

Path Reduction and Coverage Complexity for Fuzzing

by

Zekun Wang

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2025

© Zekun Wang 2025

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Coverage-guided fuzzing is one of the most effective approaches to automated software testing, yet its performance depends critically on the coverage metric that guides input generation. It is widely assumed that finer metrics—especially *path coverage*, which captures complete control-flow information—should lead to more effective fuzzing. However, practical realizations of path coverage have been limited to restricted forms due to path explosion.

In this work, we introduce a *path reduction* algorithm that bounds loop iterations in execution paths, enabling a practical form of path coverage that preserves essential control-flow information. Despite this advancement, we find that path coverage performs no better than existing metrics such as edge coverage.

To understand this phenomenon, we establish the concept of coverage complexity—a quantitative measure of the granularity of coverage metrics. Analogous to complexity and the Big- \mathcal{O} notation in algorithm analysis, coverage complexity classifies metrics into asymptotic complexity classes such as linear, polynomial, and exponential. This framework provides a structured overview of the entire space of coverage metrics, and guides the design of new coverage metrics.

Our complexity analysis and empirical evaluation on the MAGMA benchmark reveals a consistent pattern: metrics within the same complexity class tend to exhibit similar fuzzing performance, where linear-complexity metrics consistently outperform more complex metrics. This suggests a simple but powerful principle: when designing a new coverage metric, the first step is to determine its complexity class, which serves as an early predictor of its potential performance. Since higher-complexity metrics consistently underperform, our results imply that the family of linear metrics may already represent the optimal frontier of coverage-guided fuzzing, offering—for the first time—a structured overview of the landscape of coverage metrics.

Acknowledgements

I am sincerely grateful to my supervisor, Prof. Meng Xu, for his guidance, support, and countless discussions that shaped this work. I also thank my thesis readers Prof. Chengnian Sun and Prof. Yizhou Zhang for generously offering their time in reviewing my thesis.

Dedication

To my family and my cat.

Table of Contents

Author's Declaration	ii
Abstract	iii
Acknowledgements	iv
Dedication	v
List of Figures	viii
List of Tables	ix
1 Introduction	1
2 Background	5
2.1 Coverage-based Fuzzing	5
2.2 Coverage Metrics	5
3 Path Reduction and Path Coverage	7
3.1 Simple Path Reduction	7
3.2 Whole-Program Path Reduction	9
3.3 Per-Function Path Reduction	10
3.4 Path-Reduction Based Path Coverage	12

4	Complexity of Coverage Metrics	13
4.1	Sensitivity and the Lattice of Coverage Metrics	13
4.2	Complexity of Coverage Metrics	14
4.3	Asymptotic Complexity of Selected Coverage Metrics	14
4.4	Discovery of New Coverage Metrics from Complexity Analysis	16
5	The Coverage Landscape	17
6	Implementation	19
6.1	Instrumentation	19
6.2	Path Hashing	19
6.3	Seed Scheduling	20
6.4	Mutation Strategy	20
7	Evaluation	21
7.1	Experiment Setup	21
7.2	The Cost of Path Coverage(RQ0)	22
7.3	Performance of Path Coverage(RQ1)	22
7.3.1	Number of Bugs Reached	22
7.3.2	Number of Paths and Edges	23
7.4	Metric Complexity and Fuzzing Performance(RQ2)	24
8	Discussion	26
9	Related Work	28
	References	31

List of Figures

5.1 Coverage metrics lattice with asymptotic complexity classes.	18
--	----

List of Tables

5.1	Asymptotic complexity of selected coverage metrics	17
7.1	Instrumentation overhead of path coverage across benchmark targets. . . .	22
7.2	Path-reduction overhead of whole-program path coverage and per-function path coverage across benchmarks.	23
7.3	Triggered coverage for each metric across benchmarks.	23
7.4	Edge coverage and percent change of Path and PFP Coverage relative to Edge.	24
7.5	Geometric mean of normalized edge coverage across all targets (relative to edge coverage = 1.0).	25

Chapter 1

Introduction

Fuzzing has become one of the most effective and widely used techniques for software vulnerability discovery. It is now considered a state-of-the-art approach in both industry and academia, and forms the backbone of large-scale security testing platforms such as Google’s OSS-Fuzz [18].

Fuzzing has uncovered thousands of critical vulnerabilities in widely deployed systems—including Chrome, Firefox, OpenSSL, and the Linux kernel—and has led to the assignment of thousands of CVEs [44] [19] [26] [11] [34].

An important parameter of fuzzing is the coverage metric. During fuzzing, code coverage feedback is used to guide the fuzzing process: if a test case triggers new code coverage, e.g., new lines of code, it is considered interesting and mutated to generate more inputs to be tested on the target program. The coverage metric determines what is considered new code coverage, for instance, this could mean reaching new basic blocks, or new control flow edges. The *de facto* standard of coverage metric fuzzers is edge coverage, and adopted in almost all state-of-the-art fuzzers including AFL++ [13], libFuzzer [27].

It is widely assumed that one can improve fuzzing performance by using more “sensitive” coverage metrics [14, 42, 36]. For instance, edge coverage is more sensitive than basic block coverage, in the sense that a test case triggering new control flow edges does not necessarily trigger new basic blocks, but a test case triggering new basic blocks always trigger new control flow edges, so edge coverage can detect more subtle differences in code coverage among test inputs. In this perspective, path coverage, which tracks entire execution paths and so capture full control flow information, is the ultimate coverage metric. As a result, various attempts have been made to develop path coverage metrics [42, 14].

The challenges of path coverage is multifold. Besides the high instrumentation overhead of tracking and storing full execution paths, there is the well-know path explosion problem, conditions and loops can easily generate an exponential number of paths, meaning an exponential number of seed inputs to be stored and tested in the fuzzing context. Existing attempts of path coverage address this problem by using simplified version of paths. For instance, PathAFL [42], to reduce the number of paths, selectively instruments basic blocks, and keeps paths with high weights according to heuristics. The paths are stored in their hash, computed as the sum of their block IDs as integers, which essentially discards the order of blocks in the path since addition is commutative.

Raw execution paths also carry noises, or, is “oversensitive”. Consider path pX^nq where p, q are paths, and X^n is n repetitions of the same block X , resulting from a loop. Certainly $pq, pXq, pX^2q, \dots, pX^{N-1}q$ are all different paths, but we probably do not want to keep N seeds for them, since they may not exhibit much different behaviors.

To address the oversensitivity of raw paths, we propose a novel *path reduction* algorithm that reduces loops in a path to a bounded number of iterations. For example, if we set the bound to 2, then the path $pX^{42}q$ is reduced to pX^2q . In general, with bound k , path reduction effectively identifies all paths pX^nq with pX^kq for $n > k$. Path reduction reduces uninteresting distinctions in raw paths through path identification, and simultaneously reduces the number of paths to be explore —indeed this resolves the path explosion caused by loops. We also propose per-function variant of path reduction which further resolves the path explosion resulted from consecutive branches by tracking the reduced sub-execution-paths for each function. Note that while the pX^nq example illustrates the basic idea, our algorithm handles complicated cases including arbitrarily nested loops with conditional branches, function calls, and recursion.

