of the curs variable was 96, while the structure held only 96 elements at that point in the execution, while no bounds check is present in the code. The underlying cause of this bug is the index curs gradually reaching its limit value token_stack_length through repetitive execution of the same functions, while the function that increments the value of curs is called repeatedly as parse_function_declaration() skips unexpected input tokens in the stack. Even though pcguard could reach the relevant code, it failed to trigger the error, as the problematic value of curs depended on intricate conditions set by distinct paths, each within separate functions, discriminated by our path-aware feedback. Our fuzzer was instead capable of retaining “intermediate” inputs that showed this progression, and it evolved them into a crashing input through mutation.

This example is representative of one of our key experimental findings: all the 14 bugs exposed by path and missed by pcguard involve code portions covered by the pcguard campaign. This backs our claim that our more sensitive path-aware feedback allows a **more pervasive exploration of already-discovered code**, meaning that the edge-driven exploration overlooked some bugs lingering in the depths of code.

Biased exploration: When also considering our two exploration biasing methods, other interesting results emerge.

The cull fuzzer (98 bugs) **surpasses** pcguard in absolute counts by finding 9 more bugs (+10.1%), with a remarkable 27 out of these 98 (27.5%) being missed by pcguard. The path-aware baseline path misses 32 out of these 98 (32.7%) bugs, neatly reflecting the benefits of the optimization.

The opp fuzzer (83 bugs) detects 6 fewer bugs than pcguard (-6.74%) but 6 more bugs than path (+7.8%). However, 16 and 19 out of these 83 (19.3% and 22.9%) are missed by pcguard and path, respectively. In the shared initial 24-hour runs, pcguard found 76 bugs, which as we explained we do not pass to opp. Nevertheless, opp recovers 65 of them (85.5%) in the following 24-hour operation.

Moreover, cull spots 26 bugs missed by opp (+31.3%), while opp finds 11 bugs missed by cull (+11.2%). This highlights the trade-offs between the two biasing methods.

Looking at the bigger picture—namely, the third inclusion diagram from Figure 3—the reader can appreciate that both strategies find 72 common bugs, while path alone finds 77. This indicates that **biasing the exploration significantly impacts the bug finding abilities** of our path-aware fuzzer. Especially cull is capable of exposing many bugs (20) that the other path-aware fuzzers (above all, the baseline approach path) were not able to bring to light.

Further data interpretation led us to believe that *optimizing the baseline has no universal solution*. Neither strategy supersedes the other, as their effectiveness depends on the subject. More importantly, both methods unveiled many bugs that neither the other biasing method nor the path (third inclusion diagram: 20 for cull and 7 for opp) or edge baselines (discussed above and shown in the second inclusion diagram) could expose. This further substantiates our speculation that the path-aware research direction is valuable for future fuzzer design.

Lastly, path-aware fuzzers excel at discovering **alternative ways** of triggering the same bugs, as one can appreciate by looking at the unique crashes each fuzzer detects: 263 for path, 382 for cull (the highest across all of the fuzzers), and 283 for opp, compared to the 289 detected by pcguard.

Zero-days: Testing all the crashing inputs on the latest subject versions revealed 2 enduring zero-day vulnerabilities (1 each for mp42aac and mp3gain). These are found by either or both path and cull, while pcguard misses them.

As a further test, we ran a 1-week fuzzing campaign on 13 actively maintained UniFuzz subjects, 9 of which are tested daily by OSS-Fuzz [18]. For those on OSS-Fuzz, as an indicator of well-fuzzed code, we pick the harness with most bugs reported in the OSS-Fuzz statistics. We do 9 pcguard runs and 9 with the path-aware feedback (6 with our best-performing cull and 3 with path). The path-aware runs revealed another 9 zero-day vulnerabilities: 2 for mp42aac, 4 for objdump, and 1 each for jq, ffmpeg and xpdf. Strikingly, pcguard missed one third of them, and found no others.

We promptly reported all these **11 new bugs** to the developers, following standard practices for responsible disclosure.

PathAFL: We ran the PathAFL fuzzer on the UniFuzz subjects for 10 runs. PathAFL was able to find 33 bugs in total, corresponding to 29.5% of those discovered by cull, 38.9% of those found by the baseline path and 32.5% of those detected by opp. On a positive note, it found 4 bugs that none of our fuzzers found. We provide a detailed report of these results in Appendix C. The data make us believe that PathAFL’s bug finding abilities are much more limited than our path-aware fuzzers (and the standard pcguard, too).

