Revisiting Path Coverage Tracing from a Node-Centric View

HEQING HUANG✉, City University of Hong Kong, China
ZHENDONG SU, ETH Zurich, Switzerland

Path coverage tracing is one of the fundamental components for supporting a wide range of dynamic program analyses, such as testing, debugging, profiling, and many others. Since one needs to insert code into a program to trace its coverage, runtime overhead becomes the main bottleneck for scalability. As finding the minimum number of instrumentation points is NP-hard, extensive work has focused on reducing the number of instrumented edges under diverse assumptions, and thus suffers from the trade-off between precision and efficiency. Departing from this *edge-centric* view, we introduce, in this work, a novel perspective, namely the *node-centric* view, where we aim to find the minimum number of *blocks*, rather than edges as in existing work, that can differentiate all edges and paths in the program. This new perspective allows us to design a *linear-time algorithm* that is *provably correct* and *optimal*—it finds the minimum set of blocks for correctly differentiating edge/path coverage for *arbitrary control-flow graphs*. Our key insight is that optimal node-level instrumentation only needs to distinguish undifferentiated paths at the block where they converge, enabling our algorithm to have linear-time complexity regarding the number of basic blocks.

We implement our algorithm as InsOpt and compare it against state-of-the-art edge-coverage instrumentation techniques on the real-world vulnerability-detection benchmark, Magma. Our evaluation results demonstrate significant improvements: InsOpt needs 2.8x less instrumentation with only 17% basic blocks instrumented. This reduced instrumentation yields a 1.6x speedup and a substantial 2.4x reduction in runtime overhead. Moreover, we also demonstrate substantial potential for InsOpt across other applications. Specifically, our integration of InsOpt with AFL++, a state-of-the-art fuzzer, shows a 5.0x speedup in vulnerability detection and a 1.5x performance improvement. Notably, this efficiency gain further benefits InsOpt in detecting five previously unknown bugs in frequently evaluated projects by other state-of-the-art tools.

CCS Concepts: • **Software and its engineering → Software maintenance tools**; • **Security and privacy → Software security engineering**.

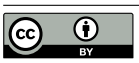
Additional Key Words and Phrases: Coverage tracing, Instrumentation, Node-centric view

ACM Reference Format:
Heqing Huang and Zhendong Su. 2026. Revisiting Path Coverage Tracing from a Node-Centric View. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 332 (October 2026), 27 pages. https://doi.org/10.1145/3839464

1 Introduction

Coverage tracing is a fundamental component for dynamic program analyses such as runtime optimization [11, 31, 32], profiling [7, 12, 22, 43], and testing [15, 20, 21, 25, 28, 42]. It serves not only as the essential metric to quantify analysis effectiveness, but also as an adapted optimization method by analyzing the collected runtime information. For example, fuzz testing [2], one of the most effective techniques within software security and quality assurance nowadays, relies on coverage feedback to effectively guide input generation and steer its examination of different parts of the program. Modern optimizing compilers use profiling data of the program to produce more performant code [32].

Authors' Contact Information: Heqing Huang, City University of Hong Kong, Computer Science, Hong Kong, China, heqhuang@cityu.edu.hk; Zhendong Su, ETH Zurich, Zurich, Switzerland, zhendong.su@inf.ethz.ch.



ACM 2475-1421/2026/10-ART332
https://doi.org/10.1145/3839464

The main goal in coverage tracing is *to record the precise executed path with low overhead*. To this end, the major trend of state-of-the-art methods is *edge-centric*, which utilizes different combinations of control-flow edges recorded through instrumentation to differentiate paths [3, 7, 12, 15, 25, 27, 28, 30, 36, 37, 42]. However, this goal has been proven to be an NP-hard problem to collect accurate path coverage[1] [37, 43]. Moreover, the drastic increase in real-world program complexity further aggravated this problem. For example, coverage tracing accounts for over 70% of the total fuzz testing time in the most widely-used fuzzer, AFL [42]. Distinguishing paths can also suffer from imprecision due to hash collision when indexing a large number of paths originating from loops and iterations [7, 17, 22].

Existing approaches: edge-centric instrumentation. These challenges force existing efforts to balance inevitable tradeoffs. To ensure coverage precision, the main intuition in existing efforts is to instrument fewer edges while retaining accurate coverage achieved after execution [3, 10, 22, 27, 30, 36, 37]. For example, only instrumenting the incoming edges of the branches at basic blocks l, e, j, and s suffices to distinguish all paths of the code shown in Figure 1. However, due to the NP-hard complexity in theory, one line of existing approaches has to propose different assumptions to ignore complex program structures, e.g., loop cycles, as a compromise for efficiency, and design approximated solutions with potential precision loss [3, 22, 27, 30, 36, 37]. Nevertheless, coverage tracing remains costly despite being extensively studied, and its performance varies across projects since these assumptions are not always held. For example, when analyzing large real-world codebases such as the Linux kernel, coverage profiling can introduce over 3x runtime overhead than native execution [44].

To target efficiency, other lines of existing work [7, 20, 25, 28, 29, 41, 42] trade precision for better performance by heuristically removing instrumentation [7, 20, 25, 29, 41] or degenerating to coarse-grained coverage metrics [28, 42]. Although execution speed can be improved by adaptively removing partial instrumentation guided with meta-heuristics or approximations, the lack of precise coverage feedback may degrade the overall performance of its applications. For instance, Untracer [28], a state-of-the-art coverage-guided fuzzing technique, executes 1.55x more inputs within the same time budget than the baseline fuzzer, AFL [1], by heuristically removing the instrumentation once covered for block coverage. However, according to the existing literature [41, 42], the lack of subsequent coverage feedback hinders Untracer's performance by detecting 8.31% less edge coverage than AFL.

This work: novel node-centric view and instrumentation. We observe that the root cause for such a paradox is the so-called *edge trap*: The *edge-centric* view traps existing efforts to explore optimized instrumentation by focusing solely on minimizing the number of *edges* selected, using various dependencies within a limited scope, e.g., domination between neighbor edges, and thus misses the opportunity to find effective solutions based on the entire control-flow graph. In this paper, we revisit this fundamental problem from a novel perspective, namely, *node-centric* edge coverage tracing, to achieve edge coverage by finding the minimum number of basic blocks needed to be instrumented in any given program. Such a novel perspective has demonstrated great potential for addressing the original problem by enabling correct instrumentation, resulting in 2.4x runtime overhead reduction compared to state-of-the-art *edge-centric* approaches in our evaluation. The insight that drives this significant improvement is that paths can be differentiated until they converge on a small proportion of basic blocks (*node*), which are the only places that need to be instrumented. For example, as shown in Figure 1, reaching block j implies that at least one of the edges between blocks f and j must also be traversed. Therefore, if we instrument block j

[1]As a path can be recovered from its edges in theory, we use the instrumentation for edge coverage to facilitate the demonstration in the rest of the paper.

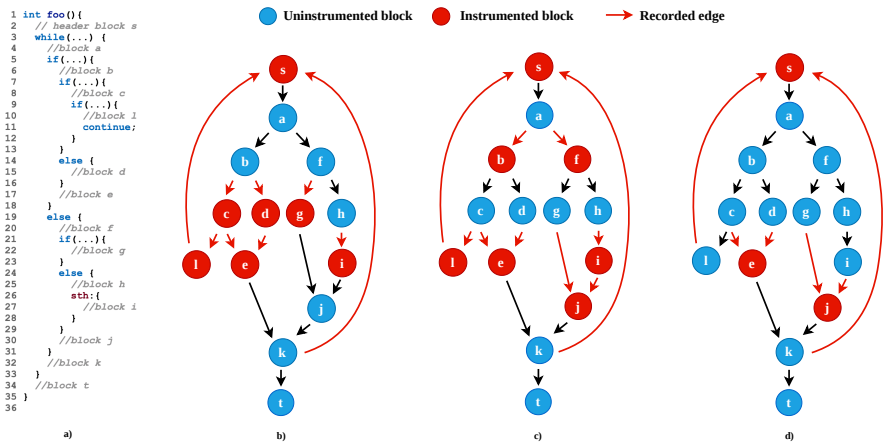


Fig. 1. Motivating example. a) is a program under test. b) is the edge coverage instrumentation generated from existing edge-centric algorithms [3, 15, 36]. 9 edges are recorded by instrumenting 7 blocks, respectively. c) is the instrumentation adapting the approximation algorithm for finding minimal vertex cover [27, 43]. 10 edges connecting to the predecessor blocks of the 7 instrumented blocks are recorded. d) is our edge coverage instrumentation generated from the node-centric view, with only 3 nodes instrumented and 6 edges recorded. We do not explicitly instrument edges; instead, we use the block to recover edge coverage and achieve minimal instrumentation. The details of how we instrument can be found in Section 3.2, Figure 4.

by recording its incoming edges from block g to j and block i to j, we can still differentiate the two edges from block f to g and block f to h, whose instrumentation for coverage tracing can be safely removed. To efficiently identify those blocks that need to be instrumented, we conceptualize coverage tracing as information flow on the control-flow graph, whose coverage information originates from blocks with multiple successors and converges at blocks with multiple predecessors. Blocks that do not lose path information from their predecessors are deemed unnecessary for instrumentation to differentiate those paths containing these blocks. For instance, in contrast to block c in Figure 1, new paths are introduced due to multiple successors of block b and converge at block e. No paths will become non-differentiated if block c is not instrumented. Therefore, there is no need to consider such blocks during instrumentation.

Theoretical and empirical results highlight. Building upon this novel node-centric view, we introduce a lightweight algorithm with linear-time complexity, $O(|V+E|)$, for arbitrary control-flow graphs $G = (V, E)$. This algorithm efficiently determines the blocks that need instrumentation. We formally prove that our algorithm is correct, i.e., the resulting instrumentation is a correct and precise instrumentation. Furthermore, even though we make no claim that our solution is optimal for the original NP-hard problem of finding the minimal set of edges, we show that our algorithm finds the optimal solution for coverage tracing under the node-centric view, specifically for achieving edge coverage with the minimal set of nodes.

We realize our optimal *node-centric* edge coverage tracing algorithm as InsOpt. Our evaluation results are highly promising: InsOpt requires 2.8x less instrumentation than state-of-the-art techniques used in the modern compiler, Clang [23]. Such lightweight instrumentation introduces merely a 45% runtime overhead, representing a 2.4x reduction compared to existing work, and boosts a 1.4x speedup for program throughput. Moreover, we also demonstrate InsOpt's substantial benefits for downstream applications. Our evaluation uses the Magma [19] benchmark, which contains over 100 real-world vulnerabilities, to compare InsOpt against three state-of-the-art fuzzers:

AFL++ [15], Instrim [20], and PGE [25]. The results demonstrate that InsOpt can speed up the fuzzing process with 1.5x more inputs generated. Consequently, InsOpt significantly outperforms existing efforts in detecting vulnerabilities with a 5x speedup. Notably, InsOpt also triggers 11 additional vulnerabilities in Magma that, so far, the other evaluated fuzzers fail to detect. We further demonstrate InsOpt's practical utility in real-world scenarios by identifying five previously unknown bugs in the latest versions of the projects, none of which can be detected by state-of-the-art fuzzers.

Contributions. To sum up, we make the following key contributions:

(1) We introduce a novel perspective, i.e., the *node-centric* view, for path coverage tracing;
(2) We develop a linear-time algorithm and formally prove its correctness and optimality;
(3) We provide empirical evidence that our algorithm is substantially more efficient than state-of-the-art instrumentation techniques; and
(4) We demonstrate the benefits of adapting our algorithm to existing application settings.

2 Motivation

To better demonstrate the problem and our motivation, we first describe the relevant concepts related to coverage tracing in Section 2.1. Subsequently, we summarize the intuition of existing work from the edge-centric perspective and justify their weaknesses in Section 2.2. Finally, we outline our motivation to address these issues in Section 2.3.

