*Article*

# TriFuzz: Probabilistic Distance-Guided Hybrid Directed Fuzzing with Selective Symbolic Instrumentation

**Yufeng Li [1,2,3], Yiwei Wang [1,3], Runhan Feng [2], Jiangtao Li [1,3] and Wutao Qin [2,*]**

1 School of Computer Engineering and Science, Shanghai University, Shanghai 200444, China; liyufeng_shu@shu.edu.cn (Y.L.); raidenzwyw@shu.edu.cn (Y.W.); lijiangtao@shu.edu.cn (J.L.)
2 Purple Mountain Laboratories, Nanjing 211111, China; fengrunhan@pmlabs.com.cn
3 Shanghai Key Laboratory of Intelligent Connected Vehicle Cybersecurity, Shanghai University, Shanghai 200444, China
* Correspondence: qinwutao@pmlabs.com.cn

**Abstract**

As software systems continue to grow in scale and complexity, fuzzing has become an indispensable automated technique for vulnerability discovery. Compared with coverage-guided fuzzing, directed greybox fuzzing (DGF) focuses execution toward specific basic blocks or functions, making it widely used in scenarios such as patch testing and vulnerability reproduction. Recent studies have combined fuzzing with symbolic execution (SE) to generate inputs that are difficult to obtain through mutation alone. However, applying SE to all branch conditions along an execution path may explore many paths unrelated to the target, leading to substantial overhead in directed fuzzing. Meanwhile, existing distance metrics still have limitations in guiding seeds toward targets: AFLGo relies on structural control-flow distances, which may not precisely reflect target reachability, while existing probability-based metrics often simplify complex control-flow structures such as loops and back-edges. To address these limitations, we propose TriFuzz, a probabilistic distance-guided hybrid directed fuzzing framework that integrates a loop-aware reachability distance model, target-related selective symbolic instrumentation, and a tightly coupled AFLGo–SymCC coordination mechanism. TriFuzz uses the probability-based distance model as the primary guidance signal and applies selective symbolic instrumentation to prune irrelevant basic blocks and concentrate exploration on target-relevant code regions. Our evaluation on the AFLGo testsuite and UniBench shows that TriFuzz improves both time-to-target and time-to-exposure on most evaluated benchmarks, demonstrating the effectiveness of combining fine-grained probabilistic distance guidance with selective symbolic reasoning and tightly integrated hybrid execution.

**Keywords:** directed fuzzing; symbolic execution; distance calculation

Academic Editor: George Angelos Papadopoulos

Received: 10 April 2026
Revised: 4 May 2026
Accepted: 5 May 2026
Published: 11 May 2026

**Copyright:** © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license.

## 1. Introduction

Modern software systems continue to grow in scale and complexity, resulting in an increasing number of subtle and security-critical vulnerabilities. Fuzzing has emerged as one of the most effective automated techniques for vulnerability discovery due to its scalability and minimal reliance on program specifications. In many practical scenarios, such as patch testing, vulnerability reproduction, crash triage, and exploit hardening, analysts are required to efficiently reach and exercise specific code regions rather than maximizing overall coverage. This demand has driven significant interest in directed greybox fuzzing (DGF) [1–15], which extends traditional coverage-guided fuzzing [16,17]

by steering executions toward user-specified locations. Prior studies have demonstrated the practical effectiveness of AFLGo [1], which discovered 26 previously unknown bugs in security-critical libraries, including 10 severe vulnerabilities that were subsequently assigned CVEs.

Symbolic execution [18–25] provides precise path reasoning by solving branch constraints. By translating path conditions into symbolic constraints, it calculates the exact inputs needed to reach deep code. However, this rigorous computation leads to scalability issues when used alone. To balance efficiency and precision, hybrid fuzzing [26–32] combines fuzzing with symbolic execution. This enhances directed exploration while retaining scalability. Similarly, SymCC [18] demonstrates this effectiveness by identifying two new CVE-assigned vulnerabilities in OpenJPEG.

Despite extensive research efforts, existing DGF [1–15] and hybrid fuzzing [26–32] techniques still face fundamental challenges. First, symbolic execution was originally designed to maximize overall code coverage by exploring different execution paths. However, when traditional symbolic execution engines [18–25] are combined with DGF, they often attempt to solve branch constraints without considering whether those branches are relevant to the target. This leads to many redundant test cases on unrelated paths, reducing the overall fuzzing efficiency. Second, DGF relies on distance metrics to guide execution toward the target. These metrics estimate how close a test case is to it. However, most DGF approaches calculate this distance using static control-flow graphs. This static view completely ignores the semantic complexity of branch conditions. Recent tools like SelectFuzz [33] attempt probability-based estimation, but they oversimplify program structures. They fail to handle loops, back-edges, and unstructured jumps. Consequently, existing metrics may provide less informative guidance for target reachability in the presence of complex control-flow structures. In hybrid directed fuzzing [30,32], this inaccurate guidance may steer heavyweight symbolic execution toward infeasible or target-irrelevant paths, thereby amplifying overall inefficiency.

These limitations can be illustrated through a motivating example shown in Figure 1. The program accepts an input string $s$ and reaches a vulnerable code region only when $s$ equals the literal "DEADBEEF". A traditional fuzzer repeatedly mutates the input but is unlikely to produce this exact value through random mutations. Symbolic execution, on the other hand, can efficiently solve the equality constraint and guide execution into the basic block $E$, where a function `Vuln()` re-validates the input and may contain the actual vulnerability. However, reaching this deeper vulnerability requires not only satisfying the initial constraint but also further mutating the input to explore diverse execution paths within `Vuln()`. At the same time, existing distance metrics provide misleading guidance: when applying SelectFuzz's probability-based distance calculation [33] to this program, basic blocks A, B, C, and D are assigned reachability probabilities of 1/2, 1/2, 0, and 0, respectively, even though blocks C and D are clearly feasible paths to the target according to the CFG. This discrepancy arises because current probability models collapse or ignore cyclic and irregular control-flow structures, resulting in distorted distance information. Moreover, hybrid fuzzing frameworks that use symbolic execution without selection tend to generate redundant test cases that repeatedly go through similar and target-unrelated execution paths, further reducing fuzzing efficiency. Together, these issues highlight the need for more accurate distance modeling, more targeted mutation guidance, and tighter integration between directed fuzzing and symbolic execution.

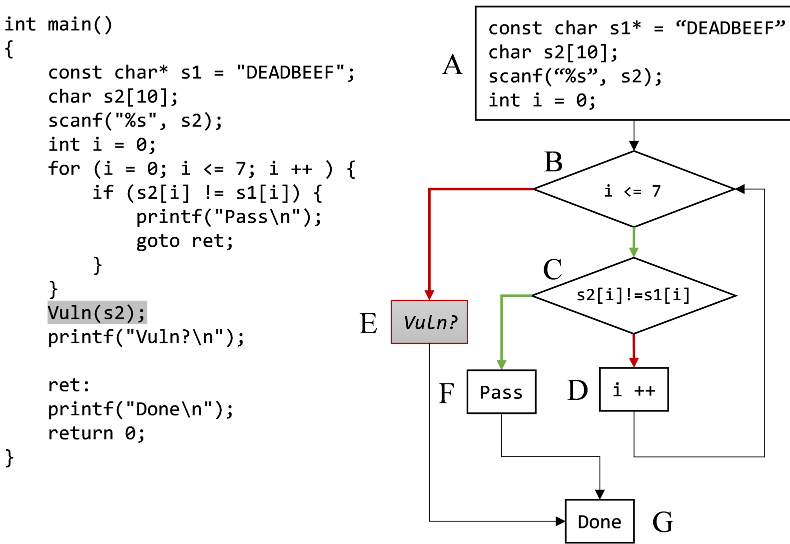


**Figure 1.** A motivational example.

These observations motivate the need for a more principled and integrated approach to hybrid directed fuzzing [30,32] that provides accurate reachability guidance, selective and lightweight symbolic execution, and tight coordination between fuzzing and symbolic reasoning. A key insight of this work is that probability-based reachability modeling can provide far more informative distance signals than traditional static metrics [1], but only when complex control-flow behaviors (loops, indirect jumps, back-edges, and recursive structures) are precisely resolved rather than collapsed or approximated. Complementing this, selective symbolic instrumentation restricts symbolic execution to control- and data-flow-dependent regions, mitigating overhead. Finally, a unified coordination mechanism allows symbolic execution and fuzzing components to share coverage, distance, and state information coherently.