Takeaway: The path-aware feedback enables a more pervasive exploration of code. This reveals a significant number of bugs that the traditional fuzzer overlooks in the code that it covers. The increased visibility can be wielded in multiple ways, exposing many further bugs that the baseline version likely misses for efficiency reasons.

B. RQ2: Runtime Costs

We now inspect the effects our more sensitive path-aware feedback mechanism entails on a fuzzer’s runtime operation.

We study two key fuzzer metrics: the throughput (test case executions per second) of the fuzzers, and the queue size at the end of a run (as queue management is a key driver for fuzzing

TABLE III
MEDIAN QUEUE SIZES AND COMPARISON AGAINST PCGUARD (RATIOS).

Benchmark	path	pcguard	cull	opp	path / pcguard	cull / pcguard	opp / pcguard
cflow	4 420	1 177	3 624	2 637	3.76	3.08	2.24
exiv2	2 619	2 461	1 446	1 689	1.06	0.59	0.69
ffmpeg	23 070	3 947	6 924	12 281	5.85	1.75	3.11
flvmeta	586	470	381	386	1.25	0.81	0.82
gdk	12 508	3 122	6 584	9 326	4.01	2.11	2.99
imginfo	4 771	1727	3 464	4 072	2.76	2.01	2.36
infotocap	222 494	3 571	6 4610	74 295	62.31	18.10	20.81
jhead	1 029	365	831	802	2.82	2.28	2.20
jq	12 982	2 037	6 013	10 024	6.37	2.95	4.92
lame	82 652	2 200	40 784	42 725	37.58	18.54	19.42
mp3gain	11 626	1 623	4 988	3 864	7.16	3.07	2.38
mp42aac	6 001	3 766	3 169	6 104	1.59	0.84	1.62
mujs	8 580	5 216	4 923	38 303	1.64	0.94	7.34
nm-new	21 882	3 678	9 606	14 703	5.95	2.61	4.00
objdump	12 069	4 551	6 286	9 055	2.65	1.38	1.99
pdftotext	13 874	6 400	8 054	15 044	2.17	1.26	2.35
sqlite3	29 254	11 555	8 110	11 029	2.53	0.70	0.95
tiffsplit	26 035	1 160	10 821	17 362	22.44	9.33	14.97
GEOMEAN					**4.46**	**2.22**	**3.15**

performance). While we omit complete data for brevity, we describe the main findings we gathered.

We observe path's fuzzing throughput to be close to pcguard's on average (about 7% lower), while for some subjects it is even higher. Factors contributing to this include not only the instrumentation overhead, but also the increased queue management costs for path, and the diverging prioritization choices the fuzzers made during the campaigns. This value is much smaller than the 1.26x instrumentation slowdown we measured for running test cases in isolation (Appendix A), as other main fuzzer tasks are unaffected by it.

Table III reports the median queue size for each fuzzer. Queue explosion effects are evident: the baseline path-aware fuzzer averages 4.46x more test cases than with the edge feedback. The queue size increases for path can range from a 1.06x factor with exiv2 to peaks of 62.31x with infotocap and 37.58x with lame. When it is considerable, queue explosion is expected to harm the bug discovery efficiency of the fuzzer: for example, path exposes only 2 bugs for infotocap, while pcguard discovers 5.

Our two exploration biasing methods deeply mitigate queue explosion. The cull fuzzer shows an average queue size increase of 2.22x, significantly lower than path's. This value is strongly influenced by pathological subjects like infotocap, lame, and tiffsplit, which exhibit 18.10x, 18.54x, and 9.33x larger queues respectively compared to pcguard, although the difference with path is evident. As expected, opp lags behind cull, with an average 3.15x queue size increase.

Takeaway: The higher sensitivity of the path-aware feedback inevitably entails runtime costs, mainly due to the queue explosion effect. When taming it with exploration biasing strategies, these costs become much more tenable.

C. RQ3: Code Coverage

We examine the code exploration abilities for our approach by analyzing the unique edges traversed, a common metric in fuzzing literature for estimating (path-unaware) exploration. For the sake of space, we present in Table IV only the most significant columns from our coverage study, easing an immediate comparison between our fuzzers and pcguard.

TABLE IV
EDGE COVERAGE ATTAINED BY THE FUZZERS CUMULATIVELY ACROSS THE 10 RUNS, AND SOME PAIRWISE COMPARISONS (SET SUBTRACTIONS).