2.1 Preliminary Concepts and Representation

Coverage tracing is fundamental for dynamic program analyses such as profiling, fuzzing, and debugging. We first describe the relevant concepts used in coverage tracing to facilitate further discussion and demonstration. Formally, given an arbitrary control-flow graph, $G(V, E)$, with nodes V, representing the basic blocks, and edges E, representing possible control flow between two basic blocks. We use $E(v_i, v_j)$ to denote the edge between nodes v_i and v_j in the graph. On top of this, we use the combination of multiple edges, $\overline{v_1 v_2 ... v_n}$, to represent the path inside the program starting from node v_1 till node v_n. Moreover, the conjunction of a path and subsequent edges, $p = \overline{p' v_1 v_2 ... v_n}$, represents a longer path p with its path prefix p' and successor nodes, $v_1 v_2 ... v_n$. Similarly, we employ the conjunction of paths, $\overline{p_1 p_2 ... p_n}$, to represent a longer path that concatenates paths p_i for $1 \leq i \leq n$.

Example 2.1. Figure 1 presents the control-flow graph of the example program. Each node with an index represents a basic block in the program. These indexes are used to represent the edges between blocks: $E(b, c)$ represents the edge from block b to c. $\overline{abcek}$ indicates the path starts from block a and goes through blocks b, c, e, eventually, ends at block k.

2.2 Edge-centric Coverage Tracing

The goal of coverage tracing is to identify which program paths have been covered during the execution. It not only serves as a metric to quantify the effectiveness of analyses, but also consists of analysis methods as guidance for further optimization. The core intuition behind employing coverage is to examine undiscovered parts of a program from a discovered intermediate program state. Therefore, the granularity of coverage metrics can significantly influence dynamic program analyses. In particular, edge coverage, which requires recording executed edges between basic blocks, is the most widely-used fine-grained metric in the literature.

Intuition behind existing edge-centric coverage. Nevertheless, a fine-grained coverage metric often incurs high runtime overhead due to the extra instrumentation required to record precise information. To maintain effectiveness, efficiently tracing coverage with low runtime overhead

is crucial for scalability. The essential problem for coverage tracing is where to place instrumentation for recording these metrics, i.e., edges, which has been proven to be NP-hard [3, 16]. The state-of-the-art efforts rely on two major intuitions to address this challenge: (1) *to statically reduce the instrumentation locations needed while maintaining the original precision*, and (2) *to dynamically cull the instrumentation on the fly for those covered/analyzed programs to reduce redundant instrumentation.*

These two intuitions both come from the edge point of view: An edge can be represented by other edges so that removing its instrumentation still ensures the recovery of complete coverage information. Such observations motivate the state-of-the-art approaches to minimize the set of edges needed for instrumentation. To precisely identify these edges, most existing efforts leverage various dependencies, e.g., control-flow domination [3, 23], to determine edges represented by other edges and only record the rest.

Example 2.2. Consider the program in Figure 1. Covering edge $E(b, d)$ represents that edge $E(d, e)$ must also be covered following the control dependency. Furthermore, such a dependency indicates that edges $E(b, c)$ and $E(b, d)$ should not be covered within the same loop iteration. Therefore, instrumenting either of the four edges suffices to recover precise coverage information with much less instrumentation. Moreover, as shown in Figure 1b), we demonstrate the results from one of the most optimized instrumentation algorithms [3] adapted in the state-of-the-art compiler, such as Clang [23], and fuzzing, such as AFL++ [15], which records a minimized set of edges to represent others while maintaining the precision of edge coverage. Specifically, they only record edges whose source nodes are neither dominators nor post-dominators in the control-flow graph, such as $E(b, c)$ and $E(b, d)$. Their instrumentation is placed at the source nodes of the chosen edges and records the index given for each edge, and thus blocks s, c, d, e, g, i, and l are instrumented.

Trade-off in existing edge-centric coverage. However, due to the complexity of the program structure, e.g., loop, it has been proven NP-hard to find the optimal solution for coverage [3, 37]. Therefore, most of the existing efforts [3, 7, 8, 20, 27, 28, 30, 36, 37, 41] conduct optimization based on various assumptions to ignore those structures and make their algorithm work **only in the *directed acyclic graph***. Although these methods can accurately reduce instrumentation in the program structure without nested loops or iterations, they usually give an approximated solution to tackle these complex structures so that no precise coverage is collected [20, 27, 28, 30, 41]. Some of them cannot even apply to the programs with cycles and loops. Thus, these approaches are usually designed for specific application scenarios and lack generality for stable performance.

A noteworthy instance of state-of-the-art methods [43] provides an approximate solution for any arbitrary program without restrictions. Specifically, it reduces the minimum-coverage tracing problem to a minimum vertex cover problem. To precisely recover the full coverage, it assumes the instrumentation at each vertex should record all edges. From this perspective, it reduces edge-coverage tracing to a vertex-cover problem and proves that finding the minimal instrumentation is NP-hard. Based on the transformation, it adapts the approximated solution for the vertex cover problem as the algorithm to decide which block should be instrumented to record its connected edges. Surprisingly, we observe that such a theoretical solution may not consistently outperform other state-of-the-art methods under simplifying assumptions since it ignores potential dependencies among these edges in the program. Such contrasting results have motivated our work.

Example 2.3. We demonstrate one potential solution using a vertex cover algorithm to instrument the program shown in Figure 1c). Unfortunately, it requires the same number of instrumented blocks (7 blocks) as the solution mentioned in the previous Example 2.2 on the given program. Moreover, it may require 1 additional edge to be recorded during runtime. This scenario underscores

the challenges and complexities of finding optimal solutions for coverage tracing in programs with intricate control-flow structures.

2.3 A New Perspective: Node-centric Coverage Tracing

The essence of coverage tracing is to record the coverage information propagating on the control-flow graph. The core intuition behind existing efforts on this topic is the ability to distinguish between individual edges in the graph. Once a particular edge can be correctly differentiated using other edges, it becomes possible to safely eliminate instrumentation for those specific edges. However, the main challenge lies in determining whether a given edge or path can be effectively differentiated using the instrumentation currently in place [8].

Revisit the problem from the node-centric view. In this paper, we introduce a novel perspective for tackling the coverage tracing problem: By shifting the focus to a node-centric view, we aim to find the minimum number of blocks to recover the correct edge coverage without any restrictions. This novel alternative view not only offers a compelling angle for understanding the problem but also lays the groundwork for a unique and effective approach to tackling it. Specifically, in the node-centric view, an effective instrumentation method merely requires distinguishing all paths at each node, i.e., basic block, to guarantee accurate coverage analysis. Meanwhile, path differentiation at each node is straightforward and lightweight since we only need to differentiate its incoming edges. Therefore, the node-centric view aims to *find the minimum number of instrumented nodes needed to ensure that all edges can be differentiated.*

Example 2.4. As demonstrated in Figure 2, coverage instrumentation aims to ensure that paths can be differentiated. Specifically, the paths that need differentiation come from the nodes with multiple predecessors shown on the left-hand side. Edge $E(s, t_1)$ and $E(s, t_2)$ propagate the two paths from node s. Nevertheless, it is unnecessary to instrument t_1 and t_2 since these paths can propagate through the control flow graph, and we can still distinguish them later before they become non-differentiable. To illustrate, this path information is not lost if their successor only has one predecessor, as shown in the middle of the Figure. Edge $E(s, t)$ is always differentiated since it is the only edge that reaches block t. Therefore, not recording the edge $E(s, t)$ can still recover the precise coverage information. However, if we do not instrument block t with two predecessor blocks s_1 and s_2 on the right-hand side, we cannot distinguish whether the execution comes from block s_1 or s_2 after executing this block. Therefore, the precise and optimal instrumentation should not lose the path information at block t with multiple predecessors while omitting instrumenting blocks with a single predecessor.

Our insight. This observation, derived from our novel perspective, catalyzes our primary insight, leading toward a more robust formalization of the problem: Rather than focusing on individual edges as existing efforts do, our objective is to guarantee that all edges are differentiable at each basic block. This insight implies that instrumentation is only necessary for those blocks where the path converges to avoid these paths being non-differentiable afterward. If we use instrumentation to distinguish its incoming edges, all paths should be differentiated at this node.

Example 2.5. Motivated by our insight, we can significantly reduce the number of instrumented blocks needed for the program shown in Figure 1. We can identify that paths converge at blocks s, e, j, and k. These are the places where we should place instrumentation to record edges $E(l, s)$, $E(k, s)$, $E(c, e)$, $E(d, e)$, $E(g, j)$, $E(i, j)$, $E(e, k)$, $E(j, k)$, and distinguish these paths. Meanwhile, the instrumented edges are sufficient to recover all paths inside the program, even if the loop exists regarding the recorded edges $E(l, s)$ and $E(k, s)$.

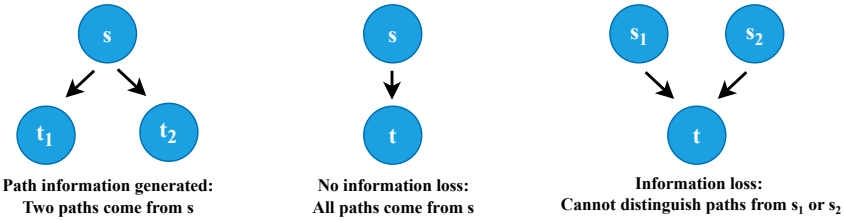


Fig. 2. Insight of InsOpt. From the node-centric view, edge/path coverage information propagates through the control-flow graph and will only be lost when multiple paths converge at one node, which should be the only place for instrumentation.

Intuitive comparison with edge-centric methods. Moreover, this new perspective helps refine the algorithm to have less instrumentation *without any assumptions about the program*. Since each node can distinguish paths reaching this node from its incoming edges, we can pursue the optimal solution by not instrumenting the node that distinguishes the same paths as other instrumented nodes have done. The redundant edges selected by vertex-cover-based algorithms [43] stem from two sources, both of which motivate our node-centric view. First, analogous to the classic vertex cover formulation, these algorithms assume that recording an edge requires instrumenting at least one of its incident vertices. This local view ignores the global propagation of path information along the control flow, resulting in redundant instrumentation at nodes whose paths have already been uniquely differentiated earlier in the execution. Second, vertex cover is NP-hard and cannot be approximated in polynomial time within a factor of 1.36 (unless P = NP) [14]; the approximation gap of practical algorithms further aggravates the instrumentation redundancy. In contrast, our node-centric view can effectively eliminate the redundancy from the first source. Although it does not guarantee an optimal solution to the underlying NP-hard coverage problem, it selects a minimal set of nodes sufficient to recover precise edge/path coverage information for arbitrary programs.

Example 2.6. Following the previous Example 2.5, we notice that block k does not need to be instrumented since all its incoming paths have been distinguished through instrumented blocks e and j: The differentiation for edges $E(e,k)$ and $E(j,k)$ can be indicated from the incoming edges of blocks e and j through their control dependency, respectively. Therefore, we can safely remove instrumentation for this node to obtain the optimal solution under our node-centric view.

3 Node-Centric Edge Coverage Tracing

In this section, we detail the intricacies of achieving precise *node-centric* coverage tracing. We first define the necessary concepts for instrumentation (Section 3.1). Then, we elaborate on the specifics of our optimized instrumentation method, designed for efficient edge coverage tracing (Section 3.2). Finally, we formally prove the correctness and optimality of our algorithm (Section 3.3).

3.1 Preliminary

One crucial difference brought by shifting the view from edge-centric to node-centric is that each instrumented node can distinguish paths going through this node, which eventually differentiates all paths in the program. Therefore, we first formally define the path a node can distinguish using the following definition.