To address these challenges, we introduce TriFuzz, a hybrid directed fuzzing framework [30,32] guided by probabilistic distance. TriFuzz integrates fine-grained distance modeling, selective compile-time symbolic instrumentation, and a tightly coupled coordination mechanism between AFLGo [1] and SymCC [18]. TriFuzz estimates path feasibility through a fine-grained probabilistic distance calculation, prunes basic blocks that are unreachable from the target, and assigns semantically meaningful distance weights to guide fuzzing more reliably. Selective symbolic instrumentation substantially reduces symbolic execution overhead, while a tightly integrated hybrid coordination mechanism ensures that symbolically generated test cases are evaluated and prioritized using the fuzzer's global state.

Our contributions are as follows:

- **TriFuzz framework.** We propose a hybrid directed fuzzing framework [30,32] along three complementary dimensions, including fine-grained probabilistic distance calculation, selective symbolic instrumentation, and hybrid coordination. These components are designed to work together to improve the efficiency of directed exploration.
- **Implementation.** We implement TriFuzz as a practical directed hybrid fuzzer by extending AFLGo [1] and SymCC [18]. The system integrates the proposed techniques into a unified framework for efficient directed fuzzing. To facilitate reproducibility and further research, the full implementation of TriFuzz has been made publicly available at https://github.com/HandsomeBoyBiu/trifuzz (accessed on 15 April 2026).

- **Comprehensive evaluation.** Experiments on AFLGo's benchmark suite [1] and UniBench [34] show that TriFuzz achieves significantly faster time-to-target and time-to-exposure, while reducing symbolic redundancy and distance-calculation overhead.

## 2. Background

The rapid expansion of modern software systems has led to unprecedented complexity in control-flow structures, input-handling logic, and execution environments. As a result, manually constructing test cases or conducting traditional code audits has become increasingly inefficient and error-prone. To address this challenge, the research community has sought automated, scalable, and effective testing techniques capable of discovering deep vulnerabilities with minimal human effort. Among these approaches, fuzzing has emerged as a cornerstone technology due to its ability to automatically generate high-volume test inputs and expose diverse classes of software defects. AFL [16] pioneered coverage-guided greybox fuzzing by mutating inputs to maximize code coverage with lightweight instrumentation, while AFL++ [17] extended this design with improved mutation stages, scheduling algorithms, and extensive modular enhancements for modern software targets. To address scenarios requiring focused exploration of specific code regions, directed greybox fuzzing was introduced, with AFLGo [1] providing the first distance-based framework for bias fuzzing toward user-specified target locations. This section provides background on three major techniques closely related to our work: directed greybox fuzzing, symbolic execution, and hybrid fuzzing.

### 2.1. Directed Greybox Fuzzing

Directed greybox fuzzing (DGF) [1–15] aims to guide the fuzzing process toward specific program locations that are of particular interest, such as patched code, crash sites, or vulnerability-related functions. Compared with traditional coverage-guided fuzzers, DGF provides target-oriented exploration [1], improving the efficiency of triggering vulnerabilities. Its key advantage lies in lightweight instrumentation and fast feedback loops, which allow it to scale to large real-world applications. However, current DGF techniques still face several limitations: distance metrics often rely on static structural approximations that fail to capture complex control-flow behaviors; mutation strategies remain mostly unguided, making it difficult to satisfy intricate path predicates; and energy scheduling depends heavily on the accuracy of precomputed distances. These challenges motivate the need for more precise, probabilistic, and execution-aware guidance mechanisms.

Building on AFLGo's distance-based seed prioritization [1], modern directed greybox fuzzers refine directed fuzzing with enhanced metrics and strategies. WindRanger [35] steers executions toward targets by leveraging deviation basic blocks and their associated data-flow information to calculate seed distances and guide mutation. BEACON [36] introduces provable path-pruning techniques that eliminate infeasible program paths, markedly improving efficiency in reaching vulnerable code areas. MC2 [37] recasts directed fuzzing as an oracle-guided search problem, using rigorous path cost modeling to systematically find target-reaching inputs faster than heuristic methods. For scalability, FISHFUZZ [38] enables directed fuzzing across thousands of target locations by prioritizing seeds toward interesting code regions, achieving more comprehensive program coverage in large-scale scenarios. More recently, learning-based approaches have emerged: DeepGo [39] uses deep neural networks to predict high-reward path transitions and a reinforcement learning agent to drive the fuzzer along an optimal path to the target site. Similarly, IDFuzz [40] leverages a neural model to learn from historical mutations and focus fuzzing on critical input fields, significantly accelerating target vulnerability reproduction and uncovering new bugs compared to baseline directed fuzzers.

### 2.2. Symbolic Execution

Symbolic execution (SE) [18–25] provides a fundamentally different approach by treating program inputs as symbolic variables and generating path constraints to systematically enumerate feasible execution paths. Its major strength lies in *deterministic path reasoning*: SE can construct inputs that satisfy specific branch conditions and reliably reach deep or hard-to-trigger program states. This makes SE particularly attractive in directed testing scenarios. Nevertheless, symbolic execution suffers from well-known scalability issues, including path explosion, solver overhead, and difficulties modeling external environments or system interactions. Even modern advances such as solver optimizations, selective instrumentation, and parallel constraint solving cannot fully eliminate these inherent bottlenecks. Consequently, symbolic execution alone is insufficient for efficiently exploring large or complex software targets.

Symbolic execution provides a systematic method for exploring program paths by encoding branch conditions as symbolic constraints, enabling the deterministic generation of target-reaching inputs. Classical engines such as KLEE [19] and S2E [20] remain foundational due to their comprehensive path constraint management and mature solver integrations, while more recent LLVM-based engines [41] like SymCC [18] significantly accelerate symbolic execution by compiling symbolic expressions directly into native code to reduce solving latency. Further advancements include JIGSAW [42], which JIT-compiles path constraints to improve solver throughput by over an order of magnitude, enabling scalable multi-core constraint solving. Nonetheless, symbolic execution remains fundamentally challenged by *path explosion*, external environment modeling, and high solver overhead. To mitigate these scalability issues, recent work attempts to prune unproductive paths before invoking the solver, but pure symbolic approaches still struggle on complex real-world codebases. These persistent limitations highlight the necessity of cooperative frameworks that combine symbolic precision with fuzzing efficiency.

### 2.3. Hybrid Fuzzing

Hybrid fuzzing [26–31] combines the efficiency of fuzzing with the precision of symbolic execution, invoking constraint solving only when random mutations fail to make progress. This cooperative design allows hybrid systems to overcome path-blocking conditions, traverse deep program states, and generate inputs that satisfy complex branch predicates. Hybrid fuzzers benefit from fuzzing's scalability and symbolic execution's targeted reasoning, making them effective across a wide range of applications. However, their performance remains constrained by the overhead of symbolic execution and the difficulty of deciding *when* and *where* SE should be applied. Furthermore, existing hybrid frameworks typically lack accurate distance modeling and often explore symbolic paths unrelated to the ultimate target, leading to unnecessary solver consumption.