Benchmark	path	pcguard	cull	opp	path \ pcguard	cull \ pcguard	opp \ pcguard
cflow	1165	1165	1165	1165	0	0	0
exiv2	5252	5387	5338	5064	101	93	25
ffmpeg	15026	23608	15249	22207	306	630	1082
flvmeta	250	250	250	250	0	0	0
gdk	2315	2323	2316	2312	6	7	0
imginfo	2274	2276	2199	2270	4	3	32
infotocap	1239	1392	1328	1392	1	2	3
jhead	197	197	197	197	0	0	0
jq	1967	1996	1954	1993	1	1	0
lame	2691	2693	2683	2693	0	0	0
mp3gain	930	935	931	935	0	3	1
mp42aac	2253	2364	2323	2227	6	76	7
mujs	2685	2744	2656	2711	57	25	2
nm-new	3506	3518	3559	3521	88	185	26
objdump	5615	5043	4914	4843	786	93	30
pdftotext	6270	7813	6693	7072	63	169	93
sqlite3	17188	17699	17199	17748	354	313	212
tiffsplit	1622	1569	1534	1583	105	21	23
TOTAL	**72445**	**82972**	**72488**	**80183**	**1878**	**1621**	**1536**

For the **baseline** path-aware implementation, we observe the impact of a reduced general fuzzing efficiency in favor of an increased visibility: path reaches a lower amount of edges than pcguard for 13 subjects out of 18 (72.2%), the same amount for 3 (16.6%), and more edges only in 2 subjects (11.2%). The total amount of edges we measure for path is **87.3%** (70567 vs. 82972) **of those traversed** in total by pcguard, whereas if we account for edge identity path covers 85.05% (70567) of the edges traversed by pcguard. Occasionally, path reaches edges that pcguard does not: 1878 in total across 13 subjects.

Moving to our two **biasing** methods, the reader can appreciate that cull's benefits are not completely aligned with those of path. Although cull explored only 43 more edges than path and 10,484 fewer than pcguard, it uniquely explored 1,621 edges **missed** by pcguard and, critically, 3,552 edges not reached by path. The queue culling enhancement enabled the fuzzer to explore *distinct code* than in path.

While opp outperforms path and cull, this is largely attributed to its opportunistic initial 24-hour exploration under pcguard. Yet, when opp and pcguard continue with their respective feedbacks on the same initial queues, opp covers 1,536 unique edges, whereas pcguard covers 4,325.

It is evident that increasing the sensitivity of the feedback mechanism can be detrimental for code coverage, especially when looking at absolute counts. Nevertheless, factoring in the results of Section V-A, path-aware fuzzers appear capable of finding different, if not more, bugs over smaller areas of code *because they explore these areas more pervasively*. Additionally, we learned here that they can occasionally unlock code regions that an edge coverage-guided fuzzer struggles to reach.

As, according to Klees et al. [32], no fundamental correlation between maximizing code coverage and finding more bugs has been established, we remark that the decreased coverage should not be considered a fundamental weak spot of our solution.

Future research may look into this efficiency gap, e.g., by interleaving in longer runs path-aware "exploitation" stages with edge-based "exploration" stages that advance coverage.

Takeaway: The more pervasive exploration by the path-aware fuzzer hinders its efficiency in reaching more code. Yet, taming queue explosion allows it to surpass edge coverage-based fuzzing for global bug finding efficiency, despite the lower code coverage. Biasing the exploration proves to also partially affect what code is reached.

VI. Discussion

Our experiments confirm that implementing a path-aware fuzzing feedback mechanism, by tracing acyclic paths within functions, is both feasible and highly promising. This is evident in the significant number of unique bugs detected by our path-aware fuzzers that both the state-of-the-art edge coverage-guided fuzzer `pcguard` and the path-aware PathAFL missed. This increased sensitivity can be leveraged to mitigate queue explosion and supports our belief that intra-procedural path-aware fuzzing offers a rich design space for fuzzer architects.

Missing some bugs with respect to `pcguard` is an expected outcome. Fuzzing is a zero-sum game: since our approach redistributes the available energy, it is natural that some bugs will be lost. The key is that we uncover bugs that would otherwise not have been found. We observed the same phenomenon in other works close to us in spirit (e.g., [4], [33]), where the authors proposed different enhancements for some existing fuzzing technique: in doing so, they missed many bugs from `pcguard` and other baselines, but notably found many that were overlooked with the standard technique.

We consider it a strong point that these results were achieved by altering only a *single component*—the feedback mechanism—of an existing fuzzer. Our approach thus readily benefits from years of research in mutation methods, scheduling, and more, all built upon edge coverage feedback. This warrants further investigation into tailoring other features, like the power schedule, to better leverage our path-aware feedback.