Definition 3.1. (**Path Set**). Given a node, $v \in V$, the path set of v is the set of paths that start from the entry block s and end at block v, denoted as $\Phi(s,v)$. For simplicity, we denote it as $\Phi(v)$. We use $\Phi_u(v_1,v_2)$ to denote the set of paths from block v_1 to v_2 going through the basic block u.

Example 3.1. In Figure 1, the path set of basic block e, $\Phi(e)$, is $\{\overline{abce}, \overline{abde}\}$ (omit the loop for simplifying the presentation).

By introducing the concept of the path set for paths in the node-centric view, we can efficiently identify certain blocks in the program that distinguish the same paths, representing that instrumenting any of these nodes is adequate for distinguishing paths during coverage tracing. Formally, we measure this adequacy for a block v as:

$$\forall p \in \Phi(u), \exists p' \in \Phi(v), p \text{ is the prefix of } p'$$

where u and v are arbitrary blocks in the control-flow graph. In this case, all paths in $\Phi(u)$ can be differentiated at block v, and we can correctly remove the instrumentation for node u.

Example 3.2. Considering the previous motivating program in Figure 1, every path in $\Phi(h)$ is the prefix of a path in $\Phi(i)$. Therefore, we only need to instrument either of the blocks while we can still differentiate these paths, e.g., $\overline{afhi}$. Moreover, we can further notice that every path in $\Phi(i)$ is the prefix of a path in $\Phi(j)$. Therefore, instrumenting block j can distinguish not only these paths going through blocks i and h, i.e., $\overline{afhi}$, but also paths in $\Phi(g)$, e.g., $\overline{afg}$.

Based on the path set, we further define the concept of entropy to measure the path information of each edge that helps path differentiation and avoid redundant instrumentation.

Definition 3.2. (**Edge Entropy**). Given an edge, $e \in E$, the edge entropy of e is the value that indicates whether this edge can be used to differentiate the path involving this edge from other paths. For simplicity, we denote it by a Boolean value, $H(e)$. We use $H(e) = 1$ to denote that this edge can be used for path differentiation. $H(e) = 0$ for the opposite situation.

Example 3.3. In Figure 2, the entropy of edge $E(s, t)$, $H(E(s, t))$ should be zero since this edge does not distinguish all the paths from s to t, $\Phi(s, t)$. On the other hand, the entropy for edges $E(s_1, t)$ and $E(s_2, t)$ should be one because these two edges can help distinguish the paths coming from node s_1 and s_2, respectively.

Based on the concept of edge entropy measuring the path converging at each node, we can now select the minimal set of nodes for instrumentation.

3.2 Tracing Edge Coverage with Instrumented Blocks

To recap, our key intuition for achieving minimal instrumentation for edge coverage is to only instrument blocks where paths converge. Next, we summarize the conditions for identifying such nodes in the control-flow graph.

Where to Instrument. To achieve minimum instrumentation for coverage, we select the basic blocks in the given control flow graph $G = (V, E)$ following Algorithm 1. To facilitate illustration, we assume the graph is added with a single pseudo entry and exit. The key purpose is to use the entropy of each edge to indicate which nodes lead to path merging. We initialize all edges not involved in path differentiation with the entropy, $H(e) = 0$ (Lines 1-3). Then, we check all nodes once based on their number of predecessors and successors using breadth-first search (Lines 6-22). We use $pred(v)$ and $succ(v)$ to denote the set of predecessors and successors of a node v. Specifically, there are three main situations:

1) Nodes with only one predecessor and successor do not generate new paths (Lines 8-9). Its connected edges only transit existing paths from its predecessor node to its successor node.

2) Only nodes with multiple successors, e.g., if-else branches, produce new paths that need differentiation (Lines 10-13). Therefore, we need to mark all its successor edges with 1 to indicate that new paths are generated.

Algorithm 1 Node-centric Instrumentation

Require: Control-flow graph $G(V, E)$ with single entry and exit.

1: **for** $e \in E$ **do** ▷ Initialization
2: $H(e) \leftarrow 0$
3: **end for**
4: $V_{selected} \leftarrow \emptyset$
5: $V_{checked}.enqueue$(EntryNode) ▷ Start with the entry node
6: **while** $V_{checked} \neq \emptyset$ **do** ▷ Checks for all nodes
7: $v \leftarrow V_{checked}.dequeue()$
8: **if** $|pred(v)| = 1 \wedge |succ(v)| = 1$ **then** ▷ Nodes do not generate new path
9: $H(E(u, v)) \leftarrow H(E(v, w)), u \in pred(v), w \in succ(v)$
10: **else if** $|succ(v)| > 1$ **then** ▷ Nodes generate new path
11: **for** $w \in succ(v)$ **do**
12: $H(E(v, w)) \leftarrow 1$
13: **end for**
14: **else if** $|pred(v)| > 1 \wedge |succ(v)| = 1$ **then**
15: **if** $\sum_{u \in pred(v)} H(E(u, v)) = 1$ **then** ▷ Nodes have paths that need differentiation.
16: $H(E(v, w)) \leftarrow 1, w \in succ(v)$
17: **else** ▷ Nodes have no paths that need differentiation.
18: $H(E(v, w)) \leftarrow 0, w \in succ(v)$
19: **end if**
20: **end if**
21: $V_{checked}.enqueue(succ(\text{v}))$
22: **end while**
23: **for** $v \in V$ **do** ▷ Node for instrumentation
24: **if** $|pred(v)| > 1 \wedge \sum_{u \in pred(v)} H(E(u, v)) > 1$ **then**
25: $V_{selected} \leftarrow \{v\} \cup V_{selected}$
26: **end if**
27: **end for**
28: **return** $V_{selected}$

3) When a node does not generate new paths with only one successor, but many incoming paths merge in a node (Lines 14-20).

Suppose there are no edges from its predecessors that bring new paths that need differentiation ($\sum_{u \in pred(v)} H(E(u, v)) = 0$), indicating its successor edge does not contribute to edge differentiation and vice versa. Eventually, we only need to instrument those nodes where multiple converging paths have not been differentiated ($|pred(v)| > 1 \wedge \sum_{u \in pred(v)} H(E(u, v)) > 1$).

From the intuition perspective, we can directly instrument all nodes where paths converge ($|pred(v)| > 1$) to ensure correctness. Nevertheless, such instrumentation introduces redundancy since some paths can be differentiated at multiple nodes. Therefore, our instrumented predicates are guided by ideal scenarios: We instrument nodes only if their incoming paths have not been differentiated elsewhere ($\sum_{u \in pred(v)} H(E(u, v)) > 1$), while the node itself exhibits path convergence ($|pred(v)| > 1$). The non-zero entropy of the incoming edges indicates these paths should be differentiated from others. If we do not instrument this node, these paths can be differentiated since they merge after executing this node. On the other hand, entropy also indicates that those edges do not contribute to path differentiation. Zero entropy indicates those paths that the predecessor/ancestor nodes have differentiated. Hence, even if we do not instrument these nodes whose

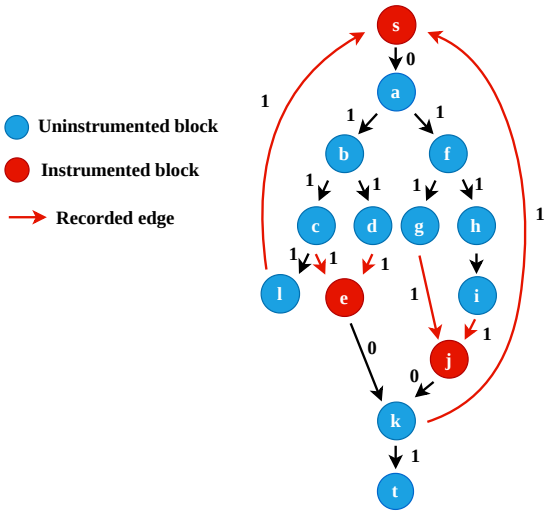


Fig. 3. Example of how InsOpt calculates the entropy of each edge and selects the node for instrumentation. The number on each edge is the entropy calculated by Algorithm 1.

sum of entropy is zero, all incoming paths are still differentiable, and the instrumentation remains correct. Unlike existing approaches based on post dominance [3], which can only identify where control-flow paths must merge, our entropy-based criterion further reduces the number of nodes needed to differentiate paths.

Example 3.4. In Figure 3, the blocks where paths converge are e, j, k, and s. Instrumentation is essential for blocks s, e, and j as it is necessary to differentiate paths stemming from different branches of blocks b and f, which is indicated by the entropy propagated from nodes b and f. Moreover, instrumenting s can also help distinguish paths with different loop iterations. However, it is unnecessary to instrument block k because edges $E(e,k)$ and $E(j,k)$ can be distinguished using the coverage information recorded by the instrumented nodes e and j. This situation can also be reflected by the entropy of edges $E(e,k)$ and $E(j,k)$, which are all zero. Therefore, to achieve correct and minimal instrumentation in this example, it suffices to instrument only blocks e and j.

This node selection condition can also be viewed with a two-step algorithm: First, we examine all nodes with multiple predecessors, which signify path convergence. As depicted in Figure 2, we can ensure the differentiation of all paths by recording all edges within these selected nodes. Second, we further refine the selection of nodes, as some may differentiate the same paths from others. In the node-centric view, we need to ensure that its multiple incoming edges do not bring path information indicated by the entropy, which indicates that their paths have not yet been recorded.

Remark 1. The concept of entropy aims to defer instrumentation until the edges are about to become non-differentiable, which offers an efficient approach to identifying the necessary nodes for instrumentation, ensuring that no paths are non-differentiable. Meanwhile, the algorithm is lightweight: it iterates over all nodes once in a breadth-first search to track how the path propagates, as indicated by the entropy. Compared with the analysis in existing efforts [3, 7, 36] to identify these transitive nodes, not distinguishing coverage with either dominator analysis or iterative reachability analysis, which can cause at least $O(|V+E|^2)$ complexity to reach a fixpoint relying on

```
; LLVM IR
b:    ; label for block b
  %prev_index = %index_b

c:    ; label for block c
  %edge_index = hash(%prev_index, %index_c)
  store_index(%edge_index)
  ...

d:    ; label for block d
  %edge_index = hash(%prev_index, %index_d)
  store_index(%edge_index)
  ...
```

Edge-centric instrumentation

```
; LLVM IR
b:    ; label for block b
  ...

c:    ; label for block c
  ...

d:    ; label for block d
  ...

e:    ; label for block e
  %index = phi i32 [%index_c, %c], [%index_d, %d]
  store_index(%index)
  ...
```

Node-centric instrumentation

Fig. 4. Edge-centric vs. node-centric instrumentation. The edge-centric view requires instrumenting two blocks for each edge. Additional memory IO overhead is also needed to preserve the predecessors' index. Instead, the node-centric view allows us to merge the relevant edge instrumentation at the selected block to reduce runtime overhead. Specifically, an additional *phi* instruction is used to differentiate the predecessor blocks during the execution.

simplifying assumptions, the node-centric view can efficiently find these nodes for instrumentation with $O(|V + E|)$ complexity, even considering programs with loops.

How to Instrument. In contrast to existing work [23, 30, 41, 42], which involved instrumenting two blocks for each edge, we record edges by instrumenting only the selected blocks, not their predecessors, to reduce instrumentation. The edge-centric view is the crucial reason that leads to the existing instrumentation for the edge coverage: As the edge is an abstract concept in the control-flow graph, the edge-centric view makes existing efforts instrument both the predecessor and the ending block of an edge to completely record it, following the definition. However, such a design results in more instrumented blocks overall. Existing methods need to maintain a global memory to preserve the index of predecessor blocks for edge index calculation, which introduces additional runtime overhead. Meanwhile, a significant number of indexes needed in practice can introduce correctness issues even if various `hash` functions are used due to hash collisions [17].