Hybrid fuzzing seeks to combine the efficiency of greybox fuzzing with the precision of symbolic execution, selectively invoking constraint solving to break through path-blocking conditions. Early influential systems such as Driller [26] and QSYM [21] established the paradigm of demand-driven symbolic execution to assist fuzzers when coverage plateaus. More recently, DigFuzz [27] enhanced hybrid exploration by prioritizing symbolic queries based on dynamic path utility, while 1dVul [43] integrated probabilistic vulnerability localization to guide solver resources toward security-critical regions. In the kernel domain, HFL [28] introduced a hybrid fuzzing framework tailored to kernel control flows, leveraging targeted symbolic execution to overcome complex system-state constraints. In addition, several modern directed hybrid fuzzers have emerged: ColorGo [29] integrates color-based target partitioning to coordinate symbolic solving with directed fuzzing; HyperGo [30] combines a multi-objective search with lightweight path constraint solving to accelerate

reaching semantically distant target locations; and HLPFuzz [31] employs LLM-based constraint inference to navigate grammatically complex input spaces. Collectively, these hybrid approaches demonstrate that carefully orchestrating fuzzing and symbolic execution can overcome the inherent limitations of each technique, motivating the cooperative design philosophy adopted in this work.

Although HyperGo [30] is closely related to TriFuzz in that both combine probability-based guidance with selective symbolic reasoning, the two systems emphasize different design stages. HyperGo mainly performs its probability-aware guidance at runtime, because it combines AFLGo-style structural distance [1] with dynamically updated path probability during fuzzing and uses the OSEC scheme to invoke symbolic execution for selected critical branches along execution paths. In contrast, TriFuzz places more emphasis on static analysis before fuzzing begins. It computes a fine-grained probabilistic distance through inter-procedural reachability modeling in the analysis stage, and it identifies target-relevant basic blocks for selective symbolic instrumentation at compile time, so its symbolic reasoning is guided by pre-extracted relevant regions rather than being decided only online during execution.

## 3. Methodology

### *3.1. Framework Overview*

TriFuzz is built on two key design principles: fine-grained probability-based distance calculation and selective symbolic execution. The former enables accurate reachability estimation by explicitly resolving complex control-flow structures, while the latter reduces redundant test cases by restricting instrumentation to target-relevant regions. These components are integrated into an AFLGo [1]-SymCC [18] hybrid directed fuzzing architecture that leverages shared runtime feedback. This section provides an overview of TriFuzz's architecture, shown in Figure 2.

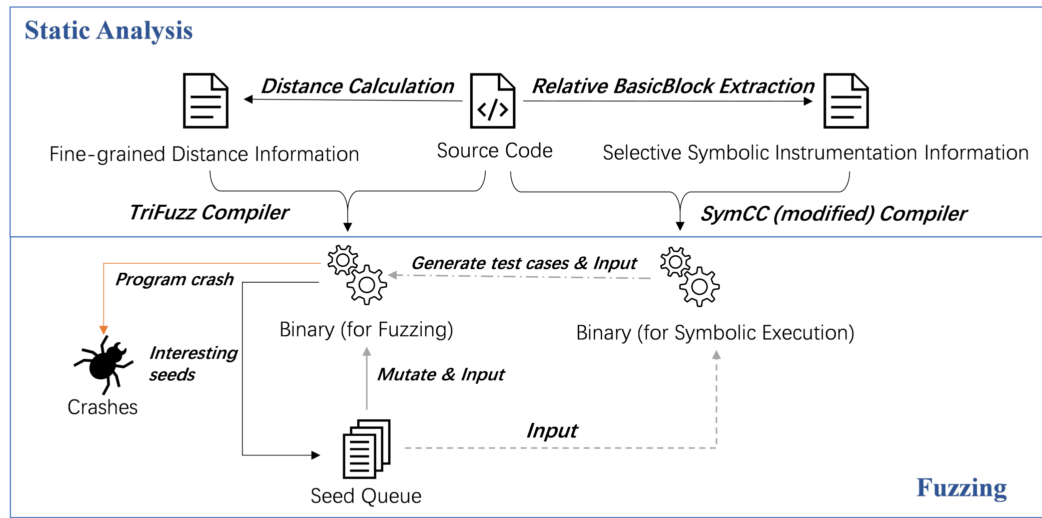


**Figure 2.** The framework of TriFuzz.

TriFuzz is organized into three modules: (1) fine-grained probability-based distance calculation, (2) selective symbolic instrumentation, and (3) hybrid fuzzing coordination.

The workflow of TriFuzz consists of three main stages. During the static analysis stage, TriFuzz computes the probability-based distance for each basic block and extracts target-related basic blocks for selective symbolic instrumentation. The detailed distance-calculation algorithm and target-related basic block extraction procedure are described in the following sections. Then, TriFuzz compiles the program using two instrumented com-

pilation pipelines. The fuzzing compiler generates the binary used by the greybox fuzzer, which contains the runtime instrumentation required for coverage collection and distance-guided seed scheduling. Meanwhile, the modified SymCC compiler generates a selectively instrumented binary, where symbolic instrumentation is inserted only into the target-related regions identified by the static analysis. Finally, TriFuzz launches the two binaries together and coordinates their execution through the hybrid fuzzing coordination mechanism. During fuzzing, promising seed inputs are forwarded to the symbolic-execution side, where the selectively instrumented binary is executed to solve relevant path constraints and generate new test cases. These generated test cases are then returned to the fuzzing side for distance-based evaluation and further mutation.

In the following subsections, we describe each module in detail.

#### 3.1.1. Fine-Grained Probability-Based Distance Calculation

The probability-based distance-calculation module aims to provide accurate and semantically meaningful guidance toward target locations. TriFuzz performs inter-procedural static analysis on the program source code to extract control-flow information and constructs a fine-grained probability-based reachability model. Based on structured analysis and iterative probability propagation, TriFuzz estimates the probability that each basic block can eventually reach the specified target. Unlike traditional structural distance metrics [1,33], this approach explicitly accounts for loops, back-edges, and indirect control-flow constructs while pruning unreachable blocks. The resulting distance profile provides a more fine-grained static approximation of target reachability and serves as the primary guidance signal for directed fuzzing.

#### 3.1.2. Selective Symbolic Instrumentation

The selective symbolic instrumentation module focuses on reducing symbolic execution overhead and the generation of redundant test cases by restricting instrumentation to semantically relevant program regions. Based on static control-flow and data-flow dependency analysis, TriFuzz identifies basic blocks and functions that either influence target reachability or are influenced by target-dependent data. These dependencies are used to generate a selective symbolic instrumentation profile, which specifies the subset of functions to be instrumented by SymCC [18]. By avoiding full-program symbolic instrumentation, TriFuzz significantly reduces path explosion and limits the generation of redundant symbolic test cases.

#### 3.1.3. Hybrid Fuzzing Coordination

The hybrid directed fuzzing coordination module integrates AFLGo [1] and SymCC [18] into a tightly coupled system. AFLGo [1] performs continuous directed fuzzing using the probability-based distance metric to prioritize promising inputs, while SymCC [18] is invoked selectively to perform symbolic execution on chosen seeds. Unlike loosely coupled hybrid designs, TriFuzz establishes a coordination mechanism based on shared memory and semaphore-based signaling, enabling AFLGo [1] and SymCC [18] to exchange execution state, coverage information, and newly generated test cases. This cooperative design ensures that symbolic execution directly benefits directed fuzzing and avoids redundant analysis. As a result, TriFuzz achieves faster convergence toward target locations and more efficient vulnerability exposure.