Contrary to previous belief deeming path tracking infeasible for fuzzing [6], [7], we show that intra-procedural path profiles offer a tenable means to achieve such feedback. Unlike the insurmountable number of whole-program paths, our approach allows the fuzzer to globally identify and reason on individual (acyclic) paths traversed during each function activation.

While sensitive feedbacks commonly lead to queue explosion [4], our two orthogonal biasing methods mitigate it, enhancing fuzzing efficiency for bug discovery. These methods each have trade-offs, and neither consistently outperforms the other; rather, they are distinct design points in a promising optimization space for revealing more bugs than current fuzzers. Our key takeaway here is that there is no single right way of wielding the increased visibility a more refined feedback can provide. We hope this may inspire new research directions and encourage the community to work towards that objective.

Nevertheless, in our tests, our culling method already surpasses edge coverage-based feedback in bug finding, both quantitatively and qualitatively, uncovering largely different bugs. However, our proposal *does not aim to replace edge coverage*; alike [4], [33], it rather means to expand the software testing arsenal with a method to uncover many **different bugs** that edge coverage-guided fuzzers struggle to expose.

For this paper, we refrained from proposing selective forms of path sensitivity where only some program regions get accurate path coverage information. As this paper establishes that the path-aware direction is profitable, fuzzer researchers can reason on optimizations of this kind.

A further interesting outcome of our methodology is that path coverage-guided fuzzers often identify many alternative ways to trigger the same bug, via different paths leading to distinct crashing stack traces. This is valuable in practice, as some complex bugs receive partial fixes that address known manifestations but not the root cause. Providing developers with more test cases to triage bugs can help them create more comprehensive fixes. Similarly, with adaptations, a path-aware fuzzer could assist in more thorough software patch testing.

While in this paper we focused on software for which the source code is available, our approach is also applicable to binary-level fuzzing with some engineering effort. The key challenges are (i) identifying function boundaries and (ii) reconstructing the CFG of program functions to compute and inject path increment values. Both might already be attempted statically using binary rewriting, as demonstrated by Retrowrite [34] for edge coverage, and we expect limited impact on fuzzing effectiveness from potential inaccuracies in either step. Dynamic instrumentation in AFL++'s QEMU-User backend would also be possible for great precision [35] and compatibility, although it may face higher runtime overhead.

Threats to validity: We acknowledge the following threats to the validity of our approach. One threat to internal validity is the inherently stochastic nature of fuzzing, which could raise doubts on whether the bugs we discovered can be attributed to the path-aware feedback. We attempt to mitigate this non-determinism by establishing a rigorous experimental setup following fuzzing best practices (e.g., [36]). A second threat of the same type is that manual bug deduplication can be an error-prone activity, but it remains arguably superior to stack trace-based deduplication (which notoriously incurs both under and over-counting issues [32]), and the landscape remains lackluster in semantic automatic methods that readily and reliably work with heterogeneous benchmarks. One threat to external validity is our fuzzer choice, though we note that the use of AFL++ is dominant in recent literature thanks to the many techniques it incorporates from fuzzing research and its high end-to-end performance [37]. Lastly, another threat to external validity lies in the choice of the UniFuzz suite to evaluate our proposal. Yet, we find that its assortment of subjects for different tasks and input types, along with other characteristics argued by its authors [10], mitigates this threat.

VII. Related Works

Paths in fuzzing: We are aware of several works in the fuzzing literature that try to refine the operation of a fuzzer with some notion of "paths". None tracks full intra-procedural execution paths as we do, and a recurrent trait in them is to enhance the fuzzer's capabilities through a loose approximation of path coverage. It is important to stress out that the concept of "paths" that some of these works discuss about differs significantly from ours. For example, by "path" CollAFL [6] refers to an AFL's implementation shorthand, which is the identity of all the edges traversed by the execution, without any notion of ordering. For the works we review next, an analytical comparison with our methods is scarcely possible, hence a qualitative discussion is given.

INSTRIM [38] employs an approximate path differentiation technique to distinguish inter-procedural paths in the program under test, treating some as identical. The work mainly aims at reducing the instrumentation required to track code coverage during fuzzing by enhancing edge coverage with coarse-grained information about inter-procedural paths. Probably due to being a preliminary work, the authors did not report experimental data for the percentage of paths ignored during the fuzzer's execution due to the approximation.