Example 3.5. As shown in Figure 4, which is the partial control-flow graph shown in Figure 1, we use blocks b, c, d, and e to demonstrate how existing instrumentation records edges. To record edges $E(b, c)$ and $E(b, d)$, instrumentation needed to be placed at all three blocks, b, c, and d, as shown in Figure 4. Overall, even though existing instrumentation for edge coverage, i.e., fuzzing [15], has adopted the dominator algorithm [3] to reduce the number of recorded edges by assigning an index for each edge, the edge-centric view still makes it instrument all four blocks, c, d, g, and i for edges, $E(b, c)$, $E(b, d)$, $E(f, g)$, and $E(h, i)$, respectively.

Since we shift to the node-centric view, we aim to record edges by only instrumenting the selected blocks without their predecessors. Apart from having fewer instrumentation sites, we also require fewer instrumented instructions than conventional edge-centric instrumentation to distinguish these edges, which can result in less runtime overhead. Meanwhile, while the number of instrumented blocks is reduced, each instrumentation carries sufficient information about all execution paths leading to that block. Specifically, we record the predecessors' labels of the basic block in LLVM. We instrument an additional `phi` instruction to track the label of the predecessor that executed immediately prior to the current instruction. This label is then used as the index for the executed edge. Therefore, we require fewer instrumentation blocks with less calculation, as shown in Figure 1d. We also record the number of times each edge is executed to approximate path coverage.

Example 3.6. We use the same control-flow graph used in the previous example, shown in Figure 4, to demonstrate our instrumentation. As e is selected to differentiate paths from blocks c and d, we insert a `phi` instruction at the beginning of block e. The parameters of `phi` involves the label of blocks c and d, indicating which block is the predecessor during the execution. Then, it returns the given index of the executed block so that we can calculate the edge index for recording the coverage. Compared with conventional edge-centric instrumentation, we require fewer instrumented instructions and do not need to calculate an additional edge index. Overall, we only need to place instrumentation at three blocks, s, e, and j, to obtain the correct edge coverage in Figure 1.

Loop handling. Unlike existing approaches that remove back edges or ignore cyclic control-flow graphs, our node-centric view of determining necessary edges for path differentiation at every node enables us to consider coverage within a much smaller node-local scope—rather than the unbounded space of paths due to loops. Our node-centric algorithm 1 can be directly applied to control-flow graphs with loops, as there is no restriction for the input graph G. It iteratively checks every node and its connected edges, including back edges, to determine the correct, minimal set of nodes, including loop-entry nodes to be instrumented (proved in Section 3.3). Since a path can be fully recovered from its sequence of edges, we can distinguish and recover paths involving loops based on the sequence of edges recorded by the instrumentation proposed. Regarding the implementation, as we deploy our algorithm for coverage-based fuzzing, an important application scenario, we adopt their implementation [15] to use edge frequency to approximate the original path (edge sequence) for better efficiency. For extreme corner cases such as non-terminating loops caused by logic errors, we can add a pseudo loop-exit block to the graph to ensure that non-terminating behavior is captured. This extension integrates seamlessly with our algorithm, which supports arbitrary control-flow graphs with a single entry and a single exit.

Remark 2. Our intuition dictates that paths should be recorded at the latest when they are on the cusp of becoming indistinguishable due to their convergence at specific nodes. To this end, we employ a node-centric algorithm to differentiate incoming paths by selectively instrumenting these nodes. This algorithm substantially reduces the necessity for instrumented nodes compared to existing edge-centric methods, while further supporting generalized control graphs with loops and unstructured control flow. On the other hand, the node-centric view also minimizes the instrumentation instructions needed to record edges by omitting the calculation of the edge index, which is widely used in existing instrumentation. Even though it is orthogonal to the paper's primary focus, namely reducing the number of sites for instrumentation (i.e., where to instrument), our current implementation using `phi` instructions still requires less instrumentation than existing edge-centric methods. As a result, our algorithm precisely records all paths for arbitrary programs with less runtime overhead.

3.3 Proof of Correctness and Optimality

Even though recovering edge coverage from node coverage is challenging without any complexity analysis [8], we would like to formally prove that our algorithm is correct and optimal after intuitively illustrating its strengths. In the node-centric view, correctness means that all paths in the given program can be differentiated using the instrumented set of nodes, while optimality means that this set is minimal.

Proof of Correctness. Let $G(V, E)$ be any control-flow graph. We assume, without loss of generality, *G has a single entry s and a single exit t*.

Lemma 3.1 (Correctness)**.** Let $I \subseteq E$ be any correct instrumentation of $G(V, E)$. For any node $n \in V$ and any paths $p_1 \neq p_2$ from the entry s to n, we have $\tau(p_1) \neq \tau(p_2)$, where $\tau(p)$ is the mapping for a path p to its set of instrumented edges.

Proof. Let p be any path from n to the exit t. $\overline{p_1 p}$ and $\overline{p_2 p}$ are two distinct paths from s to t. Since I is a correct instrumentation, $\tau(\overline{p_1 p}) \neq \tau(\overline{p_2 p})$. Therefore, $\tau(p_1) \neq \tau(p_2)$. □

Recap our instrumentation condition I from 1, $I = \{v \mid |pred(v)| > 1 \wedge \sum_{u \in pred(v)} H(E(u, v)) > 1\}$. We now prove that the set of nodes in I induces a correct edge-level instrumentation based on the correctness described by Lemma 3.1, which indicates that each path in the given program should have a unique representation using the instrumented nodes.

Proof. We establish the correctness of I via proof by contradiction. First, we suppose the contrary from Lemma 3.1: Let $p_1 \neq p_2$ be two paths from s to node n, such that $\tau(p_1) = \tau(p_2)$ and let p_1 and p_2 be the shortest of such paths. We then have two cases:

Case 1: $|pred(n)| = 1$**.** Let u denote the unique predecessor of n. Let $p_1 = \overline{p_1' E(u, n)}$, $p_2 = \overline{p_2' E(u, n)}$. Therefore, $p_1' \neq p_2'$ are two paths from s to u, and $\tau(p_1') = \tau(p_2')$. This contradicts the fact that $p_1 \neq p_2$ are the shortest such paths.

Case 2: $|pred(n)| > 1$**.** Let $p_1 = \overline{p_1' E(u_1, n)}$, $p_2 = \overline{p_2' E(u_2, n)}$ where $u_1 \neq u_2$ and $u_1, u_2 \in pred(n)$. Since $\tau(p_1) = \tau(p_2)$, the edges $E(u_1, n)$ and $E(u_2, n)$ are not instrumented. Therefore, $n \notin I$, and thus, $\sum_{u \in pred(n)}, H(E(u, v)) \leq 1$.

Case 2.1: $\sum_{u \in pred(n)} H(E(u, n)) = 0$. We assume p_1 and p_2 merge at node w, where w is the ancestor of n. Therefore, there is no node between w and n is instrumented, otherwise, $\tau(p_1) \neq \tau(p_2)$, a contradiction. Since all $H(E(u, v)) = 0$ and all of these nodes are not instrumented, these nodes have to have only one successor. However, p_1 and p_2 are different paths, indicating $succ(w) > 1$, according to our algorithm 1 for assigning entropy, a contradition.

Case 2.2: $\sum_{u \in pred(n)} H(E(u, n)) = 1$. Therefore, $\exists u \in pred(n), H(E(u, n)) = 1$. Without loss of any generality, we assume it is u_1, and we further have $H(E(u_2, n)) = 0$. Similarly, we assume p_1 and p_2 merge at node u, and no node between w and n is instrumented under the same situation discussed in Case 2.1. However, since $H(E(u_2, n)) = 0$, there must be a node, u', between w and u_2, and u' is instrumented to distinguish the path $p_2 \in \Phi_{u'}(s, n)$ and other paths in $\Phi_{u'}(s, n)$. Therefore, the entropy of its successor edges can become 0, leading to $H(E(u_2, n)) = 0$. Hence, $\tau(p_1) \neq \tau(p_2)$, a contradiction.

Therefore, in all situations, there is no path p that the instrumentation, I, cannot differentiate. □

Proof of Optimality. To be an optimal (edge-level) instrumentation with minimum instrumented nodes, we need to meet the condition:

$$\nexists I', |I'| < |I| \wedge I' \text{ is correct.} \tag{1}$$

where I and I' are the set of instrumented basic blocks that derive a correct instrumentation. We assume I' only instrumenting the incoming edges as I does when selecting a node.

Since we proved I is a correct instrumentation, the selected nodes are sufficient instrumentation to differentiate all paths in G. Therefore, if every node $n \in I$ is necessary to differentiate all paths, indicating n should exist in the optimal instrumentation; Otherwise, we cannot differentiate the paths going through n, or we need more nodes instrumented. Therefore, I is the sufficient and necessary set of nodes to differentiate all paths, and thus it should be the optimal instrumentation with the minimal number of nodes selected.

Proof. Based on this intuition, we establish the optimality of I via proof by contradiction. We assume there is a correct I' with fewer nodes needed to differentiate these paths, $|I'| < |I|$. Therefore,

there is a node $n \in I, n \notin I'$. Since $n \in I, |pred(n)| > 1 \wedge \sum_{u \in pred(n)} H(E(u,n)) > 1$. Let u_1 and u_2 be two distinct predecessors of n. Without loss of generality, we assume $H(E(u_1,n)) = H(E(u_2,n)) = 1$.

Regardless of whether n is necessary for instrumentation, we still need to distinguish the paths going through u_1 to n, $\Phi_{u_1}(s,n)$, and those going through u_2 to n, $\Phi_{u_2}(s,n)$. We assume $\Phi_{u_1}(s,n)$ and $\Phi_{u_2}(s,n)$ merge at node w, where w is the nearest ancestor of n and w may be s. If edges $E(u_1,n)$ and $E(u_2,n)$ are not recorded by instrumenting node n, I' should instrument other nodes to differentiate $\Phi_{u_1}(w,n)$ and $\Phi_{u_2}(w,n)$, e.g., instrumenting u_1 and u_2.

Case 1: *No other nodes instrumented within w and n in I'.* The edges with non-zero entropy should be recorded, indicated by $\sum_{u \in pred(n)} H(E(u,n)) > 1$; Otherwise, there must be paths in $\Phi(w,n)$ that become non-differentiable. Therefore, without selecting n, I' needs to instrument at least $\sum_{u \in pred(n)} H(E(u,n)) - 1$ nodes to differentiate these paths since they are not merging until the intermediate node w. Meanwhile, since $\sum_{u \in pred(n)} H(E(u,n)) > 1$, the number of nodes instrumented should not be fewer than one. Thus, the number of instrumented nodes in I is not fewer than solely instrumenting node n. Hence, $|I'| \geq |I|$, a contradiction. The same justification works for more predecessors exist.

Case 2: *At least one other node instrumented within w and n in I'.* We assume there is a nearest instrumented ancestor node to n. Without loss of generality, we assume u_1 is instrumented. Therefore, $|succ(u_1)| > 1$; Otherwise, we can either use u_1 or n to differentiate $\Phi_{u_1}(s,n)$ and $\Phi_{u_2}(s,n)$, which makes $|I'| = |I|$, a contradiction. Without loss of generality, we assume another successor node of u_1 as m.

Case 2.1: $|pred(u_1)| = 1$. u_1 can only differentiate $\Phi_{u_1}(w,n)$. In this case, even though u_1 is instrumented, I' still needs to differentiate paths in $\Phi(u_1,m)$ and $\Phi(u_1,n)$. Therefore, at least one more node within w and n require selected in I' as a correct instrumentation, which requires more nodes than solely selecting n. Thus, $|I'| > |I|$, a contradiction.