We evaluate the proposed path coverage metrics along with standard metrics including edge coverage and block coverage on the standard benchmarks MAGMA [20]. Our results show that per-function path coverage performs similar to block coverage and edge coverage, whereas whole-program path coverage underperforms the other three metrics in the number of bugs found, and the number of new blocks and edges covered, even though it is significantly better at finding distinct paths and reduced paths.

The failure of path coverage, as well as that of other fine-grained coverage metrics, e.g., multiple condition coverage and weak mutation coverage in [41], together with the long dominance of edge coverage in state-of-the-art fuzzers, let us ponder if the pursuit of increasingly complex coverage metrics is likely to improve fuzzing effectiveness, and if there exists a better coverage metric than edge coverage.

To guide exploration of the space of coverage metrics, we formally developed the con-

cept of the *complexity* for coverage metrics. This allows us to identify the asymptotic complexity of coverage metrics. Analogous to the complexity of algorithms, we can categorize coverage metrics into complexity classes: linear, polynomial, exponential, etc. Existing work discusses the *sensitivity* of coverage metrics [36], giving a partial order on coverage metrics for comparing their granularity. Our work further provides a quantitative measure of granularity, which allows us to identify the asymptotic complexity of coverage metrics, and ignore trivial differences in their granularity, in much the same way we identify both $f(n) = n^2$ and $g(n) = n^2 + n$ as $O(n^2)$.

As an example of how complexity analysis can guide the exploration of new coverage metrics: It turns out that the complexity of well-studied coverage metrics are all linear; this includes block coverage, edge coverage, N-gram coverage. Path reduction-based path coverage is also exponential, whereas the per-function variant is also linear. This suggests that there may be coverage metrics with quadratic complexity that are not yet to be explored. To this end, we devise hyper-edge coverage, which tracks the set of hyper-edges, i.e., pairs of basic block (B_i, B_j) where B_i is a predecessor of B_j in the execution path.

We tested a range coverage metrics, block coverage, edge coverage, per-function path coverage, hyper-edge coverage, and whole-program path coverage, with complexity ranging from linear to quadratic to exponential. The experiment is performed on the MAGMA [20] benchmark. The results reveals a pattern: metrics of the same complexity class also have similar fuzzing performance, where the class of linear metrics is the best performing group. In contrast, whole-program path coverage finds 5% less edges and 23% less bugs than edge coverage.

This paper makes the following contributions:

- We design a novel path reduction algorithm, which truncates loops in a path to a bounded number of iterations in linear time. Based on path reduction, we enable full-fledged path coverage. Our experiments shows that using path coverage does not improve the fuzzer’s ability to find new bugs; the impact on finding paths varies by program: up to 1020% more paths found for `libpng_read_fuzzer` and down to 30% less paths found for `tiff_read_rgba_fuzzer`.
- We develop a formal framework, coverage metric complexity, which provides a quantitative method for reasoning about coverage granularity. We also demonstrate how coverage metric complexity can guide the exploration of new coverage metrics with new complexity classes.
- We empirically evaluate the proposed path coverage metrics along with coverage metrics from different complexity classes. The evaluation shows that metrics within the

same complexity class tend to exhibit similar fuzzing performance, so the complexity of a coverage metric may serve as an early predictor of its potential performance. We provide an overview of the space of coverage metrics, ordered by coverage metric sensitivity, grouped by coverage metric complexity, where for each complexity class, we provide its relative fuzzing performance compared to other complexity classes.

Chapter 2

Background

2.1 Coverage-based Fuzzing

Fuzzing is the automatic exploration of program states to discover vulnerabilities. Given a program under test (PUT), a fuzzer explores the program states by generating a large number of test cases, and monitoring the program behavior on these test cases for any vulnerabilities exposed.

Grey-box fuzzers use coverage feedback to guide the fuzzing process. The typical workflow of a grey-box fuzzer is shown in [Algorithm 1](#).

2.2 Coverage Metrics

Code coverage serves as an approximation of the program's full execution path: a sequence of program states resulting from the execution of a program, and consists of, e.g., the program counter, register values, and contents of the stack and heap. Since the full execution path is infeasible to track, code coverage serves as a lightweight proxy to approximate the full execution path.

Some common coverage metrics are:

- Basic block coverage: tracks the set of basic blocks in an execution path.
- Edge coverage: tracks the set of control flow edges, i.e., tuples of consecutive basic blocks in an execution path.

Algorithm 1 Grey-box Fuzzing Algorithm

```
1: Input: Seed inputs  $S$ , Program under test  $P$ 
2:  $seed\_pool \leftarrow S$ 
3: while  $True$  do
4:    $s \leftarrow seed\_pool.schedule\_next()$ 
5:    $run(P, s)$ 
6:   if  $crashes$  then
7:      $report\_bug(s)$ 
8:   else if  $is\_interesting$  then
9:      $seed\_pool \leftarrow seed\_pool \cup mutate(s)$ 
10:  else
11:     $discard(s)$ 
12:  end if
13: end while
```

- Path coverage: tracks the set of paths, i.e., sequences of basic blocks in an execution path.

Edge coverage has become the *de facto* standard in state-of-the-art grey-box fuzzers, including AFL++ [13], libFuzzer [27], as it offers a good balance between precision and performance.

While basic block coverage is still widely used in traditional software testing, profiling, and feedback-directed optimization—for example, in tools such as `gcov` [15], `llvm-cov` [28]—its use in modern fuzzing is limited due to its inability to distinguish different control-flow transitions between the same blocks.

Path coverage has been explored in white-box fuzzers (e.g., KLEE [8], SAGE [16]), but rarely adopted in grey-box fuzzers because the high instrumentation overhead and the well-known seed explosion problem (exponential number of seeds resulting from path explosion).

Chapter 3

Path Reduction and Path Coverage

To make path coverage practical, we need to solve the aforementioned challenges of path coverage:

- The massive amount of seeds resulting from path explosion.
- The oversensitivity of raw paths: if two seeds differ only by one iterates N times in a loop and the other iterates $N + 1$ times, we may not expect them to exhibit much different behaviors.

We propose *path reduction* as the solution, which reduces the number of iterations of a loop to at most k times, where k is a parameter of the algorithm. This solves the second problem by *loop identification*, i.e., identifying all loops with $n \geq k$ iterations. This also mitigates the seed explosion problem since we only need to keep one seed for each class of identified paths.

Below we consider two variants of path reduction, *whole-program path reduction* which results in reduced paths as close to raw paths as possible with loop identification, and *per-function path reduction* which further reduces the number of paths by tracking sub-paths for each function.

3.1 Simple Path Reduction

Before diving into the details of whole-program path reduction and per-function path reduction. Let us consider the simple scenario of reducing the path of a single function body, consisting solely of basic blocks of the function, and without nested calls.