Fang et al. in [39] address the coverage saturation problem in coverage-guided fuzzers [40], where fuzzers repeatedly explore the same paths, leaving security analysts unable to intervene. They propose a human-in-the-loop *directed fuzzing* approach. Using the Ball-Larus algorithm as an external profiling oracle, they gather path coverage and related frequency data. This information is presented to a human agent, who is tasked with prioritizing those seeds that exercise the most neglected paths, thus directing the fuzzer to target specific code areas. While our work also relies on the Ball-Larus encoding to identify paths, our proposal is fully automated and pertains to coverage-guided fuzzing (as opposed to directed fuzzing): we integrate the encoding into the fuzzing process not only to discriminate paths, but also to have an off-the-shelf fuzzer use it to autonomously conduct its exploration of the program.

Some of our readers may be familiar with the *n-gram* feedback. This coverage feedback tracks the history of the most recently traversed blocks, where n is a user-controlled parameter. Trivially, $n=0$ represents block coverage and $n=1$ edge coverage, while higher values yield partial, weak forms of flow-sensitive path tracking. This feedback lacks a rigorous experimental study in the literature: we find it mentioned in two comparative studies [9], [27] of fuzzing techniques, with n empirically set to one of $\{2, 4, 8\}$ and effects on bug finding and code coverage that largely vary depending on the specific program and choice of n. We remark that, with acyclic paths, our approach fully supersedes n-gram coverage while bringing optimal spatial efficiency for the encoding. As future work, in spite of the anticipated queue explosion, we foresee an opportunity in extending our method to track 2-grams of specific acyclic paths, as when exiting loops or crossing function boundaries (as a partial form of context-sensitivity).

VIII. CONCLUSION

We presented a new approach to inform fuzzers with path-aware feedback. While prior work considers run-time path tracking infeasible for the anticipated instrumentation, storage, and performance overheads, we show a tenable way to achieve this exists if we focus on tracing intra-procedural acyclic paths. Our strategy grants an increased visibility to the fuzzer, which becomes capable of discovering bugs overlooked by a traditional fuzzer. We study two methods to wield the queue explosion effect expected with the more sensitive feedback.

Our evaluation shows that a "naive" path-aware exploration is already capable of uncovering many bugs overlooked by edge coverage and by the state-of-the-art path-aware fuzzer PathAFL. Taming queue explosion further boosts the bug finding abilities of our fuzzers, managing to surpass their traditional counterpart with the culling method.

We believe our work reveals significant untapped potential in fuzzing research, and we hope it can draw attention to this new, and promising, path-aware fuzzer design paradigm.

ACKNOWLEDGMENTS

We thank our anonymous reviewers for their feedback. We are grateful to Adrian Herrera, Andrea Fioraldi, and Odysseas Diamadis for the help they provided at the early stages of this project. This work was partially supported by the Italian MUR National Recovery and Resilience Plan funded by the European Union - NextGenerationEU through projects SERICS (PE00000014) and Rome Technopole (ECS00000024), and by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868).

SUPPLEMENTARY MATERIAL

Appendices A to C are available as supplementary material in the IEEE Xplore Digital Library.

DATA AVAILABILITY STATEMENT

This paper comes with an artifact acknowledged as both *Available* and *Reusable* by the Artifact Evaluation Committee of the conference. We make available the artifact as reviewed on Figshare under the DOI 10.6084/m9.figshare.30646583.v2 [41], together with its actively maintained implementation at https://github.com/Sap4Sec/path-aware-fuzzing. We are grateful to our anonymous evaluators for their feedback.

ARTIFACT APPENDIX

A. Abstract

We introduce a lightweight method for tracking intra-procedural execution paths, enabling efficient path-aware feedback for fuzzers. This enhances the fuzzer's ability to detect subtle bugs, even in well-tested software. Our techniques are built atop the AFL++ fuzzer (version 4.07a) and rely on LLVM version 12.0.1. We tested the artifacts on a machine running Ubuntu version 20.04. The fuzzer can run on any commodity hardware that satisfies the aforementioned requirements.