Case 2.2: $|pred(u_1)| > 1$. Based on definition, $\sum_{t \in pred(u_1)} H(E(t,u_1)) > 1$; Otherwise, according to Case 2.1, $|I'| > |I|$, a contradiction. With $|pred(u_1)| > 1$, for $\Phi(w,u_1)$, similar to Case 1, I' needs at least $\sum_{t \in pred(u_1)} H(E(t,u_1)) - 1$ nodes to differentiate these paths if u_1 is not selected. Meanwhile, regarding $\Phi_{u_2}(w,n)$, $\Phi(u_1,n)$, and $\Phi(u_1,m)$, I' needs to select another node within w and n to differentiate them. Therefore, for $\Phi(w,n)$ at node n, I' still needs at least $k = 1 + \sum_{t \in pred(u_1)} H(E(t,u_1)) - 1$ nodes for instrumentation, where $k \geq 2$. Hence, the number of the instrumented nodes of I' is not fewer than I, which solely instruments node u_1 and n, Thus, $|I'| \geq |I|$, a contradiction.

The same justification works for more instrumented predecessors exist. Hence, $\nexists I', |I'| < |I| \wedge I'$ is correct. I is the minimal and correct node-level instrumentation. □

Putting everything together, we have shown that our instrumentation algorithm is both correct and optimal in the node-centric view to achieve edge coverage tracing.

Remark 3. The generality of our formulation enables correct instrumentation of arbitrary programs, including those with loops and unstructured control flow, without assumptions on the given control-flow graph, a crucial advantage over existing edge-centric methods. Our correctness and optimality proofs further provide theoretical guarantees for approaching optimal edge-coverage instrumentation. Regarding runtime performance, selecting fewer nodes could, in principle, concentrate instrumentation on frequently executed blocks; however, this does not undermine overall performance in practice, for two reasons. First, by our correctness proof, any method must instrument a frequently executed block on a single path (e.g., a block within a loop) if it satisfies the differentiation condition; otherwise, paths with different loop iterations cannot be distinguished. Second, the optimality of our instrumentation guarantees that existing approaches execute at least as many instrumented blocks per path as ours. Together, these observations show that frequently

executed blocks unlikely increase our method's per-path overhead, and thus its overall overhead, relative to existing approaches. More broadly, the node-centric view opens the door to optimal solutions for the original edge coverage problem: the ability to identify a minimal instrumentation set of nodes lays a solid foundation for future work on recording even fewer edges.

4 Evaluation

In addition to the theoretical advancements, we are committed to demonstrating the practicality of our new algorithm. To this end, we have implemented our algorithm as InsOpt, an optimal instrumentation engine for edge coverage tracing. We have harnessed the LLVM [23] framework to carry out the analysis, which also facilitates seamless integration with the Clang compiler. Following the double-blind review process and upon acceptance of the paper, we will make the reproducible prototype publicly available. Based on the implementation, we design a series of evaluations to demonstrate the effectiveness of InsOpt by answering the following research questions:

- **RQ1**: Can InsOpt efficiently instrument fewer blocks to achieve edge coverage?
- **RQ2**: Can InsOpt effectively reduce runtime overhead?
- **RQ3**: Can InsOpt assist in other application scenarios using coverage tracing?

Our primary focus is to demonstrate the effectiveness of our algorithm with fewer instrumentation sites. Therefore, we use RQ1 to evaluate the number of instrumented basic blocks compared to state-of-the-art methods in Section 4.1. Additionally, we record the time costs of different instrumentation algorithms to demonstrate their efficiency in practice. Moreover, to illustrate the benefit brought by our optimal node-centric instrumentation, we propose RQ2 to assess the runtime speed of the instrumented binary using the same set of test cases in Section 4.2 to ensure fairness in our evaluation. Finally, to demonstrate whether our optimized instrumentation can enhance the performance of downstream applications to improve usability. Therefore, we design RQ3 and deploy our instrumentation in one of the most widely-used scenarios, fuzzing, as presented in Section 2.2.

Benchmark. As one of the most influential application scenarios for coverage tracing is testing, we use the state-of-the-art benchmark Magma [19] to help answer our research questions. Magma consists of various CVEs selected across nine benchmark real-world programs with complex control-flow structures, which are frequently evaluated in the state-of-the-art fuzzing, as shown in Table 1. For each project, we follow the instructions of the benchmark[2] to configure and adapt our instrumentation engine to existing fuzzers. All fuzzers have the same set of initial seeds provided by the benchmark. Note that not all bugs in the benchmark can be reproduced by the community [34]; we only report the bugs that can be found during our evaluation.

Baseline. Since InsOpt is built upon LLVM, we compare InsOpt with the state-of-the-art instrumentation algorithm used in the modern compiler, Clang. SanitizerCoverage [23] is the most widely-used edge coverage framework deployed in modern compilers, Clang and GCC, which is also the default algorithm used in the state-of-the-art fuzzing framework, AFL++ [15]. It has shown a lower runtime overhead against most state-of-the-art instrumentation algorithms [41]. This approach is also inspired by superblock methods optimized using domination relations [3]. Specifically, we use the configuration of `-fsanitize-coverage=inline-8bit-counters` mode in SanitizerCoverage to also record the execution frequency of each edge so that the coverage granularity is the same as other fuzzers, i.e., AFL++. We do not introduce any additional compilation flags beyond the default setting provided in the Magma benchmark. We would also like to compare the state-of-the-art approaches Casper [43], which reduces coverage tracing as a vertex cover problem, and Odin [41], which removes instrumentation on the fly. However, neither is open

[2]https://github.com/HexHive/magma/tree/v1.2/fuzzers

Table 1. Magma benchmark

Project	Size (CLOC)	Num. Blocks	Input format	Num. CVEs
libpng	95K	9,548	PNG	7
libsndfile	83K	20,498	Various	18
libtiff	95K	34,227	TIFF	14
libxml2	320K	98,692	XML	17
openssl	630K	170,167	Various	20
poppler	260K	128,735	PDF	22
sqlite3	387K	156,677	SQL	20
lua	31K	10,499	LUA	4
php	1.6M	605,041	Various	16

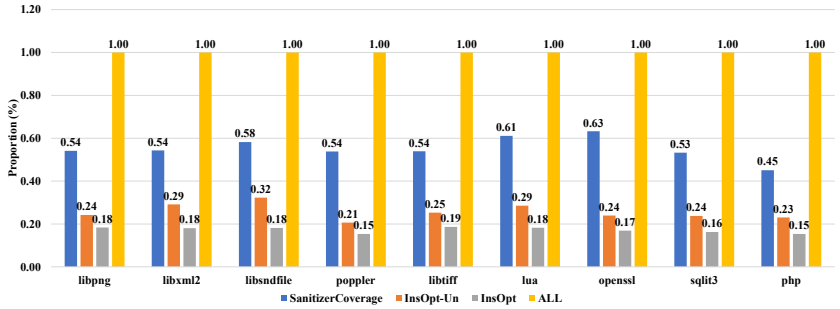


Fig. 5. Proportion of instrumented basic block in the evaluated projects using different strategies.

source, preventing direct comparison. Furthermore, Casper directly targets call traces rather than edge coverage and assumes LL(1) grammars. We have thus compared with Casper through the motivating example in Figure 1. Odin uses on-the-fly recompilation, which is orthogonal to our static approach. Therefore, we exclude them from our baseline. We also exclude those algorithms designed only for acyclic graphs, which cannot be applied to real-world programs, and most of which are not open-sourced.

Configuration. All experiments are conducted on a server equipped with an AMD Ryzen Threadripper 3990X 64-Core 2.9 GHz CPU and 256 GB of RAM, running Ubuntu 20.04.3 LTS. For the fuzzing experiment answering RQ3, we repeated each fuzzer ten times to minimize the influence brought by randomness. For each time, the experiment was run with a time budget of 24 hours, indicated by the benchmark [19]. We employ the Mann-Whitney U Test [26] to demonstrate the statistical significance of the contribution made by our framework. Some specific configuration details are mentioned in the related experiments, respectively.

4.1 Does InsOpt Efficiently Reduce the Instrumentation?

The primary goal of coverage tracing is to use minimum instrumentation while maintaining precision. Therefore, we first measure the instrumentation needed in each method to achieve edge coverage to answer RQ1. To gain a better understanding of our node-centric view, we also make a variant of InsOpt that only instruments all nodes with multiple predecessors named as InsOpt-Un, which helps demonstrate the necessity of our selection refinement.

Number of Instrumentation Needed. Figure 5 reports the results. In summary, both InsOpt and InsOpt-Un surpass existing efforts by requiring 3.2x and 2.1x fewer instrumented blocks, respectively, compared to the sanitizer coverage used in LLVM for achieving edge coverage. These notable comparisons highlight the substantial contribution of shifting from the edge-centric view

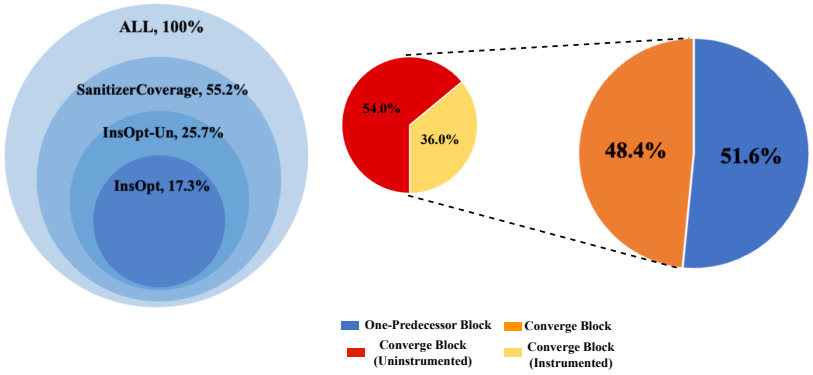


Fig. 6. The proportion of different types of blocks collected in all evaluated projects. The simplified block indicates the block with only one predecessor and one successor. A one-predecessor block indicates a block with one predecessor but multiple successors. The converge block indicates blocks with multiple predecessors.

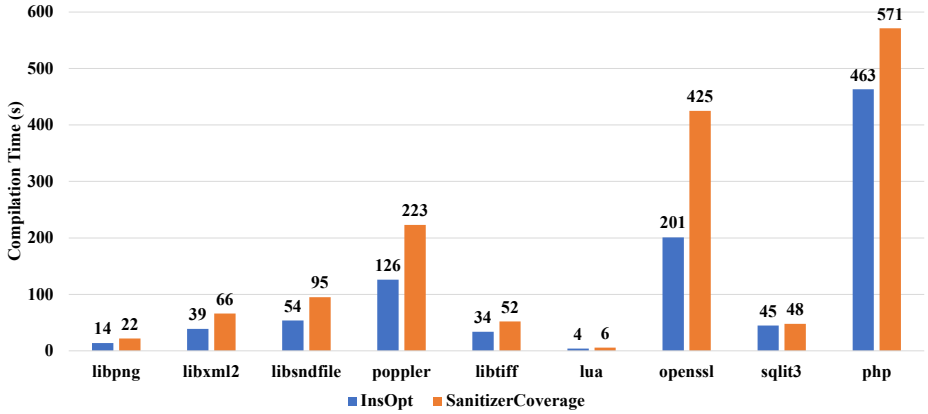


Fig. 7. The average CPU time cost of instrumenting each project while compiling.

to a node-centric view. Furthermore, InsOpt only necessitates instrumenting an average of 17.3% of basic blocks within the given programs, which is around 10% less than InsOpt-Un. Such a difference underscores the existence of redundant tracing and our effectiveness in eliminating it.

The improved performance of our algorithm is further elucidated through an analysis of the proportion of different types of basic blocks in the program. The average results, gathered from all projects in Magma, shown in Table 1, are presented in Figure 6. Notably, over half of the basic blocks (51.6%) fall into the category of *simplified blocks*, characterized by having only one predecessor and one successor in the program. The prevalence of simple blocks implies that a substantial number of edges do not significantly contribute to differentiating paths in the program. Fortunately, our *node-centric* coverage approach effectively identifies and handles these cases, leading to a reduction in the number of instrumented blocks. Moreover, our analysis results reveal the existence of redundant blocks, where the paths they track are subsumed by others. Approximately 54.0% of the blocks with multiple predecessors at path convergence points can be omitted from the instrumentation in InsOpt-Un, highlighting the algorithm's capacity to eliminate unnecessary instrumentations.