### *3.2. Fine-Grained Probability-Based Distance Calculation*

We propose a fine-grained, probability-based distance metric to evaluate how likely a given input execution is to reach the target basic block. This metric reflects both the structural proximity and the execution probability toward the target, thereby improving

https://doi.org/10.3390/electronics15102049

fuzzing guidance. Compared with existing directed greybox fuzzers, our design offers two main advantages:

1. Unreachable-block elimination: all basic blocks that cannot reach the target are pruned from the control-flow graph (CFG), avoiding misleading distance values.
2. Loop-aware distance modeling: cyclic structures are explicitly modeled, ensuring that distance calculation within loops is semantically consistent and fine-grained.

These improvements allow the fuzzer to more accurately evaluate which inputs have a higher probability of reaching the target.

To make the subsequent formulas and definitions easier to follow, Table 1 summarizes the main notations used in the rest of this section and their corresponding meanings.

**Table 1.** Notation used in the probability-based distance-calculation model.

| Notation | Description |
|---|---|
| $x$ | A basic block in the program. |
| $S_x^{(n)}$ | The set of $n$-hop successor basic blocks of basic block $x$. |
| $S_{\text{Conf}}$ | The set of conflict basic blocks. |
| $S_{\text{Predef}}$ | $S_{\text{Predef}} = S_{\text{Target}} \cup S_{\text{Exit}} \cup S_{\text{Call}}$. The set of predefined basic blocks whose probabilities are assigned before distance calculation, such as target blocks, exit blocks, and call-site blocks. |
| $S_{\text{Leaf}}$ | $S_{\text{Leaf}} = S_{\text{Predef}} \cup S_{\text{Conf}}$. The set of leaf basic blocks, including all predefined basic blocks and conflict basic blocks. |
| $\text{CT}_t$ | The conflict tree rooted at basic block $t$. |
| $\mathcal{F}_{\text{Conf}}$ | The conflict forest, i.e., the set of all conflict trees in a control-flow graph. |
| $C_{\text{CT}_t,x}$ | The leaf-coefficient vector of basic block $x$ in conflict tree $\text{CT}_t$, used to construct the linear equation system. |
| $M_{\text{CT}_t}$ | The matrix composed of the leaf-coefficient vectors in conflict tree $\text{CT}_t$. |
| $P(x)$ | The static target-reachability probability of basic block $x$. |
| $D(x)$ | The distance value derived from $P(x)$ and used by the fuzzer. |
| $D_{\text{input}}$ | The distance value of an input. |
| $Prog(input)$ | The set of functions executed by the program on input $input$. |

#### 3.2.1. Conflict Basic Block Probability Calculation

Let the function be represented as a CFG, where each node corresponds to a basic block and each edge represents a possible transfer of control. Due to the presence of loops or `goto` statements, CFGs may contain loops. We decompose CFGs into multiple subgraphs, which can be categorized as cyclic subgraphs and acyclic subgraphs. In cyclic subgraphs, basic blocks may participate in loops or recursive control transfers, creating cyclic dependencies. We define a *conflict basic block* $x$ as any block that can reach itself through one or more successor steps, i.e., there exists some $n > 0$ such that $x \in S_x^{(n)}$, where $S_x^{(n)}$ denotes the set of blocks reachable from $x$ within $n$ edges and $S_x^{(0)} = \{x\}$. The set of all such blocks is written as

$$S_{\text{Conf}} = \{x \mid \exists n > 0,\ x \in S_x^{(n)}\}$$

For each conflict basic block $t \in S_{\text{Conf}}$, we construct a **conflict tree** $\text{CT}_t$ rooted at $t$, representing all possible forward paths from $t$ until reaching either a conflict basic block or a predefined block. A node $x$ becomes a leaf of $\text{CT}_t$ if it has no successors, or if it belongs to the set of predefined blocks $S_{\text{Predef}}$, or if it is itself a conflict basic block.

$$S_{\text{Leaf}} = S_{\text{Predef}} \cup S_{\text{Conf}}$$

where $S_{\text{Predef}}$ represents the set of basic blocks whose probabilities are **predefined constants**. This set contains the target basic blocks $S_{\text{Target}}$, each assigned a probability of 1; the exit blocks $S_{\text{Exit}}$, each assigned probability of 0; and the call-site blocks $S_{\text{Call}}$, each assigned the probability of the entry block of its callee function. In other words, $S_{\text{Predef}} = S_{\text{Target}} \cup S_{\text{Exit}} \cup S_{\text{Call}}$, capturing all blocks whose probabilities can be determined without solving the conflict-tree system.

While each conflict tree captures the dependency relations within a local cycle, the program's CFG may contain **multiple interconnected cycles**. Thus, all conflict trees together form a **conflict forest**, denoted as $\mathcal{F}_{\text{Conf}} = \{\text{CT}_t \mid t \in S_{\text{Conf}}\}$, which globally represents all cyclic regions in the CFG. The forest structure enables us to compute the reachability probabilities for all conflict basic blocks in a unified manner.

For each conflict tree $\text{CT}_t$, the leaf coefficient $C_{\mathbf{CT}_t,x}$ is defined as the sum of $\text{LCM}(L_i)/|S_x^{(1)}|$, where $|S_x^{(1)}|$ is the number of successors of the basic block $x$. When $x$ has no successors, i.e., $|S_x^{(1)}| = 0$, we define $|S_x^{(1)}| = 1$ for coefficient calculation to avoid division by zero. $\text{LCM}(L_i)$ is the least common multiple of the successor counts for all basic blocks at the level $L_i$ in $\text{CT}_t$. Collectively, these coefficients form the matrix

$$M_{\text{CT}_t} = (C_{\text{CT}_t,x_1}, C_{\text{CT}_t,x_2}, \ldots, C_{\text{CT}_t,x_n})$$

which is normalized using the root coefficient $C_{R_t} = \prod_{L=1}^{D_t} \text{LCM}(L)$, where $D_t$ represents the maximum depth of the conflict tree.

By combining the normalized relations across all conflict trees, we obtain a global linear system $\text{X} = \text{AX} + \text{C}$, where $\text{X} = (x_1, x_2, \ldots, x_n)^\top$ is the vector of reachability probabilities for all conflict basic blocks, A is constructed from the scaled coefficient vectors $M_{\text{CT}_t}/C_{R_t}$, and C collects constant contributions from terminal and predefined nodes. Solving this system yields the reachability probability for each conflict basic block, thereby quantifying its likelihood of reaching the target through cyclic execution paths.

#### 3.2.2. Recursive Distance Propagation

Once all cyclic dependencies have been resolved through conflict basic block probability calculation, every basic block in the program can be treated as part of an acyclic control-flow graph. At this point, the probability of reaching the target can be computed in a unified, recursive manner across the entire program.

Let $P(x)$ denote the probability that a basic block $x$ can eventually reach any target basic block $t \in S_{\text{Target}}$. The computation proceeds in two hierarchical stages: intra-procedural propagation (within functions) and inter-procedural propagation (across function calls).