B. Artifact check-list (meta-information)

- **Program:** UniFuzz (automatically retrieved).
- **Compilation:** LLVM 12.0.1.
- **Transformations:** Instrumentation using LLVM IR pass.
- **Run-time environment:** The system was developed and tested on Ubuntu 20.04. Python 3.8 is required by AFL++ and AFLTriage, Rust 1.78.0 is required by AFLTriage.
- **Hardware:** x86-64 platform.
- **Metrics:** Unique crashes, unique bugs.
- **Experiments:** (1) Minimal working example to test the system functionality. (2) Full evaluation on the UniFuzz benchmark.
- **How much disk space required (approximately)?:** ~15 GB if using our Docker images.
- **How much time is needed to prepare workflow (approximately)?:** 10-15 minutes.
- **How much time is needed to complete experiments (approximately)?:** 48 hours – excluding the manual bug analysis.
- **Publicly available?:** Yes.
- **Code licenses (if publicly available)?:** Apache License v2.0.
- **Archived (provide DOI)?:** 10.6084/m9.figshare.30646583.v2

C. Description

1) How delivered: The artifact is available as a self-contained archive at: https://doi.org/10.6084/m9.figshare.30646583.v2, or alternatively from https://github.com/Sap4Sec/path-aware-fuzzing.

2) Hardware dependencies: commodity x86-64 machine (consumer-grade laptop is sufficient for functionality).

3) Software dependencies:

- Ubuntu 20.04 x86-64
- LLVM version 12.0.1
- Python 3.8
- Rust 1.78.0
- Dockerfile provides the full list of dependencies

4) Data sets: UNIFUZZ benchmark.

D. Installation

The AFL++ fuzzer configurations (hereafter "fuzzers") evaluated in the paper, including our path-aware fuzzers and the edge-based baseline, can be built in the following ways:

(a) For a local installation:

```
CC=clang CXX=clang++ \
  LLVM_CONFIG=llvm-config make
```

We also provide a script to automate the installation and environment configuration process: `setup.sh`. Please refer to the artifact's README file for further information.

(b) Using the provided Docker image (recommended):

```
docker build -t path-fuzzing .
```

The image can then be used directly by running:

```
docker run -it path-fuzzing:latest /bin/bash
```

Alternatively, for each UNIFUZZ subject and fuzzer (our path-aware fuzzer `path` and the mainstream `pcguard`) we also provide a dedicated Docker image located under the `unifuzz/<subject>/[path|pcguard]/` path. Each image can be built using the provided `build_image.sh` script and run via the `run_docker.sh` script.

Please refer to the artifact's README file for further details.

E. Experiment workflow

We provide shell scripts to automate the entire testing workflow. A thorough description of these scripts can be found in the artifact's README file.

In short, the `start_session.sh` script is tasked with:

1) Building the subjects using our path-aware instrumentation or the `pcguard` one (cfr. the `build_bench.sh` and the `build_pcguard.sh` scripts).
2) Starting the fuzzing campaign on the target. You can optionally pass the `-cull` parameter to `start_session.sh` to fuzz the program using our queue culling technique (i.e., switching to our path-aware fuzzer `cull`).
3) Invoking the `scripts/deduplicate_crashes.sh` script to automatically deduplicate the crashes detected by the fuzzer (if present) using the AFLTriage tool.

The results of the campaign will be located in the `afl_out_0/` directory. In particular the `u-crashes5/` subdirectory will contain the deduplicated ("unique") crashes discovered during the campaign.

Please beware that in our work we further *manually* analyzed these crashes to derive the unique bugs subsumed by them, namely those showing distinct underlying root causes. Kindly refer to Section VIII-H.

F. Evaluation and expected result

To evaluate our artifact, we performed 10 runs per $\langle subject, fuzzer \rangle$ pair, lasting 48 hours each. Then, we:

1) Manually analyzed all the detected unique crashes to derive unique bugs, which we present as aggregated results in Tables II from Section V-A (cumulative values) and VI from Appendix B (median values).
2) Computed the dimensions of the queues generated by each fuzzer, which we present in Table III.
3) Analyzed the code coverage achieved by each fuzzer over all the testing subjects using the `afl-showmap` tool and a `pcguard`-instrumented binary, which we present in Table IV.

For the `opp` fuzzer, which implements our *opportunistic* strategy, we split the 48 hours evenly, using at each run a different queue from a 24-hour run with `pcguard` (for such a queue, we take the early test cases retained in the 48-hour `pcguard` runs).

G. Experiment customization

Our path-aware fuzzers can be seamlessly employed to test any other piece of software compatible with the AFL++ fuzzer. For brevity, the instructions below are for the baseline path-aware fuzzer.

1) Correctly set the required environment variables:

```
export AFL_LLVM_INSTRUMENT="classic"
export AFL_PATH_PROFILING="1"
```

2) Build the program to be fuzzed using its own building system (`make`, `cmake`, `automake`, etc.), after defining `afl-clang-fast` and `afl-clang-fast++` as the default `C` and `C++` compilers.
3) Start the campaign as one would normally do with AFL++:

```
./afl-fuzz -t 1000+ -i <seeds> \
  -o <output_dir> -m none \
  -- ./<target_program> [<placeholder>]
```

Making sure to set the appropriate values for the `seeds` and `output_dir` directories, and the placeholder to provide the program with an input (usually: `@@`).