Table 2. The average time cost (s) of executing the same number of inputs with different instrumentation in each project. N_{exe} represents the number of inputs given to each project. *Native* is the execution time used by binary without any instrumentation.

Project	*Native*	InsOpt	InsOpt-Un	SanitizerCoverage	N_{exe}
libpng	87	96	103	110	37,916
libsndfile	114	183	245	293	57,782
libtiff	131	160	213	305	95,959
libxml2	496	536	659	816	205,021
openssl	751	1,477	1,783	2,032	131,799
poppler	145	209	238	290	29,445
sqlite3	154	166	192	230	62,231
lua	123	275	343	412	112,389
php	492	682	731	814	55,176
Avg.		**45.8%**	**75.8%**	**111.2%**	

Instrumentation Time Cost. In addition, we have assessed the time cost of the instrumentation algorithm. Given the tight correlation between coverage instrumentation and the compiler, we measure the overall compilation time for the entire evaluation project. This metric can demonstrate the compatibility of our implementation and help avoid bias introduced by specific compilation optimization configurations in a single component.

As depicted in Figure 7, InsOpt efficiently instruments all projects during compilation, taking just 6 CPU minutes on average, which is 1.6x faster than the sanitizer coverage used in Clang. This result underscores the efficiency proved by our linear complexity algorithm. Such swift compilation time makes InsOpt well-suited for deployment in scenarios involving dynamic rewriting or on-the-fly compiling techniques, facilitating rapid updates to the binary.

4.2 Does InsOpt Effectively Reduce the Runtime Overhead?

To showcase the efficiency gained from instrumentation reduction, we conduct an evaluation of the runtime overhead caused by different instrumentation strategies. To ensure a fair comparison, we utilize the AFL++ fuzzer to generate an equal number of test cases for each project under test. We then measure the time required to execute these inputs using different instrumented binaries.

In Table 2, we present results averaged across 10 runs conducted with various instrumentation algorithms. Overall, InsOpt incurs only 45.8% runtime overhead compared to native execution. In some cases, such as `libpng` and `sqlite3`, InsOpt achieves nearly the same throughput as native execution with less than 10% runtime overhead. When compared to sanitizer coverage, InsOpt exhibits 2.4x less runtime overhead, accompanied by a 1.4x speedup. This substantial contrast underscores the significant improvements InsOpt has brought to coverage tracing through its shift to a node-centric view. With fewer nodes instrumented for edge tracing during the execution, the runtime overhead is thus reduced. The stable improvements also empirically support that our instrumentation does not introduce additional overhead on those reduced but frequently executed nodes. Furthermore, InsOpt outperforms InsOpt-Un with 30.0% less runtime overhead and 1.2x speedup, demonstrating the effectiveness of our refined approach for coverage tracing.

4.3 Is InsOpt Capable of Assisting Other Application Scenarios?

To demonstrate the versatility of InsOpt, we deploy it to the downstream application scenarios to examine its practical influence. In our evaluation, we select fuzz testing as our application

Table 3. Triggering time for each target vulnerability in the Magma benchmark. *T* indicates the reproduction time (second) averaged over ten runs. *T.O.* indicates the fuzzer cannot reproduce the targets within the given time budget, 24 hours. *Ratio* and *p* indicates the improvement ratio and *p*-value compared with InsOpt. We ignore the comparison against targets that the compared fuzzers have not found. We also list the number of bugs that can be detected against InsOpt next to the name of each fuzzer.

Bug ID	InsOpt	AFL++ (32/36)			PGE (26/36)			Instrim (17/36)		
	T	*T*	*Ratio*	*p*	*T*	*Ratio*	*p*	*T*	*Ratio*	*p*
PNG003	**26**	38	1.5x	0.04	**26**	1.0x	-	30	1.2x	0.03
PNG006	**37**	1118	30.2x	<0.01	*T.O.*	-	-	*T.O.*	-	-
PNG007	**16,211**	*T.O.*	-	-	35,512	2.2x	-	*T.O.*	-	-
SND001	**203**	680	3.3x	<0.01	693	3.4x	0.01	*T.O.*	-	-
SND005	2,078	8791	4.2x	<0.01	**837**	0.4x	-	3,601	1.7x	<0.01
SND006	**2,432**	9,896	4.1x	<0.01	16,399	6.7x	0.01	*T.O.*	-	-
SND007	**1,320**	3,316	2.5x	<0.01	2,019	1.5x	0.01	*T.O.*	-	-
SND017	2,871	3,442	1.2x	<0.01	867	0.3x	-	**741**	0.3x	-
SND020	**1,720**	8,112	4.7x	<0.01	1,806	1.1x	0.01	*T.O.*	-	-
SND024	**740**	7,113	9.6x	<0.01	776	1.0x	0.02	*T.O.*	-	-
TIF005	**16,273**	31,549	1.9x	<0.01	*T.O.*	-	-	*T.O.*	-	-
TIF006	**2,728**	21,899	8.0x	<0.01	*T.O.*	-	-	*T.O.*	-	-
TIF007	**81**	318	3.9x	0.01	132	1.6x	0.01	5,409	66.8x	<0.01
TIF009	**7,479**	38522	5.2x	-	33,246	4.4x	<0.01	45,945	6.1x	<0.01
TIF012	**1,272**	9,094	7.1x	<0.01	1,598	1.3x	-	15,840	12.5x	<0.01
TIF014	**2,272**	11,241	4.9x	<0.01	*T.O.*	-	-	43,816	19.3x	<0.01
XML001	**10,452**	24,147	2.3x	<0.01	*T.O.*	-	-	*T.O.*	-	-
XML003	**13,224**	41,795	3.2x	<0.01	21,981	1.7x	<0.01	*T.O.*	-	-
XML009	**2,332**	3,506	1.5x	-	3,066	1.3x	-	*T.O.*	-	-
XML017	**42**	59	1.4x	0.04	75	1.8x	0.03	80	1.9x	0.01
SSL001	19,126	47,206	2.5x	<0.01	**16,119**	0.8x	-	*T.O.*	-	-
SSL002	121	213	1.8x	0.01	233	1.9x	0.01	**97**	0.8x	-
SSL003	**107**	317	3.0x	0.03	245	2.3x	0.01	130	1.2x	0.04
PDF003	**24,574**	38,260	1.6x	0.02	*T.O.*	-	-	*T.O.*	-	-
PDF010	3,237	5,271	1.6x	0.02	6,020	1.9x	<0.01	**2,638**	0.8x	-
PDF016	**51**	99	1.9x	0.03	139	2.7x	<0.01	84	1.6x	0.01
PDF018	**9,971**	42,010	4.2x	<0.01	51,623	5.2x	<0.01	*T.O.*	-	-
PDF021	**59,159**	*T.O.*	-	-	*T.O.*	-	-	*T.O.*	-	-
SQL002	**4,002**	4,411	1.1x	-	4,083	1.0x	0.03	*T.O.*	-	-
SQL012	**85,136**	*T.O.*	-	-	*T.O.*	-	-	*T.O.*	-	-
SQL014	23,314	**21,405**	0.9x	-	39,390	1.7x	<0.01	*T.O.*	-	-
SQL018	**7,514**	18,832	2.5x	<0.01	12,540	1.7x	0.01	74,737	9.9x	<0.01
LUA004	**15,201**	24,208	1.6x	0.01	*T.O.*	-	-	54,472	3.6x	<0.01
PHP004	**50,543**	*T.O.*	-	-	*T.O.*	-	-	79,691	1.6x	<0.01
PHP009	**5,564**	12,168	2.2x	<0.01	6,603	1.2x	0.01	31,271	5.6x	<0.01
PHP011	**372**	7,337	19.7x	<0.01	738	2.0x	<0.01	851	2.3x	<0.01
Avg.		**>4.5x**			**>2.0x**			**>8.1x**		

since it is one of the most widely-used techniques in modern software development [24, 35]. As fuzzing depends on instrumentation to collect coverage feedback for intermediate guidance from a large amount of generated input, runtime overhead becomes a significant bottleneck for its performance [25, 41]. Therefore, our goal is to examine whether our optimized instrumentation can

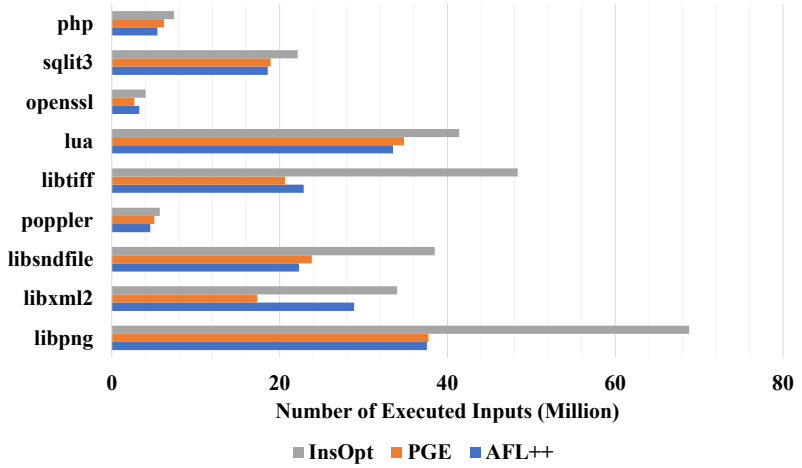


Fig. 8. The average number of inputs executed during the fuzzing evaluation in each project.

enhance the fuzzing performance and, eventually, improve its speed for detecting vulnerabilities. Specifically, we integrate InsOpt with the state-of-the-art coverage-guided fuzz testing framework, AFL++ [15], to demonstrate its ability to enhance testing performance, particularly in terms of vulnerability detection speed within the programs under test.

To demonstrate our approach's practicality, we conduct our evaluation on the state-of-the-art fuzz testing benchmark, Magma. Additionally, we employ two other state-of-the-art fuzzers, PGE [25] and Instrim [20], that are also known for minimizing the runtime overhead with reduced instrumentation. More discussion related to their technical details can be found in Section 5.1. It is worth noting that while such fuzzer frameworks are optimized for efficiency, most of the existing efforts devoted to this purpose in fuzzing sacrifice the precision of coverage tracing.

Bug Detection Speed. In Table 3, we present our results across all targets identified by the evaluated fuzzers. We report the time required to detect all bugs found by these fuzzers.

In summary, InsOpt stands out by necessitating 4.9x less time to detect the target vulnerabilities than the state-of-the-art fuzzers, demonstrating the effectiveness of reducing runtime overhead with our node-centric instrumentation algorithm. Furthermore, InsOpt identifies all the targets detected by other fuzzers and uncovers an additional 11 vulnerabilities on average. Notably, in comparison to state-of-the-art fuzzers such as AFL++, PGE, and Instrim, InsOpt showcases significant improvement, triggering target vulnerabilities 4.5x, 2.0x, and 8.1x faster on average and discovering 4, 10, and 19 more targets, respectively. These figures underscore the effectiveness of our design in optimizing the instrumentation algorithm with fewer instrumented blocks to record edge coverage.

Moreover, it is still noticed that even though existing efforts can detect some cases faster than InsOpt, e.g., `SND005` and `SSL001`, their performance is unstable, and they fail to find some target vulnerabilities within the evaluation time. Such fluctuating performance indicates the necessity of designing precise instrumentation for effective feedback for other applications. Meanwhile, since execution time serves as one important metric for input generation in fuzz testing [1, 33], the straightforward integration of InsOpt with AFL++ may negatively influence performance, which causes the baseline AFL++ to outperform InsOpt in some cases, e.g., `SND006`. Such orthogonal problems can be studied in future work to achieve better integration of the techniques. Still, the overall performance improvement with statistical evidence (all $p-$values smaller than 0.05) demonstrates the effectiveness of our instrumentation in precisely recording the edge coverage.