We reduce the path so that each loop is truncated to at most k iterations, where k is a parameter of the algorithm; for each loop with more than k iterations, we keep the first $k - 1$ iterations and the last iteration. Intuitively, given a path, we look for the outermost loops, truncate them to at most k iterations, then recursively descend into and reduce the inner loops within each iteration of the outer loop. For instance, given a path with two outermost loops $2 \dots 2 \dots 2$ and $5 \dots 5 \dots 5$,

```

1 2      2 4 2 5 6 5 7 5 8 9
  3 3 3

```

We reduce the outermost loops to $2 \dots 2$ and $5 \dots 5$,

```

1 2      2 5 6 5 8 9
  3 3 3

```

Then we reduce the inner loops 333 within the $2 \dots 2$ loop, and obtain the reduced path

```

1 2 3 3 2 5 6 5 8 9

```

Generally, given a path p with loops. Let X be the first block that appears multiple times in p , which marks the beginning of the first loop. We can decompose p in the form

$$p_0 X p_1 X p_2 \dots X p_{k-1} X p_k \dots X p_{n-1} X p_n$$

where X does not appear in the sub-paths p_0, p_1, \dots, p_n . Then we can identify the first $n - 1$ iterations, $X p_1, X p_2, \dots, X p_{k-1} X$, of the first loop with loop header X in p with n iterations. In the above example, we have $X = 2$, $p_0 = 1$, $p_1 = 333$, $p_2 = 4$, $p_3 = 5657589$.

We then drop the extra $n - k$ iterations $X p_k \dots X p_{n-1}$, and obtain the partially reduced path

$$p_0 X p_1 X p_2 \dots X p_{k-1} X p_n$$

where the first loop in the path has been reduced. We then recursively descend into iterations p_1, \dots, p_k of the first loop to reduce inner loops, and also reduce p_n for subsequent loops on the same depth with the first loop. This is summarized in Algorithm 2.

Algorithm 2 Simple Path Reduction Algorithm

Require: A path p ; a loop bound parameter k

Ensure: Returns p reduced where each loop in p is truncated to at most k iterations

```
1: function REDUCESIMPLEPATH( $p, k$ )
2:   if  $p$  has no repeated blocks then
3:     return  $p$ 
4:   end if
5:    $X \leftarrow$  first recurring block in  $p$ 
6:    $p_0 X p_1 X p_2 \dots X p_{n-1} X p_n \leftarrow p$ 
7:   for  $i = 1 \dots \min(k - 1, n - 1)$  and  $i = n$  do
8:      $p'_i \leftarrow$  REDUCESIMPLEPATH( $p_i$ )
9:   end for
10:  return  $p_0 X p'_1 X p'_2 \dots X p'_{\min(k-1, n-1)} X p'_n$ 
11: end function
```

3.2 Whole-Program Path Reduction

In *whole-program path reduction*, we track and reduce the entire execution path, so that the reduced path is as close to raw paths as possible, but without the oversensitivity problem of raw paths.

The whole-program path reduction algorithm also reduces the path so that each loop is truncated to at most k iterations, but it takes the execution path of the *entire program*. Thus the path could contain sub-paths corresponding to function calls made by the outer most function call corresponding to the entire path.

The simple algorithm from the previous section no longer works, because blocks between separate function calls interfere. Suppose we have a function `abs` starting with block X , and in a path we call `abs` in three separate places, then we may have a path of the form

$$p_0 X p_1 X p_2 \dots X p_n.$$

However, we do not want to reduce the path to $p_0 X p'_1 X p'_n$ if we set $k = 2$, since the repeated appearances of X does not signify a loop. Indeed, to avoid blocks from different sub-paths corresponding to different call sites from interfering, we need to reduce those sub-paths on their own.

Let p be the whole-program path, resulting from the execution of say function `main`, we can decompose p in the form

$$p_0 f_1 p_1 f_2 p_2 \dots f_n p_n.$$

where f_i is the path of the i -th function call directly made by `main`. Let X be the first block that appears multiple times in $p_0p_1 \dots p_n$, marking the beginning of the first loop in p . We can decompose p in the form

$$q_0Xq_1Xq_2 \dots Xq_m.$$

In simple path reduction, we leave q_0 as is since it does not contain loops; but this time, $q_0 = q_{0_0}f_1q_{0_1}f_2 \dots q_{0_l}f_l$ may contain function calls f_1, \dots, f_l , so we need to reduce each f_i recursively. Then as in simple path reduction, we proceed by dropping extra iterations $Xq_{k+1}Xq_{k+2} \dots$, and recursively reducing q_1, \dots, q_m . Note that the algorithm terminates since eventually we will reach the leaf call whose q_0 does not contain other function calls.

Another problem is recursive calls. If we reduce $N > k$ iterations to k iterations, then we may as well reduce N nested recursive calls to k nested recursions. We achieve this by maintaining an abstract stack parameter s tracking the stack depth of the current path segment, and drop the recursive calls with depth greater than k .

The algorithm is given in Algorithm 3

3.3 Per-Function Path Reduction

Note that by whole-program path reduction, we only solved path-explosion caused by loop iterations, but can still have an exponential number of paths caused by $\mathcal{O}(n)$ consecutive branches. Consider a program `f_1(); f_2(); \dots, f_n()` where f_i has at most N distinct paths after loop identification. Even with whole-program path reduction, we can have up to an exponential number of N^n reduced paths.

To mitigate this issue, we introduce *per-function path reduction*, where we track and reduce the execution path for each function call, which further reduces the number of paths. With per-function path reduction, we have at most a linear number of $n \times N$ reduced paths.

Per-function path reduction is straightforward given whole-program path reduction; we only need to apply the same procedure but to each sub-path corresponding to each function call in the whole path.

Let p be the whole-program path, we again decompose p in the form

$$p_0f_1p_1f_2p_2 \dots f_np_n.$$

Then we replace f_i with its first block $f_i[0]$, which serves as an identifier for callee of sub-path f_i . We keep the first block of the sub-path so that we can distinguish $p_0f_1p_1$

Algorithm 3 Whole-Program Path Reduction Algorithm

Require: A path p ; a loop bound parameter k ; stack depth s mapping function to their depth in the current stack

Ensure: Each loop in p is truncated to at most k iterations

```
1: function REDUCEPATH( $p, s, k$ )
2:   // Compute current stack depth
3:   if  $p[0] \notin s$  then
4:      $s[p[0]] \leftarrow 0$ 
5:   else
6:      $s[p[0]] \leftarrow s[p[0]] + 1$ 
7:   end if
8:   if  $s[p[0]] \geq k$  then
9:     return empty path
10:  end if
11:   $p_0 f_1 p_1 f_2 p_2 \dots f_n p_n \leftarrow p$ 
12:   $X \leftarrow$  first recurring block in  $p_0 p_1 \dots p_n$ 
13:   $q_0 X q_1 X q_2 \dots X q_m \leftarrow p$ 
14:   $q'_0 \leftarrow$  REDUCEHEAD( $q_0$ )
15:  for  $i = 1 \dots \min(k - 1, m - 1)$  and  $i = m$  do
16:     $q'_i \leftarrow$  REDUCEPATH( $q_i$ )
17:  end for
18:   $s[p[0]] \leftarrow s[p[0]] - 1$ 
19:  return  $q'_0 X q'_1 \dots X q'_{\min(k-1, m-1)} X q'_m$ 
20: end function
21: function REDUCEHEAD( $q, s, k$ )
22:   $q_0 f_1 q_1 f_2 q_2 \dots f_n q_n \leftarrow q$ 
23:  for  $i = 1 \dots n$  do
24:     $f'_i \leftarrow$  REDUCEPATH( $f_i, s, k$ )
25:  end for
26:  return  $q_0 f'_1 q_1 f'_2 q_2 \dots f'_n q_n \leftarrow q$ 
27: end function
```

from $p_0.f_2.p_2$ for different callees f_1 and f_2 . We then reduced the whole-path to p' with whole-program path reduction, and record the reduced path for function $p[0]$. For each sub-path corresponding to callee f_i , we recursively reduce and record it with per-function path reduction.