H. Notes

For the CGO artifact evaluators: we refer to the *Docker Image (recommended)* section of the README file, and in particular we recommend to follow the steps in the *Minimal Working Example* section to run a minimal working example of our system to explore its functionality.

Please make sure to select reasonable values for the fuzzer's overall runtime and the culling script's fuzzing rounds using the dedicated environment variables (respectively: `RUNTIME` - default: 48h; and `FUZZING_WINDOW_ORIG` - default: 6h):

```
export RUNTIME=10800  # 3 hours
export FUZZING_WINDOW_ORIG=3600  # 1 hour
```

We did not apply for the *Results Validated and Reproduced* badge due to the large number of fuzzing runs we performed in our evaluation (which also require to run for 48 hours) to meet modern fuzzing paper evaluation standards. For the same reason, we find the manual bug analysis step that such evaluation practices recommend (to identify unique bugs from the automatically identified unique crashes) to be beyond reasonable effort for AEC evaluators. Therefore, we do not expect the AEC to fully reproduce our results.

References

[1] M. Payer, "The Fuzzing Hype-Train: How Random Testing Triggers Thousands of Crashes," *IEEE Security & Privacy*, vol. 17, no. 1, pp. 78–82, 2019.

[2] M. Zalewski, "American Fuzzy Lop - Whitepaper," https://lcamtuf.coredump.cx/afl/technical_details.txt, 2016, accessed: 2024-06-12.

[3] T. Ball and J. Larus, "Efficient Path Profiling," in *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*, 1996, pp. 46–57.

[4] P. Borrello, A. Fioraldi, D. C. D'Elia, D. Balzarotti, L. Querzoni, and C. Giuffrida, "Predictive Context-sensitive Fuzzing," in *Network and Distributed System Security Symposium (NDSS)*, 2024.

[5] B. Hardekopf and C. Lin, "Flow-sensitive pointer analysis for millions of lines of code," in *International Symposium on Code Generation and Optimization (CGO 2011)*, 2011, pp. 289–298.

[6] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen, "CollAFL: Path Sensitive Fuzzing," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 679–696.

[7] S. Yan, C. Wu, H. Li, W. Shao, and C. Jia, "PathAFL: Path-Coverage Assisted Fuzzing," in *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, ser. ASIA CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 598–609. [Online]. Available: https://doi.org/10.1145/3320269.3384736

[8] J. R. Larus, "Whole program paths," in *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, ser. PLDI '99. New York, NY, USA: Association for Computing Machinery, 1999, pp. 259–269. [Online]. Available: https://doi.org/10.1145/301618.301678

[9] J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," in *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, 2019, pp. 1–15.

[10] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers," in *Proceedings of the 30th USENIX Security Symposium*, 2021.

[11] "libFuzzer - a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, LLVM Project, 2024, accessed: 2024-06-12.

[12] J. Demott, D. Richard, R. Enbody, D. William, and W. Punch, "Revolutionizing the Field of Grey-box Attack Surface Testing with Evolutionary Fuzzing," *Black Hat USA*, 2007.

[13] D. C. D'Elia and C. Demetrescu, "Ball-Larus path profiling across multiple loop iterations," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, ser. OOPSLA '13. New York, NY, USA: Association for Computing Machinery, 2013, p. 373–390. [Online]. Available: https://doi.org/10.1145/2509136.2509521

[14] M. Bond and K. McKinley, "Practical Path Profiling for Dynamic Optimizers," in *International Symposium on Code Generation and Optimization*, 2005, pp. 205–216.

[15] T. M. Chilimbi, B. Liblit, K. Mehra, A. V. Nori, and K. Vaswani, "HOLMES: Effective statistical debugging via efficient path profiling," in *2009 IEEE 31st International Conference on Software Engineering*, 2009, pp. 34–44.

[16] T. Apiwattanapong and M. J. Harrold, "Selective path profiling," in *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, ser. PASTE '02. New York, NY, USA: Association for Computing Machinery, 2002, pp. 35–42. [Online]. Available: https://doi.org/10.1145/586094.586104

[17] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, "Homo in Machina: Improving Fuzz Testing Coverage via Compartment Analysis," in *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, 2023, pp. 117–128.

[18] "OSS-Fuzz: continuous fuzzing of open source software," https://github.com/google/oss-fuzz, Google, 2016, accessed: 2024-06-12.