Within a function $F$, the control-flow graph has no unresolved cycles after conflict resolution. We recursively calculate the reachability probability for each basic block starting from the function's entry block. For a basic block $x$ whose probability has not yet been assigned, its probability $P(x)$ is computed as follows. For basic blocks that already have probability values, their existing probabilities are directly reused.

$$P(x) = \frac{1}{|S_x^{(1)}|} \sum_{i \in S_x^{(1)}} P(i)$$

This recursive formulation ensures that each basic block's reachability probability reflects both local control-flow transitions and cross-function propagation through call sites. After computing all intra-procedural probabilities, we extend the analysis across functions using the call graph of the program. For each call edge $(f_i, f_j)$ where the function $f_i$ invokes $f_j$, the probability of the call site block in $f_i$ is updated using the entry block of $f_j$:

$$P(\text{call}_{f_i \to f_j}) = P(\text{entry}(f_j))$$

Algorithm 1 summarizes the probability-based distance-calculation procedure designed in this work. At the intra-procedural level, the function `bb_level_probability` first invokes the `resolve_conflict` routine to compute probability values for all *conflict basic blocks* and eliminate cyclic dependencies within the function. The detailed resolution of

conflict basic blocks is discussed in Section 3.2.1. After all conflict basic blocks are resolved, the algorithm computes the probability of each basic block starting from the function's entry block, recursively aggregating successor probabilities until the entire function obtains a complete probability assignment.

At the inter-procedural level, the function `func_level_probability` takes the entire program as input and begins by evaluating all functions that directly contain target basic blocks. It then recursively traverses the reversed call graph, computing the probability of each caller function based on the probabilities of its callees. This propagation continues until every function that can eventually reach the target basic block has been assigned a stable probability value.

**Algorithm 1** Probability-based distance calculation

**Require:** *Program* with target function set $\mathcal{S}_T$, constant probability map *ConstProb*
**Ensure:** *Prob* : probability map over all basic blocks

```
 1: function FUNC_LEVEL_PROBABILITY(Program, ConstProb)
 2:     initialize Prob(b) ← ⊥ for all basic blocks b in Program
 3:     worklist ← ∅
 4:     for each function F in S_T do
 5:         worklist.push(F)
 6:     end for
 7:     while worklist ≠ ∅ do
 8:         F ← worklist.pop()
 9:         BB_LEVEL_PROBABILITY(F, ConstProb, Prob)
10:         for each caller function G of F do
11:             if G has not been processed then
12:                 worklist.push(G)
13:             end if
14:         end for
15:     end while
16:     return Prob
17: end function
18:
19: function BB_LEVEL_PROBABILITY(F, ConstProb, Prob)
20:     RESOLVE_CONFLICT(F, ConstProb, Prob)
21:     COMPUTEBBPROB(F.EntryBlock, Prob)
22: end function
23:
24: function COMPUTEBBPROB(BB, Prob)
25:     if BB.Status is PREDEF or CONFLICT then
26:         return Prob(BB)
27:     end if
28:     sum ← 0
29:     for each successor SuccBB of BB do
30:         p ← COMPUTEBBPROB(SuccBB, Prob)
31:         sum ← sum + p
32:     end for
33:     Prob(BB) ← sum/|Succ(BB)|
34:     return Prob(BB)
35: end function
```

Algorithm 2 presents the pseudocode of the `resolve_conflict` procedure, which consists of two main stages. The first stage constructs the conflict trees. Since a function may contain multiple loops, several conflict trees may need to be built within the same function, with each tree corresponding to one conflict basic block. Specifically, the construction process determines whether a basic block belongs to a cycle. If so, that basic block is treated as a conflict basic block and used as the root of a conflict tree. Therefore, each conflict basic block corresponds to one conflict tree.

**Algorithm 2** RESOLVE_CONFLICT: conflict basic block probability calculation

**Require:** Function $F$, constant probability map $ConstProb$
**Ensure:** $Prob$: probability map over basic blocks in $F$
1: **function** RESOLVE_CONFLICT($F, ConstProb, Prob$)
2: $S_{\text{Conf}} \leftarrow \emptyset$
3: $\mathcal{F}_{\text{Conf}} \leftarrow \emptyset$
4: $S_{\text{Predef}} \leftarrow \text{dom}(ConstProb)$
▷ **Stage 1: construct the conflict forest**
5: **for** each basic block $x$ in $F$ **do**
6: **if** $\exists n > 0$ such that $x \in S_x^{(n)}$ **then**
7: $S_{\text{Conf}} \leftarrow S_{\text{Conf}} \cup \{x\}$
8: **end if**
9: **end for**
10: **for** each conflict basic block $t \in S_{\text{Conf}}$ **do**
11: $CT_t \leftarrow$ BUILDCONFLICTTREE($F, t, S_{\text{Conf}}, S_{\text{Predef}}$)
12: $\mathcal{F}_{\text{Conf}} \leftarrow \mathcal{F}_{\text{Conf}} \cup \{CT_t\}$
13: **end for**
▷ **Stage 2: construct and solve the linear system**
14: enumerate $S_{\text{Conf}}$ as $\{b_1, b_2, \ldots, b_m\}$
15: initialize $A \in \mathbb{R}^{m \times m}$ with zeros
16: initialize $C \in \mathbb{R}^m$ with zeros
17: **for** each conflict tree $CT_t \in \mathcal{F}_{\text{Conf}}$ **do**
18: $r \leftarrow \text{index}(t)$
19: initialize $M_{CT_t}(x) \leftarrow 0$ for each conflict basic block $x$ in $S_{\text{Conf}}$
20: **for** each layer $L_i$ in $CT_t$ **do**
21: $\text{LCM}(L_i) \leftarrow$ least common multiple of successor counts of blocks in $L_i$
22: **end for**
23:
24: **for** each leaf node $x \in S_{\text{Leaf}}$ of $CT_t$ **do**
25: $L_i \leftarrow x.$**getLayer**()
26: $d_x \leftarrow \max(|successor(x)|, 1)$
27: $C_{CT_t,x} \leftarrow \text{LCM}(L_i)/d_x$
28: **if** $x \in S_{\text{Conf}}$ **then**
29: $M_{CT_t}(x) \leftarrow M_{CT_t}(x) + C_{CT_t,x}$
30: **else if** $x \in S_{\text{Predef}}$ **then**
31: $C[r] \leftarrow C[r] + C_{CT_t,x} \cdot ConstProb[x]$
32: **end if**
33: **end for**
34:
35: $C_{R_t} \leftarrow 1$
36: **for** each layer $L_i$ in $CT_t$ **do**
37: $C_{R_t} \leftarrow C_{R_t} \cdot \text{LCM}(L_i)$
38: **end for**
39:
40: **for** each conflict basic block $x$ with $M_{CT_t}(x) \neq 0$ **do**
41: $\alpha \leftarrow M_{CT_t}(x)/C_{R_t}$
42: $c \leftarrow \text{index}(x)$
43: $A[r, c] \leftarrow \alpha$
44: **end for**
45: $C[r] \leftarrow C[r]/C_{R_t}$
46: **end for**
47:
48: $X \leftarrow$ LINEARSOLVE($I - A, C$)
49: **for** $i \leftarrow 1$ to $m$ **do**
50: $Prob(b_i) \leftarrow X[i]$
51: **end for**
52: **for** each basic block $x \in S_{\text{Predef}}$ **do**
53: $Prob(x) \leftarrow ConstProb(x)$
54: **end for**
55: **return** $Prob$
56: **end function**

The second stage builds and solves the corresponding system of linear equations. To compute the probabilities of conflict basic blocks, it is necessary to derive the coefficients associated with both conflict basic blocks and constant-probability basic blocks. Lines 27–47 of Algorithm 2 describe the calculation of these coefficients. After obtaining the coefficients contributed by all conflict trees, the system of equations in the form $X = AX + C$ can be constructed and solved, yielding the probability values of all conflict basic blocks in the current CFG.