Algorithm 4 Per-Function Path-Reduction Algorithm

Require: A path p ; a loop bound parameter k

Ensure: Record the reduced path for p and the paths of all child calls in p

```

1: function REDUCEPATHPERFUNCTION( $p, k$ )
2:    $p_0.f_1.p_1.f_2.p_2 \dots f_n.p_n \leftarrow p$ 
3:   for  $i = 1 \dots n$  do
4:     REDUCEPATHPERFUNCTION( $f_i, k$ )
5:   end for
6:    $p \leftarrow p_0.f_1[0].p_1.f_2[0].p_2 \dots f_n[0].p_n$ 
7:   REDUCESIMPLEPATH( $p, k$ )
8:   RECORD( $p$ )
9: end function

```

In Algorithm 4, the helper function RECORD use $p[0]$ as the identifier for the function corresponding to p , and save the reduced path for $p[0]$, say in a hash map from function identifier to set of reduced sub-paths.

There is a subtle problem that the $f_i[0]$'s may be repetitive, thus captured by ReduceSimplePath as loops. So let us assume here we are using a fixed version of ReduceSimplePath that ignores the head block of functions when calculating the first repetitive block.

3.4 Path-Reduction Based Path Coverage

Path coverage based on path reduction treats the reduced execution path of a program as the coverage measurement. For each input, we compute its reduced path using the path-reduction algorithm, and the coverage metric records whether this reduced path has been seen before.

Chapter 4

Complexity of Coverage Metrics

In this section, we formally define and discuss the concept of the complexity of coverage metrics, as a quantitative measure of the granularity of coverage metrics. In the same way we study the complexity of an algorithm P by analyzing its running time $T_P(n)$ as a function of the input size n ; we study the complexity of a coverage metric C by analyzing the size $T_C(n)$ of the coverage set of a program of size n .

4.1 Sensitivity and the Lattice of Coverage Metrics

In this section we show the lattice structure of the space of coverage metrics, ordered by the "granularity" of the coverage metrics.

First recall the definitions of coverage metrics and sensitivity from [37]. We define a coverage metric $C \in \mathcal{C}$, formally as a function $C : \mathcal{P} \times \mathcal{I}_{\mathcal{P}} \rightarrow \mathcal{M}$, which given a program P from the program space \mathcal{P} , a test input I from the input space $\mathcal{I}_{\mathcal{P}}$ of P , produces a coverage measurement $M \in \mathcal{M}$.

For two coverage metrics C_1 and C_2 , we say C_1 is *at least as sensitive* as C_2 , denoted as $C_1 \succeq C_2$, if

$$\forall P \in \mathcal{P}, \forall I_1, I_2 \in \mathcal{I}_{\mathcal{P}}, C_1(P, I_1) = C_1(P, I_2) \implies C_2(P, I_1) = C_2(P, I_2).$$

Previous work [37] established the sensitivity partial order on the space of coverage metrics. To define the join operation of a (semi)lattice, we further require that the measurement set \mathcal{M} is a semilattice equipped with a join operator \sqcup . Now for two coverage

metrics $C_1 : \mathcal{P} \times \mathcal{I}_{\mathcal{P}} \rightarrow \mathcal{M}_1$ and $C_2 : \mathcal{P} \times \mathcal{I}_{\mathcal{P}} \rightarrow \mathcal{M}_2$, the join of C_1 and C_2 is the coverage metric $C_1 \sqcup C_2 : \mathcal{P} \times \mathcal{I}_{\mathcal{P}} \rightarrow \mathcal{M}_1 \times \mathcal{M}_2$, where $\mathcal{M}_1 \times \mathcal{M}_2$ is the product lattice, is given by

$$(C_1 \sqcup C_2)(P, I) = (C_1(P, I), C_2(P, I)).$$

Intuitively, the joined metric $C_1 \sqcup C_2$ considers a seed interesting if C_1 or C_2 consider the seed interesting.

One application of the join operator is to fix an issue with path reduction based path coverage. Both variants path coverage is *not* more sensitive than edge coverage, since there could be new edges in the truncated loops. Therefore, whenever we refer to path coverage later, we mean the join of path coverage and edge coverage.

4.2 Complexity of Coverage Metrics

Given a program P , its *saturated coverage set* under a coverage metric C is the set of all reachable coverage measurements, defined by

$$\mathcal{S}_C(P) = \bigsqcup_{I \in \mathcal{I}_{\mathcal{P}}} C(P, I).$$

Now given a metric C , we define the *complexity* of C as

$$T_C(n) = \max_{\forall P, |P|=n} |\mathcal{S}_C(P)|,$$

that is, the maximum number of coverage measurements a program of size n can achieve. Here, $|P|$ denotes the size of the program P , and $|\mathcal{S}_C(P)|$ denotes the cardinality of the saturated coverage set.

4.3 Asymptotic Complexity of Selected Coverage Metrics

We assume that the size of a program P is measured by the number of basic blocks $|P| = n$.

To exclude pathological programs that rarely show up in practice, say a program with a fully connected control flow graph, we make the following practical assumptions:

1. *The number of successors of a basic block is bounded by a constant M .* Conditional and unconditional branches result in at most two successors. For switch statements, the number of cases does not grow with the size of the program— in practice, people do not define an enum type with n variants in a program of size n .
2. *The level of nesting of loops is bounded by a constant L .* The level of nesting of a loop also does not grow the size of the program in practice. Even though in theory one could have a nested loop `loop { loop { loop { ... } } }` of $\mathcal{O}(n)$ levels.
3. *The number of basic blocks in a function is bounded by a constant B .* Function is the unit of abstraction, and its complexity is naturally bounded by the cognitive ability of humans. The Linux kernel [35] has over 40 million lines of code, but the largest function `lustre_assert_wire_constants` is still only 4,179 lines of code.

Now let P be a program with n basic blocks.

Trivial Coverage The trivial coverage \mathcal{C}_\perp is the bottom of the coverage metrics lattice; it is the coarsest coverage metric, given by $\mathcal{C}_\perp(P, I) = \emptyset$. Intuitively, with the trivial coverage, `is_interesting(new_seed)` returns `false` for all inputs. Thus $T(n) = 1 \in \mathcal{O}(1)$.

Block Coverage When the n basic blocks are all reachable, the coverage set is the entire set of basic blocks and so $T(n) = n \in \mathcal{O}(n)$.

Edge Coverage We have at most $n \cdot M$ edges, when they are all reachable we have $T(n) = M \cdot n \in \mathcal{O}(n)$.