Table 4. Bugs finding capability using different fuzzers. We list the number of bugs that can be found by each fuzzer within three days of evaluation.

Project	InsOpt	AFL++	PGE	Instrim
libsndfile	1	1	1	0
binutils	1	0	0	0
gpac	3	1	1	1
sqlite3	1	0	0	0
php	1	0	0	0

Runtime Overhead. To further assess whether the benefits of efficient bug finding indeed come from our optimized instrumentation, we examine the execution throughput of the fuzzing engine in comparison to the baseline, AFL++, and the fuzzer built upon AFL++, PGE. The results are presented in Figure 8. On average, InsOpt delivers a 1.5x speedup across the examined projects in Magma. Notably, in contrast to the unstable performance of PGE, which is influenced by the additional time cost of determining whether the input needs execution based on partial instrumentation, InsOpt demonstrates a consistently improved performance in executing more inputs.

New Bug Finding. We also evaluate the number of new bugs that can be discovered using different instrumentation strategies, as it is one of the primary purposes of fuzzing. Specifically, we employ the evaluated fuzzers on the latest versions of projects routinely examined by the industrial-scale fuzzer, OSS-Fuzz [2], to assess their capability to detect new vulnerabilities. The projects chosen are also frequently evaluated by the state-of-the-art fuzzing frameworks [20, 25, 28, 42]. To ensure fairness, we allocate the same time budget (three days) for each fuzzer to detect these fresh vulnerabilities.

We list the number of bugs found by each fuzzer in Table 4. All bugs are confirmed and patched by the developers. It is vital to notice that every version of these projects has been routinely examined by OSS-Fuzz [2]. Still, InsOpt can detect four more bugs in the evaluated projects than other state-of-the-art fuzzers. Such dramatic time improvements highlight the substantial gains from our node-centric optimal instrumentation. Due to the double-blind requirement, we will release the link to the bug reports once the paper is accepted, as it involves personal information.

To further assess effectiveness, we extend the evaluation period for other fuzzers to 7 days. Surprisingly, they are still unable to detect these issues. We also sought to understand the potential reasons for this improvement, since, in theory, performance enhancements should not significantly affect the results of the fuzzer in detecting the same bugs. We investigate the reasons that can be due to execution-time factors in path scheduling strategies, which makes the fuzzer not prioritize or completely ignore paths with vulnerabilities but high execution time. Interestingly, we found that most state-of-the-art fuzzers use the execution time of each input as a prioritization metric for seed selection. As a result, the optimized instrumentation can indeed impact the prioritization of inputs selected by the fuzzer, directing it to explore different parts of the programs.

This phenomenon also aligns with the conventional conclusion that improvements in throughput can yield cumulative gains in fuzzing, as observed in existing efforts [28, 41]. The throughput gains from InsOpt can improve the input-generation policy, i.e., path scheduling, by reducing bias from additional runtime instrumentation overhead and enabling better use of fine-grained runtime feedback to improve effectiveness. This combination can have a greater impact on bug detection than a sole throughput improvement. Nevertheless, as fuzzing is a complex, non-deterministic process, overall performance may vary, as our method can introduce side effects by deprioritizing bug-relevant seeds that require necessary logics from those unselected nodes. Such issues point to a

```
1 bool display_file(FILE* fd) {
2   char* buff;
3   int length, index;
4
5   load_table(fd, buff, &length);
6
7   // processing the header
8   while(index < length
9     && index <= MAX_LENGTH) {
10    char* header = load_header_data(buff, index);
11    switch(header) {
12       // data processing for different cases
13    }
14    index++;
15  }
16
17  if(...) {
18    // The bug occurs when the header is malformed
19    // so that the last index can exceed the
20    // maximum length, causing overflow here.
21    display(buff[index]);
22  }
23 }
```

Fig. 9. Case study of bugs found uniquely by InsOpt. This overflow requires visiting a long, but malformed, header buffer, causing the index to mistakenly exceed the buffer boundary at Line 21. However, the high overhead introduced by such long paths is not prioritized by existing fuzzers, leading to false negatives in bug detection.

promising future direction for adapting optimized instrumentation to diverse application scenarios, i.e., fuzz testing. Since this is an orthogonal problem to the one addressed in this paper, we plan to explore it in future work. We can examine the possibility of leveraging the reduced instrumentation to optimize the original fuzzing methodology.

Case study. To understand why our instrumentation helps fuzzers find more bugs, we examine a bug in Binutils uniquely found by InsOpt. As shown in Figure 9, triggering this overflow requires traversing a long but malformed header buffer, causing the index to exceed the buffer boundary at Line 21. Existing fuzzers, however, deprioritize such time-consuming paths once the loop has been covered, because they instrument most nodes involved in processing the different cases. In contrast, our node-centric approach instruments only one node after the processing of each header, which reduces the execution time along this path and, in turn, raises the priority of seeds containing larger headers. As a result, InsOpt finds this bug while existing tools do not. This case highlights the necessity of reducing coverage-instrumentation overhead and suggests that downstream applications relying on runtime feedback should also account for the bias introduced by instrumentation.

4.4 Discussion

After demonstrating the effectiveness of InsOpt with a thorough evaluation, We would also like to demonstrate the potential of future work through our node-centric instrumentation.

Control-flow-based Compiler Optimization. To meet the demands of modern software and architectures, compilers need to incorporate an ever larger number of increasingly complex program structures with diverse transformation algorithms [38, 39]. However, optimizing code transformation remains challenging, as these transformations can sometimes degrade performance or interfere with subsequent optimizations. As one of the most effective categories of optimization methods, control-flow-based optimization rules, e.g., `simplifycfg` and `instcombine`, have exhibited great potential in improving execution speed and reducing the size of compiled binaries [5]. Our node-centric view introduces a new perspective to refine existing optimization efforts by redefining

what constitutes a "node". For example, we can optimize the pass, `simplifycfg`, to merge more blocks since their dependency is equivalent to edges in the control-flow graph. Moreover, we can define a node as a function that finds additional potential functions to inline together through the call graph, enabling more effective inlining and facilitating application scenarios such as dynamic debugging, performance profiling, and compiler optimization.

Compositional Program Analysis. As program sizes and complexities proliferate, the imperative for practical compositional analysis becomes increasingly apparent for enhanced scalability. A key challenge in compositional program analysis lies in accurately determining the locations within the program where slicing can occur. Despite substantial efforts in this realm, our node-centric view introduces a novel understanding of both where and why the program can be sliced. In this paper, we narrow our focus to edge coverage as the chosen semantics for determining the placement of instrumentation, ensuring that coverage information is retained. This choice is rooted in the ability to differentiate edges between instrumented blocks precisely, marking them as optimal locations for slicing the program to trace coverage semantics. Inspired by our node-centric view, we can redefine the convergence points of semantics in other scenarios, allowing us to specify where these semantics should be recorded to support compositional analysis. This approach ensures minimal precision loss and maximizes parallelity, addressing a critical aspect of the challenge of determining suitable locations for program slicing in the context of compositional analysis.

5 Related Work

Coverage tracing is a fundamental component for many application scenarios. In this section, we would like to discuss their potentials and differences compared with InsOpt. Specifically, we discuss about optimized instrumentation in fuzz testing (Section 5.1) and profiling (Section 5.2).

5.1 Optimized Coverage Feedback for Coverage-guided Fuzzing

Fuzzing stands out as one of the most effective methods for detecting vulnerabilities today, with coverage serving as a crucial feedback mechanism to guide the fuzzer in thoroughly examining the program under test. Consequently, low runtime overhead becomes a critical factor for scalability. Efforts to address these challenges can be broadly categorized into two main approaches:

Statically Removing Unnecessary Instrumentation From the Program. This category of approaches shares the underlying intuition that edges can be distinguished from others based on specific program dependencies, such as control dependency. However, none of the existing methods can provide a guarantee as InsOpt does for both correctness and optimality. For instance, Instrim [20] employs a minimized set of edges, reduced by domination and reachability, to recover precise coverage information. Still, it faces limitations in distinguishing paths within loops, resulting in a sacrifice in precision and an omission of the analysis of any loop structure within the program under test. Additionally, its approximation algorithm introduces errors in indexing due to imprecise edge recording. In response, AFL++ [15] adapts the design of Instrim and implementation from LLVM [23] with a more precise edge indexing to ensure accuracy. Despite these efforts, all these methods rely on specific simplifying assumptions without a systematic view due to the scalability challenges in identifying paths within the complex program structure. Consequently, they struggle to formalize the problem and provide theoretical analyses for their results, a deficiency addressed by InsOpt through shifting to the unique perspective of a node-centric view.

Dynamically Adjusting the Instrumentation for Better Efficiency. This line of approaches typically shares the intuition of trading off a certain level of precision in coverage tracing for improved efficiency. For example, Untracer [28] first suggests removing instrumentation once it is covered. However, this aggressive approach may result in a loss of valuable feedback, leading to

an overall decline in performance [42]. Therefore, subsequent work in this domain adopts more nuanced strategies by incorporating diverse granularity levels of coverage feedback. For instance, Zeror [42] first utilizes three different granularities of coverage with individual binaries to minimize runtime overhead. Only executions found interesting according to coarse-grained metrics are then sent to binaries with fine-grained instrumentation to collect precise feedback. The challenge in this context lies in establishing standards for transitioning between fine-grained and coarse-grained metrics to optimize effectiveness. PGE [25] tackles this challenge by proposing the use of path prefix similarity to predict whether the input under execution is worth coverage tracing; otherwise, the entire tracing process is removed. Even though this line of approaches indeed improves the efficiency, their results can vary due to some precise feedback failing to collect, as shown in our evaluation compared to InsOpt or even AFL++. Nevertheless, there exists an opportunity to explore the potential to integrate InsOpt with this line of approaches to achieve better outcomes, which can redefine what kind of instrumentation can be safely removed.

5.2 Reduced Edge Recording for Efficient Path Profiling

Profiling is a succinct and effective method to obtain program behavior through inserting additional code and executing the modified program [6–9, 12, 13, 18]. As its result directly comes from concretely executing the program, profiling can provide precise feedback for debugging, optimization, and testing. Similar to other dynamic analysis scenarios, profiling grapples with the challenge of balancing runtime overhead to achieve optimal precision and scalability. This challenge has spurred efforts in two main directions: Minimizing the number of locations for instrumentation and reducing the information that needs to be recorded.

Minimizing the Number of Locations for Instrumentation. The connection between graph theory and the control-flow graphs of programs inspires this line of methods. Early efforts [18] explored leveraging the minimum spanning tree feature to identify a minimized set of edges for instrumentation. However, the spanning tree assumption proves overly simplistic for programs, as control-flow graphs involve more intricate properties such as edge direction and cycles. Therefore, various algorithms have emerged to address these complex graph features present in programs, including considerations for edge directions [7] and diamond cycles [8]. Despite the empirical success of these efforts in enhancing performance, identifying the optimal edge set for edge coverage profiling remains an NP-hard problem [8, 43] due to the inherent complexity of program structures.

In contrast to explicitly seeking the optimal instrumentation for edges, we propose an alternative approach by shifting to a node-centric view for edge/path coverage. Through this perspective, we have successfully demonstrated that our algorithm can handle arbitrary program structures, providing correct and optimal results in node-centric settings. The empirical evidence from our evaluation further underscores the significance of this approach in terms of performance and generality. For future work, it is possible to extend InsOpt with adapted recovery algorithms to support diverse profiling purposes.