### 3.2.3. Input Distance

After computing reachability probabilities for all basic blocks and functions, we convert these values into distance measures that quantify how far an input execution is from the target. For a basic block $x$, its distance $D(x)$ is defined as a monotonic transformation of its reachability probability. Target-equivalent blocks with $P(x) = 1$ are assigned distance 0, unreachable blocks with $P(x) = 0$ are assigned `INT_MAX`, and all intermediate cases use the inverse form $D(x) = 1/P(x)$. This formulation is motivated by both intuitive and practical considerations. It preserves a monotonic relationship between reachability and distance, such that less reachable regions are assigned larger distances, which aligns with the goal of directed fuzzing. Moreover, the inverse form is directly compatible with AFLGo's distance-based scheduling mechanism, enabling integration without additional normalization overhead. The distance is defined as follows:

$$D(x) = \begin{cases} 0, & P(x) = 1, \\ \text{INT_MAX}, & P(x) = 0, \\ \frac{1}{P(x)}, & \text{otherwise} \end{cases}$$

The function-level distance $D(f)$ is defined analogously as the distance associated with the entry block of function $f$. During fuzzing, each generated input is assigned an input-level distance, which summarizes its proximity to the target by aggregating the distances of all functions invoked during execution. Following AFLGo's design philosophy, this input distance is computed as the average distance across all executed functions, formally expressed as

$$D_{\text{input}} = \frac{1}{|Prog(input)|} \times \sum_{f \in Prog(input)} D(f)$$

where $Prog(input)$ denotes the set of executed functions for the given test case and $|Prog(input)|$ is its cardinality. This formulation ensures that inputs traversing functions with high reachability probabilities receive proportionally smaller distance values, thereby guiding mutation toward increasingly promising execution paths.

### 3.2.4. Distance-Calculation Example

To illustrate the proposed probability-based distance calculation, we present a simplified example derived from the CFG in Figure 1; the reduced form used for analysis is shown in Figure 3. In this CFG, basic block E is the designated target and is therefore assigned probability 1, while block G is an exit block whose probability is defined as 0. The remaining blocks propagate their probabilities based on the structure of the CFG.

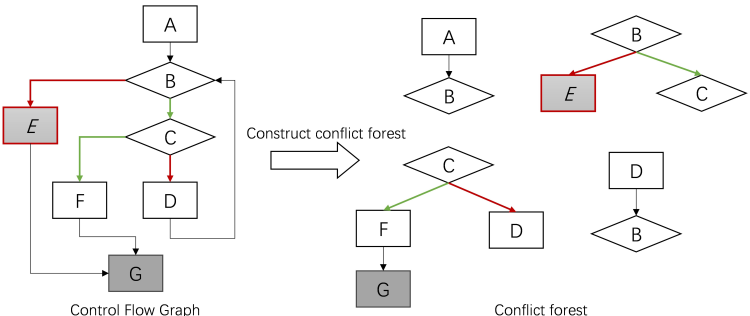


**Figure 3.** A simplified example for distance calculation.

We first transform the CFG into its corresponding conflict forest, which separates cyclic regions and allows each conflict component to be represented independently. Using the conflict forest, we derive the system of linear equations capturing the probabilistic dependencies among basic blocks. For this example, the recurrence relations are:

- Block $A$ inherits the probability of its successor $B$;
- Block $B$ branches to $E$ and $C$, giving $B = (1 + C)/2$;
- Block $C$ branches to $F$ and $D$; since $F \to G$ contributes probability 0, this reduces to $C = D/2$;
- Block $D$ returns to $B$, completing the cycle.

These relations yield the following linear system:

$$\begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \\ 0 & 1 & 0 & 0 \end{pmatrix} \begin{pmatrix} A \\ B \\ C \\ D \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \\ 0 \\ 0 \end{pmatrix}.$$

Solving this system yields the reachability probabilities

$$A = \frac{2}{3}, \quad B = \frac{2}{3}, \quad C = \frac{1}{3}, \quad D = \frac{2}{3}.$$

This example highlights a key advantage of our approach: unlike prior probability-based distance models that collapse or approximate loops, the proposed method computes exact reachability probabilities for cyclic basic blocks, ensuring that guidance remains accurate even in the presence of recursive structures or strongly connected CFG components. Consequently, the resulting distance values provide a much finer-grained and semantically meaningful gradient for directed fuzzing.

### 3.3. Selective Symbolic Instrumentation

Symbolic execution can precisely reason about path constraints, but its effectiveness in hybrid directed fuzzing critically depends on where symbolic reasoning is applied. Unrestricted symbolic instrumentation often leads to the generation of large numbers of test cases that are irrelevant to the target. For example, as illustrated in Figure 4 where $G$ is the target basic block, suppose a seed initially follows the execution path $A \to B \to C \to E \to I$. If symbolic execution is applied to all basic blocks, it may generate two new test cases whose execution paths are $A \to B \to C \to H \to I$ and $A \to B \to D \to F \to I$, respectively. Among them, the path $A \to B \to C \to H \to I$ is entirely irrelevant to the target and thus wastes symbolic exploration effort. In contrast, if symbolic instrumentation is restricted to the target-relevant basic blocks $B$ and $D$, the first round of symbolic execution generates the path $A \to B \to D \to F \to I$, and the second round further generates the path $A \to B \to D \to G \to I$, which successfully reaches the target basic block $G$. This

example shows that selective symbolic instrumentation can better focus symbolic reasoning on target-related paths. To address this challenge, TriFuzz introduces a selective symbolic instrumentation module that constrains symbolic execution to semantically relevant program regions. By leveraging static control-flow and data-flow dependency analysis, this module identifies basic blocks and functions that directly influence target reachability or vulnerability triggering. Restricting symbolic instrumentation to these regions significantly reduces symbolic execution overhead while ensuring that symbolic exploration contributes meaningfully to directed fuzzing.

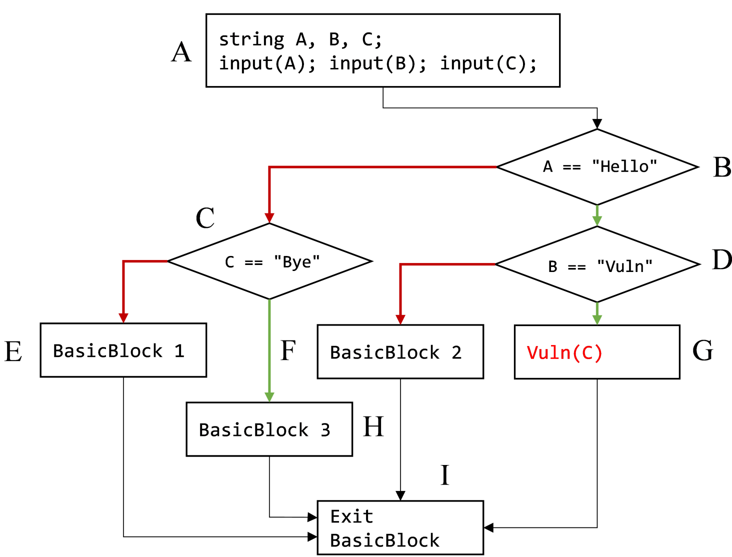


**Figure 4.** Example of selective symbolic execution.

To minimize symbolic execution overhead and enhance the relevance of generated test cases, we perform selective compile-time instrumentation during the SymCC [18] compilation phase. Instead of instrumenting all basic blocks uniformly, our method selectively instruments only those blocks that are semantically related to the target, determined through a combination of control-flow and data-flow dependency analysis. All unrelated basic blocks are excluded from symbolic instrumentation, ensuring that symbolic exploration primarily focuses on execution paths that contribute to reaching or triggering the target.