N-Gram Coverage The coverage measurement is of the form $(b_1, b_2, \dots, b_{k+1})$, where b_{i+1} is a successor of b_i . Since we have n choices for b_1 , and each b_i has at most M different successors, there are at most $M^k \cdot n$ such tuples. Thus $T(n) \leq M^k \cdot n \in \mathcal{O}(n)$.

Per-Function Path Coverage In a reduced path of a function, a block can appear at most k^L times. So the reduced path has length bounded by $B \cdot k^L$ where each block appears k^L times. Then for each function the number of reduced paths is bounded by $B^{B \cdot k^L}$ — still constant — if we assume that any block may appear in any position of the reduced path. For n functions we thus have $T(n) \leq B^{B \cdot k^L} \cdot n \in \mathcal{O}(n)$; still linear!

Whole-Program Path Coverage For a program with n basic blocks, we have at least $m = n/B$ functions f_1, f_2, \dots, f_m . Suppose each f_i has at least one if-branch, and each f_i calls f_{i+1} . In the case that all branches are reachable, we have at *least* 2^m paths, so $T(n) \geq 2^m = 2^{n/B} = (2^{1/B})^n \in \Omega(2^{n/B})$.

We arrive at the same lower bound if we assume f_1 calls f_2, \dots, f_m sequentially. In essence, it is the $\mathcal{O}(n)$ consecutive branches in a program leading to the path explosion. This is eliminated in per-function path reduction because the path of each function is reduced separately, and after path reduction each function contains only a bounded number of branches.

Raw Path Coverage Since the number of distinct paths could be infinite in the presence of loops without path reduction, we have $T(n) = \infty$.

Discrete Coverage The discrete coverage \mathcal{C}_\top is the top of the coverage metrics lattice, i.e., the finest coverage metric. It distinguishes any distinct inputs, given by $\mathcal{C}_\top(P, I) = \{I\}$. Intuitively, under the discrete coverage, `is_interesting(new_seed)` returns `true` for all inputs. It is easy to see that $T(n) = |\mathcal{I}_P| = \infty$, assuming that the input space is infinite.

4.4 Discovery of New Coverage Metrics from Complexity Analysis

The metrics we analyzed so far leave a gap between linear and exponential, suggesting the exploration of metrics with polynomial complexities.

We propose a *hyper-edge* coverage with quadratic complexity. It tracks the set of hyper-edges in an execution path where a hyper-edge is a tuple of edges $(p[i], p[j])$, $i < j$, in path p . There are at most n^2 such tuples and so the metric is $\mathcal{O}(n^2)$. The $\mathcal{O}(n^2)$ many hyper-edges can be achieved when a program has $\mathcal{O}(n)$ different functions called in an execution path.

Chapter 5

The Coverage Landscape

Table 5.1 and Figure 5.1 summarize the space of coverage metrics discussed in this paper. Coverage metrics can be ordered by their sensitivity and classified by their asymptotic complexity.

Table 5.1: Asymptotic complexity of selected coverage metrics

Coverage Metric	Complexity
Trivial Coverage	$\mathcal{O}(1)$
Block Coverage	$\mathcal{O}(n)$
Edge Coverage	$\mathcal{O}(n)$
N -Gram Coverage	$\mathcal{O}(n)$
Per-function k -Reduced Path Coverage	$\mathcal{O}(n)$
Hyper-edge Coverage	$\mathcal{O}(n^2)$
k -Reduced Path Coverage	$\Omega(2^{n/B})$
Path Coverage	∞
Discrete Coverage	∞

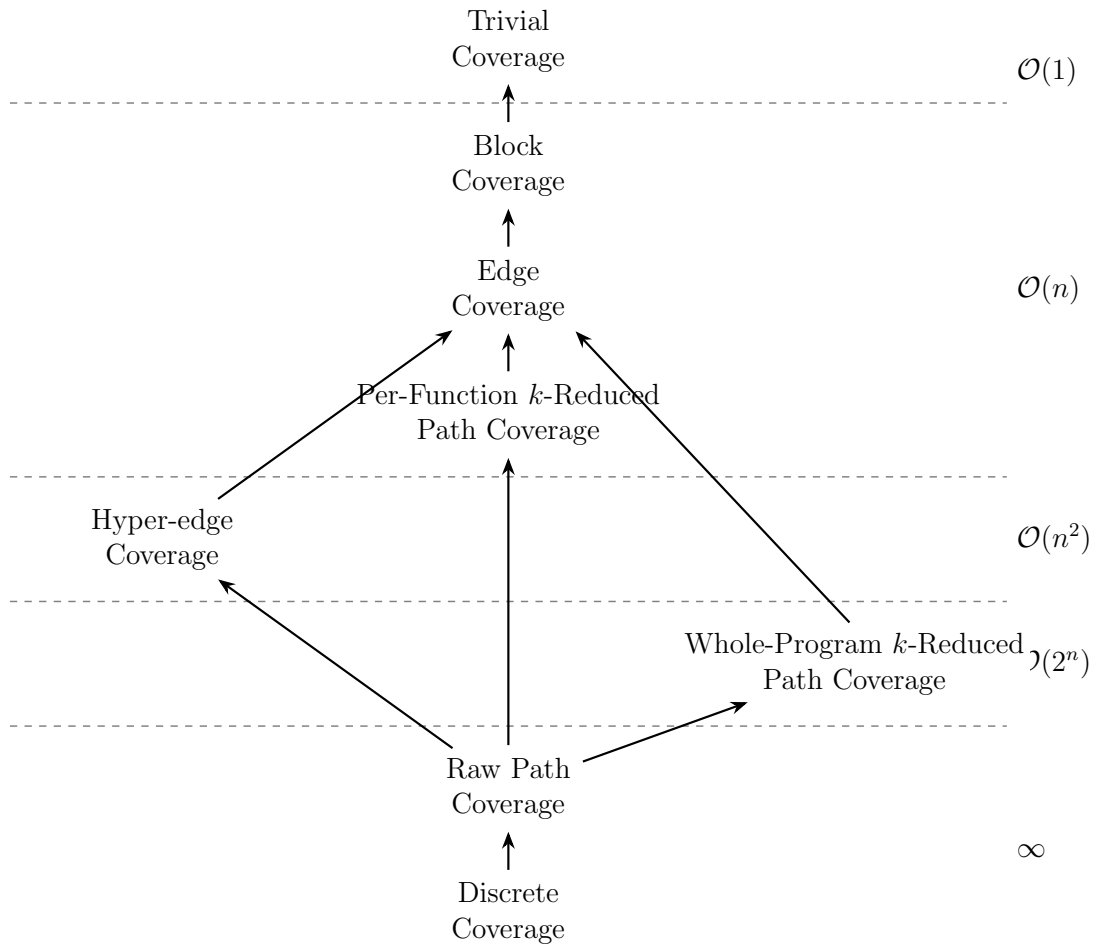


Figure 5.1: Coverage metrics lattice with asymptotic complexity classes.

Chapter 6

Implementation

We implement a naive fuzzer as outlined in algorithm 1, without advanced optimizations such as the forksvr [3], since the emphasis is on the relative performance of different coverage metrics. Our fuzzer supports any coverage metric that is defined above. After executing the instrumented program, the fuzzer reads the execution path from a piece of shared memory, and computes the coverage from the recorded execution path.