[19] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[20] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining Incremental Steps of Fuzzing Research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[21] M. Wang, J. Liang, C. Zhou, Z. Wu, J. Fu, Z. Su, Q. Liao, B. Gu, B. Wu, and Y. Jiang, "Data Coverage for Guided Fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 2511–2526.

[22] G. Ammons and J. R. Larus, "Improving data-flow analysis with path profiles," in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 72–84. [Online]. Available: https://doi.org/10.1145/277651.277660

[23] S. Fujita, "Path Profiling," https://github.com/syoyo/LLVM/blob/master/lib/Transforms/Instrumentation/PathProfiling.cpp, 2011, accessed: 2024-06-12.

[24] A. Ahmed, J. D. Hiser, A. Nguyen-Tuong, J. W. Davidson, and K. Skadron, "BigMap: Future-proofing Fuzzers with Efficient Large Maps," in *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2021, pp. 531–542.

[25] G. Cormode, H. Karloff, and A. Wirth, "Set cover algorithms for very large datasets," in *Proceedings of the 19th ACM International Conference on Information and Knowledge Management*, ser. CIKM '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 479–488. [Online]. Available: https://doi.org/10.1145/1871437.1871501

[26] M. Böhme, L. Szekeres, and J. Metzman, "On the Reliability of Coverage-based Fuzzer Benchmarking," in *Proceedings of the 44th International Conference on Software Engineering*, ser. ICSE '22, 2022, pp. 1–13.

[27] A. Fioraldi, A. Mantovani, D. Maier, and D. Balzarotti, "Dissecting American Fuzzy Lop: A FuzzBench Evaluation," *ACM Trans. Softw. Eng. Methodol.*, vol. 32, no. 2, mar 2023. [Online]. Available: https://doi.org/10.1145/3580596

[28] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A Fast Address Sanity Checker," in *USENIX ATC 2012*, 2012.

[29] S. Li and Z. Su, "Accelerating Fuzzing through Prefix-Guided Execution," *Proc. ACM Program. Lang.*, vol. 7, no. OOPSLA1, Apr. 2023. [Online]. Available: https://doi.org/10.1145/3586027

[30] S. Song, C. Song, Y. Jang, and B. Lee, "CrFuzz: fuzzing multi-purpose programs through input validation," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 690–700. [Online]. Available: https://doi.org/10.1145/3368089.3409769

[31] C. Lyu, H. Liang, S. Ji, X. Zhang, B. Zhao, M. Han, Y. Li, Z. Wang, W. Wang, and R. Beyah, "SLIME: program-sensitive energy allocation for fuzzing," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 365–377. [Online]. Available: https://doi.org/10.1145/3533767.3534385

[32] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating Fuzz Testing," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2123–2138. [Online]. Available: https://doi.org/10.1145/3243734.3243804

[33] A. Fioraldi, D. C. D'Elia, and D. Balzarotti, "The use of likely invariants as feedback for fuzzers," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2829–2846.

[34] S. Dinesh, N. Burow, D. Xu, and M. Payer, "Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1497–1511.

[35] F. De Goër, S. Rawat, D. Andriesse, H. Bos, and R. Groz, "Now you see me: Real-time dynamic function call detection," in *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC '18)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 618–628. [Online]. Available: https://doi.org/10.1145/3274694.3274712

[36] M. Schloegel, N. Bars, N. Schiller, L. Bernhard, T. Scharnowski, A. Crump, A. Ale-Ebrahim, N. Bissantz, M. Muench, and T. Holz, "SoK: Prudent Evaluation Practices for Fuzzing," in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 140–140. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00137

[37] J. Metzman, L. Szekeres, L. Maurice Romain Simon, R. Trevelin Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932

[38] C.-C. Hsu, C.-Y. Wu, H.-C. Hsiao, and S.-K. Huang, "INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing," in *2018 Workshop on Binary Analysis Research (BAR)*, 01 2018.

[39] H. Fang, K. Zhang, D. Yu, and Y. Zhang, "DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling," in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 832–843. [Online]. Available: https://doi.org/10.1145/3650212.3680324

[40] J. R. A. Groce, "The Saturation Effect in Fuzzing," https://blog.regehr.org/archives/1796, 2020, accessed: 2024-11-01.

[41] G. Priamo, D. C. D'Elia, M. Payer, and L. Querzoni, "Artifact: path-aware-fuzzing," 11 2025. [Online]. Available: https://doi.org/10.6084/m9.figshare.30646583.v2