A control-flow-dependent basic block is one that lies on any potential control-flow path leading to the target basic block. Formally, let $S_x^{(n)}$ denote the set of basic blocks reachable from block $x$ within $n$ control-flow edges. Then, the set of control-flow-dependent basic blocks is defined as

$$S_{\mathrm{CF}} = \{x \mid \exists n,\ S_x^{(n)} \cap S_{\mathrm{Target}} \neq \varnothing\}$$

That is, any basic block $x$ whose reachable successors include at least one target basic block belongs to $S_{\mathrm{CF}}$. This ensures that all blocks that can influence the execution path toward the target are retained for instrumentation.

For inter-procedural control dependencies, we extend this definition across function boundaries. If function $f_i$ contains a call site that invokes a callee $f_j$, and any block in $f_j$ belongs to $S_{\mathrm{CF}}$, then the call site block in $f_i$ is also marked as control-flow-dependent:

$$\forall f_i, f_j,\ (\mathrm{call}_{i \to j} \in f_i) \wedge (S_{\mathrm{CF}} \cap f_j \neq \varnothing) \Rightarrow \mathrm{call}_{i \to j} \in S_{\mathrm{CF}}$$

This propagation ensures that the analysis captures control dependencies that span multiple functions, thereby preserving the full set of inter-procedural paths to the target.

While control-flow dependencies facilitate reaching the target code, data-flow dependencies help trigger the actual vulnerability once the target is reached. We identify data-flow-dependent basic blocks based on variable read/write relationships relative to the target basic block. For inter-procedural data dependencies, the analysis is extended to function parameters, pointer dereferences, and return values.

Let $V$ be the set of program variables and $B$ the set of basic blocks. Each block $b \in B$ has a set of defined variables $\text{Def}(b) \subseteq V$ and used variables $\text{Use}(b) \subseteq V$. We define a data-dependence relation $\mathcal{D} \subseteq B \times B$ as

$$(b_i, b_j) \in \mathcal{D} \iff \exists v \in V,\ v \in \text{Def}(b_i) \land v \in \text{Use}(b_j)$$

This relation connects any pair of basic blocks where a variable written in $b_i$ is later read in $b_j$. By recursively applying this relation, we obtain its transitive closure $\mathcal{D}^*$, which models indirect dependencies through intermediate variables or definitions.

Based on this relation, the data-flow dependency closure from a target basic block $t$ is defined as

$$S_{\text{DF}}(t) = \{\, b \in B \mid (b, t) \in \mathcal{D}^* \ \lor\ (t, b) \in \mathcal{D}^* \,\}$$

Here, the first condition $(b, t) \in \mathcal{D}^*$ captures all blocks that influence the target (backward dependencies), while the second condition $(t, b) \in \mathcal{D}^*$ captures all blocks influenced by the target (forward dependencies). The global set of data-flow-dependent blocks for all targets is then as follows:

$$S_{\text{DF}} = \bigcup_{t \in S_{\text{Target}}} S_{\text{DF}}(t)$$

This formalism provides a unified representation of data-flow relationships and can be interpreted as a static approximation of taint propagation: it captures both backward (use-to-definition) and forward (definition-to-use) dependency chains across the program.

To clarify how control-flow and data-flow information is extracted to support selective symbolic instrumentation, we present an illustrative example that demonstrates the analysis process in detail. Consider a target statement `c = a + b`. Since `c` is written, we perform forward tracking on `c` to find all later statements that read it. If we find `e = c + d`, we further trace the new variable `e`, following its subsequent reads recursively. Simultaneously, we perform backward tracking on the read variables `a` and `b`: if `a` is defined as `a = d + e`, the analysis recursively follows `d` and `e` to their definitions. The union of these forward and backward dependency chains yields the complete set of data-flow-dependent blocks. Only these semantically relevant regions are selected for symbolic instrumentation, ensuring that symbolic execution focuses on inputs most likely to influence the target.

### *3.4. Hybrid Fuzzing Coordination*

While selective symbolic instrumentation reduces symbolic overhead and the generation of redundant test cases, its benefits can only be realized through effective coordination between symbolic execution and directed fuzzing. Existing hybrid fuzzing systems often rely on loosely coupled designs, in which symbolic execution operates in isolation and produces test cases without awareness of the fuzzer's global state, leading to redundancy and inefficient feedback utilization. To overcome these limitations, TriFuzz introduces a hybrid directed fuzzing coordination module that tightly integrates AFLGo [1] and SymCC [18] through an explicit coordination mechanism. This module uses shared memory and semaphore-based synchronization, enabling the two engines to exchange execution context, coverage feedback, and generated test cases in real time. As a result, symbolic

reasoning is invoked precisely when it can assist directed exploration, and its outputs are immediately incorporated into the fuzzing workflow.

We redesign the coordination mechanism that integrates AFLGo [1] with SymCC [18], referred to as the *bridge* (i.e., the connector between AFLGo [1] and SymCC [18]). The overall architecture of the bridge is illustrated in Figure 5.

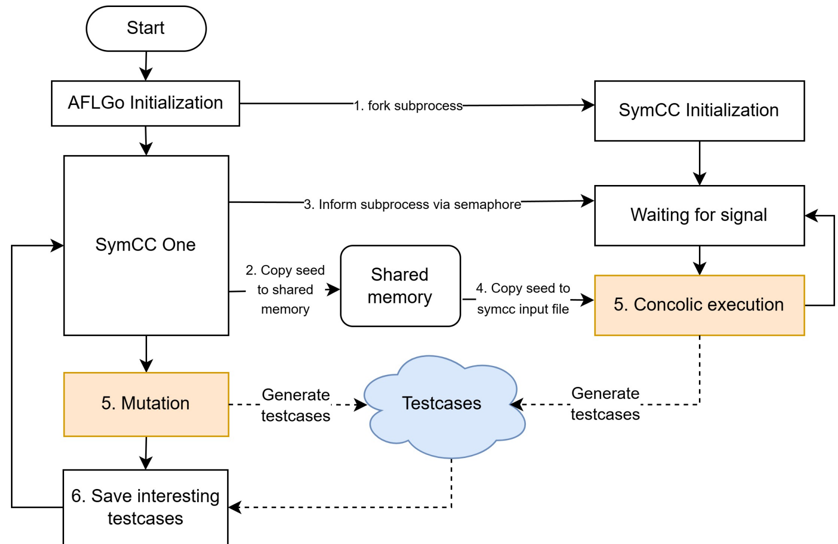


**Figure 5.** The framework of hybrid fuzzing coordination.

After AFLGo [1] completes its initialization, the bridge is immediately set up. This initialization process involves creating shared memory regions, initializing semaphores, and configuring execution parameters and environment variables. Once the setup is complete, the fuzzer forks a child process, within which SymCC [18] runs concurrently with the fuzzer. During execution, SymCC [18] continuously receives signals from the parent fuzzer process, allowing both symbolic and fuzzing components to operate in parallel and exchange information efficiently.

Since the bridge is integrated directly into the fuzzer engine, both components can share global variables. This design enables the fuzzer to evaluate test cases generated by SymCC's [18] symbolic execution using its own internal quality metrics. In contrast, in the original SymCC [18] implementation, the *fuzz helper* assessed the "interestingness" of generated inputs using only local information, such as its own coverage bitmap. As a result, redundant test cases were often mislabeled as interesting, even when the fuzzer had already explored them.

Furthermore, the redesigned bridge reuses the existing functions within the fuzzer engine. This not only achieves a higher level of integration but also reduces the overhead associated with process creation and teardown. Unlike the SymCC [18] fuzz helper, which repeatedly invokes external system commands to execute the `afl-showmap` [16] utility, the integrated bridge enables internal calls within the same process space. As a result, the number of process launches is significantly reduced, improving the overall efficiency of hybrid directed fuzzing.