6.1 Instrumentation

We instrument programs statically so that the instrumented programs write their execution paths to a piece of shared memory. We assign a globally unique `u32` identifier to each basic block, and the shared memory holds an array of basic block IDs representing the execution path. At the beginning of each basic block, the instrumented program pushes the current block ID to the end of global array holding the execution path. The instrumentation is implemented as a custom LLVM [25] pass, and loaded as a clang [12] plugin.

During instrumentation, we record the entry and exit blocks of each function, this information is used to determine if a block marks the entry or exit of a function in the path reduction algorithm.

6.2 Path Hashing

Since storing every reduced path verbatim would incur substantial memory and comparison overhead, we instead store the hash of each reduced path. We use MD5 as the hash function

because it is fast, widely available, and produces a fixed 128-bit digest with a negligible collision rate for our use case.

6.3 Seed Scheduling

We prioritize a seed based on the uniqueness of the blocks it covers, to encourage exploring rare program paths. For each block in the path of a seed, we look at the total number of times the block has been hit during the fuzzing process so far, and use the minimal hit count as the priority—the lower the hit count, the higher the priority.

When using multiple coverage metrics, we consider a seed triggering new coverage in at least one of the coverage metrics as interesting. Seeds triggering a new coarser coverage, say edge coverage, is given higher priority than seeds triggering a new finer coverage, say path coverage.

6.4 Mutation Strategy

We use *Havoc* [40] as the mutation strategy. *Havoc* randomly selects a stack of 1, 2, 4, ..., 128 mutators from a set of mutators (one mutator can be selected multiple times) and applies them to a seed to generate the mutant. For the mutators, we choose bit flip, byte replacement, chunk deletion, chunk clone, and chunk insertion.

Chapter 7

Evaluation

We aim to answer the following research questions:

RQ0 Is path-reduction-based path coverage a practical form of path coverage?

RQ1 Does path-reduction-based path coverage perform better than existing coverage metrics?

RQ2 What is the relationship between coverage complexity and fuzzing performance?

7.1 Experiment Setup

We ran all experiments on one machine with Intel Core Ultra 9 285K processor (24-Core, 76MB Total Cache, 3.7GHz to 5.7GHz), 32 GB of RAM.

We perform experiments on the standard benchmark MAGMA [20], a widely used benchmark suite for evaluating greybox fuzzers. It consists of real-world software projects (e.g., `libpng`, `libtiff`, `openssl`) into which historical bugs have been systematically reintroduced. Each injected bug has a unique identifier and a deterministic trigger condition, allowing us to treat these bugs as ground truth for evaluation. For each PUT in MAGMA, we ran it for 24 hours and repeated 4 times.

7.2 The Cost of Path Coverage(RQ0)

To evaluate the cost of path coverage instrumentation, we run each program in the benchmark with on all seeds in the benchmark 100 times, and compare the execution time with and without path coverage instrumentation. The instrumentation overhead in relative to the execution path length is summarized in Table 7.1 for some representative targets. Results shows that path-reduction-based path coverage remains practical: its instrumentation and reduction overhead scales with path length but remains within a moderate factor of traditional edge coverage on most targets, even with our prototype implementation.

Table 7.1: Instrumentation overhead of path coverage across benchmark targets.

Target	Avg. Path Length	Overhead (%)
libpng_read_fuzzer	10,526	11.53
tiff_read_rgba_fuzzer	71,135	36.46
tiffcp	188,246	113.70
sndfile_fuzzer	448,868	140.07

We further evaluate the additional cost introduced by path reduction. We run each program in the benchmark with on all seeds in the benchmark 100 times, and time the program execution and the path reduction time respectively. Table 7.1 summarizes the instrumentation overhead and the average path length across benchmark programs. The results show that the overhead of path coverage varies substantially with the average path length of the target program—programs; possibly due to increased trace writes. Table 7.2 summarizes the overhead resulted from reducing paths. Again the overhead increases along with the program paths. Overall, the total cost of path-coverage instrumentation remains within a manageable factor — given that our implementation of the code instrumentation and path reduction can still be substantially improved —demonstrating the practicality of path-reduction based coverage for fuzzing experiments.

7.3 Performance of Path Coverage(RQ1)

7.3.1 Number of Bugs Reached

Table 7.3 shows the number of bugs triggered across the 4 trials of each target per coverage, where PFP stands for per-function path coverage, HE stands for hyper-edge coverage, Path

Table 7.2: Path-reduction overhead of whole-program path coverage and per-function path coverage across benchmarks.

Target	Avg. P Len	Path (%)	PFP (%)
<code>libpng_read_fuzzer</code>	10,526.5	0.47	0.82
<code>tiff_read_rgba_fuzzer</code>	71,134.8	14.98	28.24
<code>tiffcp</code>	188,245.8	25.76	31.04
<code>sndfile_fuzzer</code>	448,867.7	133.76	206.63

Table 7.3: Triggered coverage for each metric across benchmarks.

Target	Block	Edge	PFP	Quad	Path
<code>libpng_read_fuzzer</code>	6	6	4	4	4
<code>tiff_read_rgba_fuzzer</code>	9	11	8	7	7
<code>tiffcp</code>	12	9	5	4	7
<code>x509</code>	4	4	4	4	4
<code>server</code>	4	4	4	4	0
<code>exif</code>	12	12	12	12	12
<code>libxml2_xml_read_memory_fuzzer</code>	16	16	15	12	13
<code>lua</code>	4	4	5	4	4
Total	67	66	57	51	51

stands for whole-program path coverage.

From the table, we see the both variants of path coverage consistently underperform block and edge coverage, where per-function path coverage finds 20% less bug than edge, and whole-program path coverage finds 26% less.

7.3.2 Number of Paths and Edges

Table 7.4 summarizes the median number of paths and edges found by fuzzers guided by edge coverage, and two variants of path coverage, across trials. Column 2, 3 shows the absolute number of coverage found by edge guided fuzzer, and the subsequent columns show the percentage change in coverage found by path fuzzers.

Path coverage shows highly unstable behavior: while it occasionally triggers large increases in the number of explored paths (e.g., a 1020% increase on `libpng_read_fuzzer`),

Table 7.4: Edge coverage and percent change of Path and PFP Coverage relative to Edge.

Target	Edge Fuzzer		Path Fuzzer		PFP Fuzzer	
	Paths	Edges	Paths	Edges	Paths	Edges
exif	1946	10152	-11.23%	-0.99%	-19.80%	-0.91%
libpng_read_fuzzer	6068	2439	+1020.21%	-4.14%	-10.97%	-0.76%
libxml2_xml_read_memory_fuzzer	839	19893	+601.67%	-3.07%	+75.58%	+0.38%
lua	30	8012	+25.00%	-2.70%	-10.00%	-1.65%
server	771	32988	-27.61%	-0.34%	+0.78%	-0.00%
tiff_read_rgba_fuzzer	27130	6794	-29.92%	-18.72%	+17.97%	+3.41%
tiffcp	37721	8121	-5.48%	-16.07%	-34.89%	-6.70%
x509	2106	19000	-9.85%	-0.10%	-40.34%	-0.09%
Average	10451.4	13750.0	+195.10%	-5.02%	-2.71%	-0.55%

these increases rarely translate into improved edge coverage and often reduce it, with an average edge coverage decrease of 5.02%. This instability is consistent with the exponential complexity of raw path coverage, which tends to overspecialize on spurious path distinctions. In contrast, per-function path reduction (PFP) exhibits far more stable performance. Its path counts fluctuate modestly, and its edge coverage remains almost identical to the baseline, with only a 0.55% average decrease. For the coverage growth of each fuzzer, see Appendix.