Reducing the Information Needs to be Recorded. As contemporary program analysis techniques serve diverse purposes, various semantics have been proposed to be collected from program executions. In the context of edge/path coverage tracing addressed by this paper, the primary intuition of existing efforts is to devise optimized indexing mechanisms to minimize the instrumentation required for distinguishing edges. For example, bit tracing [8] first proposed the concept of *edge weightening* to assign a one-bit value index with each edge to reduce the memory usage of instrumentation. SPP [4] designs a more sophisticated algorithm to properly design the values that should be distributed for each edge. Meanwhile, optimized structures for efficiently identifying the values are also proposed. For example, Riff [40] leverages the vector processing abilities of modern

processors to support identifying coverage through a vectorized scan. Casper [43] designs a logged model based on LL(1) grammars to record the edges require instrumentation.

Despite these remarkable successes, it is crucial to note that these methods, while orthogonal to the primary focus of this paper on finding optimal instrumentation locations, can be integrated with InsOpt to achieve better performance in future work. This possibility suggests a potential avenue for further exploration and improvement in the field.

6 Conclusion

In this paper, we have proposed a novel perspective, the *node-centric* view, to achieve edge coverage from the conventional edge-centric view with an optimal set of instrumented basic blocks. Indeed, we have devised a linear-time algorithm and formally established its correctness and optimality (i.e., using the minimum number of instrumented blocks for coverage tracing without precision loss). To showcase the practical utility of our algorithm, we have implemented it as the InsOpt tool and provided significant empirical evidence: we require 3.2x less instrumentation while achieving 2.4x less runtime overhead. Furthermore, we have shown the strong generality of our work as the integration of InsOpt and fuzzing leads to a 4.9x speedup in bug finding and detects 3.5x more previously unseen bugs than the state-of-the-art fuzzers on the benchmark. This significant efficiency improvement has also enabled InsOpt to detect five previously unknown vulnerabilities in extensively tested projects by state-of-the-art efforts from both academia and industry.

Data-Availability Statement

The software and benchmarks that support the evaluation results are available on Github.

7 Funding and Acknowledgment

We thank the anonymous reviewers and the shepherd for their valuable feedback. The authors are supported, in part, by Hong Kong Research Grant Council under Grant No. RGC21201925 and National Natural Science Foundation of China under Grant No. 62502403. Heqing Huang is the corresponding author.

References

[1] 2013. AFL: american fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed: 2013.

[2] 2018. OSS-FUZZ report. https://security.googleblog.com/2018/11/a-new-chapter-for-oss-fuzz.html. Accessed: 2018-11-06.

[3] Hiralal Agrawal. 1994. Dominators, super blocks, and program coverage. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Portland, Oregon, USA) *(POPL '94)*. Association for Computing Machinery, New York, NY, USA, 25–34. doi:10.1145/174675.175935

[4] Taweesup Apiwattanapong and Mary Jean Harrold. 2002. Selective Path Profiling. In *Proceedings of the 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (Charleston, South Carolina, USA) *(PASTE '02)*. Association for Computing Machinery, New York, NY, USA, 35–42. doi:10.1145/586094.586104

[5] D.I. August, W.W. Hwu, and S.A. Mahlke. 1997. A framework for balancing control flow and predication. In *Proceedings of 30th Annual International Symposium on Microarchitecture*. 92–103. doi:10.1109/MICRO.1997.645801

[6] Thoms Ball. 1999. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes* 24, 6 (Oct. 1999), 216–234. doi:10.1145/318774.318944

[7] T. Ball and J.R. Larus. 1996. Efficient path profiling. In *Proceedings of the 29th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO 29*. 46–57. doi:10.1109/MICRO.1996.566449

[8] Thomas Ball and James R. Larus. 1994. Optimally Profiling and Tracing Programs. *ACM Trans. Program. Lang. Syst.* 16, 4 (jul 1994), 1319–1360. doi:10.1145/183432.183527

[9] Thomas Ball, Peter Mataga, and Mooly Sagiv. 1998. Edge Profiling versus Path Profiling: The Showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '98)*. Association for Computing Machinery, New York, NY, USA, 134–148. doi:10.1145/268946.268958

[10] Michael D. Bond and Kathryn S. McKinley. 2005. Continuous Path and Edge Profiling. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture* (Barcelona, Spain) *(MICRO 38)*. IEEE Computer Society, USA, 130–140. doi:10.1109/MICRO.2005.16

[11] Dehao Chen, David Xinliang Li, and Tipp Moseley. 2016. AutoFDO: automatic feedback-directed optimization for warehouse-scale applications. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization* (Barcelona, Spain) *(CGO '16)*. Association for Computing Machinery, New York, NY, USA, 12–23. doi:10.1145/2854038.2854044

[12] Trishul M. Chilimbi, Ben Liblit, Krishna Mehra, Aditya V. Nori, and Kapil Vaswani. 2009. HOLMES: Effective statistical debugging via efficient path profiling. In *2009 IEEE 31st International Conference on Software Engineering*. 34–44. doi:10.1109/ICSE.2009.5070506

[13] Trishul M. Chilimbi, Aditya V. Nori, and Kapil Vaswani. 2007. Quantifying the Effectiveness of Testing via Efficient Residual Path Profiling. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering* (Dubrovnik, Croatia) *(ESEC-FSE '07)*. Association for Computing Machinery, New York, NY, USA, 545–548. doi:10.1145/1287624.1287706

[14] Irit Dinur and Samuel Safra. 2005. On the hardness of approximating vertex cover. *Annals of Mathematics* 162 (07 2005), 439–485. doi:10.4007/annals.2005.162.439

[15] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[16] H.N. Gabow, S.N. Maheshwari, and L.J. Osterweil. 1976. On Two Problems in the Generation of Program Test Paths. *IEEE Transactions on Software Engineering* SE-2, 3 (1976), 227–231. doi:10.1109/TSE.1976.233819

[17] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. 2018. CollAFL: Path Sensitive Fuzzing. In *2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA*. 679–696. doi:10.1109/SP.2018.00040

[18] Aaron J Goldberg. 1991. *Reducing overhead in counter-based execution profiling*. Computer Systems Laboratory, Stanford University. https://oac.cdlib.org/findaid/ark:/13030/c8c53sh1

[19] Ahmad Hazimeh, Adrian Herrera, and Mathias Payer. 2020. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.* 4, 3, Article 49 (Dec. 2020), 29 pages. doi:10.1145/3428334

[20] Chin-Chia Hsu, Che-Yu Wu, Hsu-Chun Hsiao, and Shih-Kun Huang. 2018. INSTRIM: Lightweight Instrumentation for Coverage-guided Fuzzing. In *Symposium on Network and Distributed System Security (NDSS), Workshop on Binary Analysis Research*. doi:10.14722/bar.2018.23014

[21] H. Huang, P. Yao, R. Wu, Q. Shi, and C. Zhang. 2020. Pangolin: Incremental Hybrid Fuzzing with Polyhedral Path Abstraction. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 1144–1158. doi:10.1109/SP40000.2020.00063

[22] James R. Larus. 1999. Whole program paths. *SIGPLAN Not.* 34, 5 (may 1999), 259–269. doi:10.1145/301631.301678

[23] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. https://dl.acm.org/doi/10.5555/977395.977673

[24] Jun Li, Bodong Zhao, and Chao Zhang. 2018. Fuzzing: a survey. *Cybersecurity* 1, 1 (2018), 1–13. doi:10.1186/s42400-018-0002-y

[25] Shaohua Li and Zhendong Su. 2023. Accelerating Fuzzing through Prefix-Guided Execution. *Proc. ACM Program. Lang.* 7, OOPSLA1, Article 75 (apr 2023), 27 pages. doi:10.1145/3586027

[26] Patrick E. McKnight and Julius Najab. 2010. *Mann-Whitney U Test*. American Cancer Society, 1–1. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/9780470479216.corpsy0524 doi:10.1002/9780470479216.corpsy0524

[27] Carla Michini, Peter Ohmann, Ben Liblit, and Jeff Linderoth. 2024. A Set-Covering Approach to Customized Coverage Instrumentation. *INFORMS Journal on Computing* 36, 1 (2024), 21–38. doi:10.1287/ijoc.2021.0349

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

[29] Stefan Nagy, Anh Nguyen-Tuong, Jason D. Hiser, Jack W. Davidson, and Matthew Hicks. 2021. Same Coverage, Less Bloat: Accelerating Binary-Only Fuzzing with Coverage-Preserving Coverage-Guided Tracing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event, Republic of Korea) *(CCS '21)*. Association for Computing Machinery, New York, NY, USA, 351–365. doi:10.1145/3460120.3484787

[30] Peter Ohmann, David Bingham Brown, Naveen Neelakandan, Jeff Linderoth, and Ben Liblit. 2016. Optimizing customized program coverage. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 27–38.

doi:10.1145/2970276.2970351

[31] Guilherme Ottoni and Bertrand Maher. 2017. Optimizing function placement for large-scale data-center applications. In *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 233–244. doi:10.1109/CGO.2017.7863743

[32] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. 2019. BOLT: A Practical Binary Optimizer for Data Centers and Beyond. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 2–14. doi:10.5555/3314872.3314876

[33] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. 2017. SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. ACM, New York, NY, USA, 2155–2168. doi:10.1145/3133956.3134073

[34] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. 2024. SoK: Prudent Evaluation Practices for Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. 1974–1993. doi:10.1109/SP54263.2024.00137

[35] Ari Takanen, Jared D Demott, Charles Miller, and Atte Kettunen. 2018. *Fuzzing for software security testing and quality arance*. Artech House. https://dl.acm.org/doi/10.5555/3275309

[36] Mustafa M Tikir and Jeffrey K Hollingsworth. 2002. Efficient instrumentation for code coverage testing. *ACM SIGSOFT Software Engineering Notes* 27, 4 (2002), 86–96. doi:10.1145/566171.566186

[37] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 2005. Efficient online computation of statement coverage. *J. Syst. Softw.* 78, 2 (nov 2005), 146–165. doi:10.1016/j.jss.2004.12.021

[38] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary Hall, and Jeffrey K. Hollingsworth. 2009. A scalable auto-tuning framework for compiler optimization. In *2009 IEEE International Symposium on Parallel and Distributed Processing*. 1–12. doi:10.1109/IPDPS.2009.5161054

[39] S. Triantafyllis, M. Vachharajani, N. Vachharajani, and D.I. August. 2003. Compiler optimization-space exploration. In *International Symposium on Code Generation and Optimization, 2003. CGO 2003.* 204–215. doi:10.1109/CGO.2003.1191546

[40] Mingzhe Wang, Jie Liang, Chijin Zhou, Yu Jiang, Rui Wang, Chengnian Sun, and Jiaguang Sun. 2021. RIFF: Reduced Instruction Footprint for Coverage-Guided Fuzzing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 147–159. https://www.usenix.org/conference/atc21/presentation/wang-mingzhe

[41] Mingzhe Wang, Jie Liang, Chijin Zhou, Zhiyong Wu, Xinyi Xu, and Yu Jiang. 2022. Odin: on-demand instrumentation with on-the-fly recompilation. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 1010–1024. doi:10.1145/3519939.3523428

[42] Renzhi Wu, Sanya Chaba, Saurabh Sawlani, Xu Chu, and Saravanan Thirumuruganathan. 2020. ZeroER: Entity Resolution using Zero Labeled Examples. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data* (Portland, OR, USA) *(SIGMOD '20)*. Association for Computing Machinery, New York, NY, USA, 1149–1164. doi:10.1145/3318464.3389743

[43] Rongxin Wu, Xiao Xiao, Shing-Chi Cheung, Hongyu Zhang, and Charles Zhang. 2016. Casper: An Efficient Approach to Call Trace Collection. *SIGPLAN Not.* 51, 1 (jan 2016), 678–690. doi:10.1145/2914770.2837619

[44] Meng Xu, Sanidhya Kashyap, Hanqing Zhao, and Taesoo Kim. 2020. Krace: Data Race Fuzzing for Kernel File Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1643–1660. doi:10.1109/SP40000.2020.00078

Received 2026-03-17; accepted 2026-08-06