Both whole-program and per-function path coverage fail to outperform edge coverage in terms of bugs found, and whole-program path coverage in particular trades substantial additional path diversity for a net loss in edge coverage, indicating that finer path sensitivity does not translate into better fuzzing effectiveness.

7.4 Metric Complexity and Fuzzing Performance(RQ2)

For each fuzzer-PUT pair (f, p) , we run T independent trials and compute $M(f, p)$ as the median number of bugs found at the end of the time budget. To compare fuzzers across benchmarks with different scales, we normalize each $M(f, p)$ by the corresponding value of the edge-coverage baseline on the same benchmark:

$$R(f, b) = \frac{M(f, b)}{M(f_{\text{edge}}, b)}.$$

Table 7.5: Geometric mean of normalized edge coverage across all targets (relative to edge coverage = 1.0).

Metric	Complexity Class	Norm. Edge Cov.
Block	Linear	0.986
Edge (baseline)	Linear	1.000
Per-function Path	Linear	0.989
Hyper-edge	Quadratic	0.834
Whole-program Path	Exponential	0.916

We then aggregate these per-benchmark ratios using the geometric mean

$$S(f) = \left(\prod_{p \in P} R(f, p) \right)^{1/|P|},$$

which yields a single scalar score $S(f)$ per fuzzer measuring its ability to find edges compared to edge guided fuzzers. A value $S(f) > 1$ indicates that f outperforms the edge-coverage baseline on average, whereas $S(f) < 1$ indicates weaker average performance.

The result is shown in Table 7.5. Linear-complexity metrics (block, edge, per-function path coverage) all cluster tightly around 1.0, indicating similar overall effectiveness. In contrast, the quadratic metric (hyper-edge coverage) and the exponential metric (whole-program path coverage) both underperform: their normalized scores drop to 0.83 and 0.92, respectively.

Our results suggest that the asymptotic complexity class of a coverage metric is predictive of its fuzzing performance: linear-complexity metrics (block, edge, per-function path) cluster near the best overall effectiveness, while quadratic (hyper-edge) and exponential (whole-program path) metrics systematically underperform despite being strictly more sensitive.

Chapter 8

Discussion

In this section, we discuss possible causes of the poor performance of whole-program path coverage.

Bloated Number of Paths One key factor of the poor performance of whole-program path coverage is the sequencing of *independent* operations.

For instance, consider testing the type checker of a compiler

```
ast = parse(source);
type_check(ast);
...
```

In this pipeline, the parser and type checker are sequential but independent. From the perspective of testing the type checker, the internal path taken by `parse()` is irrelevant—the AST it produces is the actual input to `type_check()`, and we are not concerned about how the AST was parsed. $\text{num_path}(\text{parse}) \times \text{num_path}(\text{type_check})$ paths, even though only $\text{num_path}(\text{type_check})$ distinct behaviors matter. This illustrates a fundamental *anti-pattern* of path coverage: independent operations waste fuzzing effort exploring redundant combinations of independent sub-paths, dragging performance

Local Path Density Traps A second reason whole-program path coverage performs poorly is that it creates path-dense regions may trap fuzzing effort. Even after reduction, consecutive branches can produce exponentially many distinct paths in a small portion of

the program. Instead of distributing effort across the broader state space, the fuzzer may keep discovering new paths locally and repeatedly mutates seeds that stay inside this dense region. As a result, fuzzing energy is concentrated in an possibly unimportant corner of the program, while the rest of the program remains under-explored.

Remote Data-Dependency Noise A third factor is that whole-program path coverage as well as hyper-edge coverage are inherently “global” metrics: they track relationships between program locations that may be extremely far apart in the execution. In real software, however, remote data dependencies are relatively rare. As a result, most of the global relationships these metrics record are not semantically meaningful, but accidental correlations created by control-flow structure or input shape. This leads to highly noisy measurements—tiny, irrelevant changes in distant parts of the execution can produce new hyper-edges or new reduced paths. Because the fuzzer interprets all such global distinctions as potentially important, it ends up reacting to noise rather than signal, further degrading guidance quality.

Chapter 9

Related Work

Black-box fuzzing. Black-box fuzzers generate inputs without inspecting the program under test and typically do not use execution feedback to guide mutations. Classic studies established the efficacy of purely random mutation on real systems [31]. Practical examples include mutation-based file or stream fuzzers such as *zzuf* [2], and protocol generators like SPIKE [1], which can be grammar/model-driven while still operating without program introspection. These approaches are easy to deploy and scale, but their lack of guidance limits deep path discovery in programs with complex parsers or hard-to-satisfy branch conditions.

White-box fuzzing. White-box fuzzing or symbolic execution [22] leverage constraint solving to systematically steer executions toward targeted paths. Exemplars include *SAGE* and *KLEE*, which use path constraints to synthesize inputs that satisfy specific predicates [16, 8]. While powerful for reaching deep states, they face path explosion and solver overhead, and often require heavier engineering.

Hybrid fuzzing. Hybrid designs combine fast mutation-based fuzzing with targeted solver assistance. A common pattern runs a grey-box fuzzer and selectively invokes concolic/symbolic execution when coverage plateaus. QSYM [43] is a state-of-the-art hybrid fuzzer that uses a fast concolic execution engine that combines symbolic execution with native execution tightly.

Other core fuzzer components. Beyond the coverage signal itself, modern fuzzers hinge on several orthogonal components. *Seed scheduling and power schedules* allocate

effort to promising inputs (e.g., AFLFast’s Markov-model schedule and FairFuzz’s rare-branch targeting) [7, 24]. *Mutation engines* range from AFL’s Havoc to adaptive operator selection (MOpt) [29] and data-/taint-guided mutations that steer bytes toward satisfying predicates (Angora, RedQueen) [9, 4]. *Structure-aware generation* leverages grammars or learned models to respect input formats (Skyfire, Learn&Fuzz, NAUTILUS, Superion), improving reachability in parser-heavy targets [38, 17, 5, 39]. *Directed fuzzing* biases exploration toward specific code regions or crash traces AFLGo [6]. Finally, robust *oracles and triage*—sanitizers for memory/UB errors (ASan) and reduction techniques such as delta debugging—make findings actionable and reproducible [32, 45]. These mechanisms are complementary to our focus on the *granularity* (complexity class) of coverage metrics: improving scheduling, mutation, or structure-awareness can amplify any coverage signal but does not by itself change the signal’s asymptotic complexity.

Wang et al. [36] conduct the first systematic study on how different coverage metrics affect greybox fuzzing performance. They introduce the concept of *sensitivity*, compare metrics including branch coverage, context-sensitive branch coverage, *n*-gram branch coverage, and memory-access-aware variants. They find that no single coverage metric is the best, where each metric finds a different set of vulnerabilities. More sensitive metrics chooses generates more seeds, which results in fewer mutation opportunities for each seed, where less sensitive metrics give each seed more mutation opportunities but may miss some interesting ones.

Wu et al. [41] propose a general source-to-source transformation that makes fine-grained coverage criteria out-of-box supported by existing fuzzers. Each fine-grained coverage objective is materialized as an explicit conditional/branch in the transformed program. They use this approach to support multiple condition coverage and weak mutation on AFL++ [13] and QSYM [43]. Their evaluation shows that the effect of using finer coverage metrics is hard to predict and is either neutral or negative in most cases. Subsequently the authors do not recommend using them in general fuzzing scenarios.

Yan et al. [42] introduce PathAFL, a greybox fuzzer that augments AFL [44] by tracking *h-paths*, which are new paths whose edges were all covered before. To reduce the number of *h-paths*, PathAFL does selective instrumentation, and only keep only high-weight *h-paths* according to heuristics. In their evaluation, PathAFL finds 38% more paths and 9.3% more edges over AFL.

Gan et al. [14] show that AFL’s bitmap hashing can induce substantial edge-collision noise, which degrades both path discovery and seed scheduling. CollAFL mitigates this by assigning CFG-aware, near-collision-free edge identifiers and then exploiting the cleaner signal with three seed-selection policies (untouched-branch, untouched-descendant, and

memory-access-guided). Their evaluation on LAVA-M and 24 real-world programs reports higher path discovery and more unique crashes than AFL under comparable budgets, indicating that improving *accuracy* of edge feedback can yield practical gains.[\[14\]](#)

References

- [1] Project 18: Fuzzing with spike (15 pts.). <https://samsclass.info/127/proj/p18-spike.htm>.
- [2] Zzuf – Caca Labs. <http://caca.zoy.org/wiki/zzuf>.
- [3] Fuzzing random programs without `execve()`, October 2014.
- [4] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Ali Abbasi, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [5] Cornelius Aschermann, Sergej Schumilo, Robert Gawlik, Tim Blazytko, and Thorsten Holz. NAUTILUS: Fishing for deep bugs with grammars. In *Network and Distributed System Security Symposium (NDSS)*, 2019.
- [6] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2329–2344, Dallas Texas USA, October 2017. ACM.
- [7] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.
- [8] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.
- [9] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

- [10] Peng Chen and Hao Chen. Angora: Efficient fuzzing by principled search. In *IEEE S&P*, 2018.
- [11] Chromium Project. Fuzz testing in Chromium, 2025.
- [12] Clang Team. Clang: a C language family frontend for LLVM. <https://clang.llvm.org/>. Accessed: 2025-08-13.
- [13] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++: Combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.
- [14] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696, May 2018.
- [15] GNU Project. *Gcov: a test coverage program*, 2025. <https://gcc.gnu.org/onlinedocs/gcc/Gcov.html>.
- [16] Patrice Godefroid, Michael Y. Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. In *Communications of the ACM*, 2012.
- [17] Patrice Godefroid, Hila Peleg, and Rishabh Singh. Learn&fuzz: Machine learning for input fuzzing. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 50–59, 2017.
- [18] Google. OSS-Fuzz. <https://google.github.io/oss-fuzz/>, 2025.
- [19] Google Open Source Blog. OSS-Fuzz: Five Months Later, and Rewarding Projects, May 2017.
- [20] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 4(3):1–29, November 2020.
- [21] Jan Hubicka et al. Feedback-directed optimization in gcc and llvm. In *GCC Developers Summit*, 2012.
- [22] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.
- [23] Robert Swiecki Kupczyk. *honggfuzz*, 2025. <https://github.com/google/honggfuzz>.

- [24] Caroline Lemieux and Koushik Sen. Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *Proceedings of the 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2018.
- [25] LLVM Project. LLVM Compiler Infrastructure Project. <https://llvm.org/>. Accessed: 2025-08-13.
- [26] LLVM Project. libFuzzer – a library for coverage-guided fuzz testing. — LLVM 22.0.0git documentation. <https://llvm.org/docs/LibFuzzer.html#trophies>, 2025.
- [27] LLVM Project. *LibFuzzer: A library for coverage-guided fuzz testing*, 2025. <https://llvm.org/docs/LibFuzzer.html>.
- [28] LLVM Project. *llvm-cov Tool*, 2025. <https://llvm.org/docs/CommandGuide/llvm-cov.html>.
- [29] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. MOPT: Optimized Mutation Scheduling for Fuzzers. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1949–1966, 2019.
- [30] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek 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*, pages 1393–1403, Athens Greece, August 2021. ACM.
- [31] Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, December 1990.
- [32] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, 2012. USENIX Association.
- [33] Daming She, Kexin Pei, Dave Epstein, Suman Jana, Baishakhi Ray, and Ramesh Karri. Neuzz: Efficient fuzzing with neural program smoothing. In *IEEE S&P*, 2019.
- [34] Tyson Smith, Jesse Schwartzenuber, and Sylvestre Ledru. Browser fuzzing at Mozilla, May 2021.

- [35] Linus Torvalds et al. Linux kernel source code. <https://github.com/torvalds/linux>, 2025. Commit 53e760d8949895390e256e723e7ee46618310361, accessed August 12, 2025.
- [36] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu 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)*, pages 1–15, 2019.
- [37] Jinghan Wang, Yue Duan, Wei Song, Heng Yin, and Chengyu 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)*, pages 1–15, Chaoyang District, Beijing, September 2019. USENIX Association.
- [38] Junjie Wang, Bihuan Chen, Lili Wei, and Yang Liu. Skyfire: Data-driven seed generation for fuzzing. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 579–594, 2017.
- [39] Junjie Wang, Bihuan Chen, Lingwei Wei, and Yang Liu. Superior: Grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering (ICSE)*, 2019.
- [40] Mingyuan Wu, Ling Jiang, Jiahong Xiang, Yanwei Huang, Heming Cui, Lingming Zhang, and Yuqun Zhang. One fuzzing strategy to rule them all. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1634–1645, Pittsburgh Pennsylvania, May 2022. ACM.
- [41] Wei-Cheng Wu, Bernard Nongpoh, Marwan Nour, Michaël Marcozzi, Sébastien Bardin, and Christophe Hauser. Fine-grained Coverage-based Fuzzing. *ACM Trans. Softw. Eng. Methodol.*, 33(5):138:1–138:41, June 2024.
- [42] Shengbo Yan, Chenlu Wu, Hang Li, Wei Shao, and Chunfu Jia. PathAFL: Path-Coverage Assisted Fuzzing. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, pages 598–609, Taipei Taiwan, October 2020. ACM.
- [43] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *Proceedings of the 27th USENIX Conference on Security Symposium, SEC’18*, pages 745–761, USA, August 2018. USENIX Association.

- [44] Michał Zalewski. American fuzzy lop: a security-oriented fuzzer. 2014. <https://lcamtuf.coredump.cx/afl/>.
- [45] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.