## 4. Implementation

We implemented **TriFuzz** in about 2800 lines of C/C++ code. Of these, roughly 2100 lines are devoted to the static analysis module for SymCC's [18] selective symbolic instrumentation and distance calculation, and about 700 lines implement the integration bridge between AFLGo [1] and SymCC [18]. TriFuzz is built on the LLVM framework [41], using custom LLVM passes for control-flow analysis, data-flow analysis, and distance

calculation. TriFuzz operates on LLVM bitcode, enabling fine-grained program analysis and compatibility with standard Clang/LLVM toolchains.

Next, we discuss a few important implementation details.

Distance Calculation

During distance calculation, we observed that for large-scale programs (e.g., *objdump*), the analysis could take over one hour and consume more than 20 GB of memory. We traced the cause to the recursive construction of conflict trees, each rooted at a conflict basic block. Since each tree requires multiple depth-first traversals, a CFG with numerous conflict basic blocks leads to significant computational overhead.

To mitigate this issue, we introduced a pruning optimization based on subtree reuse. We found that many conflict trees share structural similarities and overlapping subgraphs. Therefore, information obtained from the traversal of one conflict tree can be reused when constructing subsequent trees, such as node hierarchy, depth levels, and subtree structures. By caching and reusing these intermediate results, we avoid redundant traversals, substantially reducing both analysis time and memory consumption while preserving the accuracy of distance calculation.

Hybrid Fuzzing Coordination

To enable parallel execution of AFLGo [1] and SymCC [18] while preventing race conditions, we designed an inter-process communication mechanism within the hybrid fuzzing bridge. The bridge relies on shared memory and semaphores as its communication mechanisms, enabling coordinated data transfer and execution scheduling between the fuzzer and the symbolic executor.

During each mutation stage, AFLGo [1] copies the current seed data into a shared memory region. This operation is protected by a locking mechanism to avoid concurrent access conflicts. Once the data is written, a semaphore signal is sent to notify the SymCC [18] child process. Upon receiving the signal, SymCC [18] reads the seed from shared memory, writes it to its input file, and begins symbolic execution. All operations following the signal reception proceed in parallel with AFLGo's [1] ongoing fuzzing loop, maximizing CPU utilization and overall throughput.

Indirect Call

To ensure the completeness of inter-procedural dependency detection, we resolve indirect function calls. We employ the Andersen points-to analysis from the SVF [44] (Static Value-Flow Analysis Framework) to build a conservative call graph and identify possible callees for each indirect call site. Depending on the program scale, implementation, and programming language, this conservative approach may introduce different numbers of irrelevant functions. We will discuss the impact of these irrelevant functions on the experimental results in later sections.

## 5. Evaluation

In this section, we conducted an evaluation of **TriFuzz** using two widely adopted fuzzing benchmarks, the AFLGo testsuite [1] and UniBench [34]. To demonstrate the effectiveness of our approach, we compared TriFuzz with several state-of-the-art directed and hybrid fuzzers. To ensure fairness, all compared fuzzers were built on the same image, with consistent settings for system libraries, including the *libc* 2.31 and Python 3.8.10 runtime. Under this criterion, we selected QSYM, SelectFuzz, and BEACON as baselines, since they can be executed in nearly identical environments and share a common subset of benchmark programs with TriFuzz. In addition, QSYM is related to our selective symbolic execution design, SelectFuzz is closely related to our distance-calculation strategy,

and BEACON represents a strong directed fuzzing baseline. To isolate the contribution of each component, we designed an ablation study by comparing TriFuzz with TriFuzz-Full, TriFuzz-nSE, and AFLGo. Specifically, TriFuzz-Full applies full symbolic instrumentation to all basic blocks while keeping other components the same as TriFuzz, thereby evaluating the effect of selective instrumentation. TriFuzz-nSE corresponds to AFLGo equipped with our fine-grained distance model (i.e., TriFuzz without symbolic execution), allowing us to assess the contribution of the distance model independently. These settings enable a clearer understanding of how each component affects the overall performance. Our experiment answered the following questions:

- **RQ1:** How efficiently can TriFuzz reach the designated target locations?
- **RQ2:** How effectively can TriFuzz expose target vulnerabilities?
- **RQ3:** What advantages does selective symbolic instrumentation provide over full instrumentation?
- **RQ4:** Does our probability-based fine-grained distance metric outperform traditional distance metrics in guiding fuzzing toward the target?

To address these questions, we compared TriFuzz against multiple baselines and examined its performance across nine benchmark cases for which SymCC [18] can successfully generate inputs.

### *5.1. Evaluation Setup*

#### 5.1.1. Evaluation Criteria

We use three criteria to quantitatively evaluate the effectiveness of directed fuzzers.

**Time-To-Expose (TTE):** The elapsed time from the start of fuzzing until the first program crash is triggered. Since a crash corresponds to a triggered vulnerability, TTE directly reflects vulnerability discovery efficiency.

**Time-To-Reach (TTR):** The elapsed time from the start of fuzzing until the first execution reaches the designated target basic block(s). This metric measures the quality of the distance metric and guidance effectiveness.

**Redundant Test Case Variance (RTV):** Designed to evaluate RQ3, RTV quantifies redundancy by measuring the consistency of input effectiveness across repeated runs. For each test case, we run multiple fuzzing trials and compute the variance of its TTE values:

$$\mathrm{RTV} = \frac{1}{n}\sum_{i=1}^{n}(\mathrm{TTE}_i - \bar{\mathrm{TTE}})^2$$

Lower RTV indicates fewer redundant or misleading test cases and thus better instrumentation selectiveness.

$\hat{A}_{12}$ **Effect Size:** We additionally report the Vargha–Delaney $\hat{A}_{12}$ effect size [45] to quantify the practical performance difference between fuzzers. $\hat{A}_{12}$ measures the probability that a randomly selected run of one fuzzer outperforms that of another. An $\hat{A}_{12}$ value of 0.5 indicates no difference, while larger values indicate stronger performance advantage.

**Statistical Significance (*p*-value):** To determine whether the performance differences between fuzzers are statistically significant, we compute $p$-values using the Mann–Whitney U test. This non-parametric test compares the results of repeated runs without assuming normal distributions. In our evaluation, differences are considered statistically significant when $p < 0.05$.

#### 5.1.2. Evaluation Benchmarks

We evaluate TriFuzz on AFLGo testsuite [1] and UniBench [34]. AFLGo test suite [1] was introduced in the AFLGo paper. It contains a range of real-world programs with manually specified target locations and is commonly adopted as a benchmark in directed

fuzzing studies to test the efficiency of directed fuzzing tools. UniBench [34] is a widely adopted fuzzing benchmark tool that contains a set of real-world target programs designed to represent the complexity and vulnerability characteristics of practical software. It provides a unified execution environment and standardized configuration, and has become a commonly used benchmark for evaluating modern fuzzing tools. Since SymCC [18] can only generate valid symbolic inputs for a subset of test cases, we restrict our evaluation to the nine benchmark cases that support SymCC-based symbolic execution.

Table 2 summarizes all benchmark programs used in our evaluation. As shown in the table, the benchmark suite covers a diverse set of input domains, including image, text, audio, and video processing, which helps provide a broader assessment of the effectiveness of TriFuzz across different application types. To give readers a clearer sense of program complexity, we use lines of code (LoCs) as an approximate indicator. The selected benchmarks span a wide range of code sizes, suggesting that the evaluation includes both relatively small and more complex real-world programs. It is worth noting that *LMS* and *Palindrome* originate from the DARPA CGC Challenge and therefore do not have corresponding version numbers or CVE identifiers. For *jasper*, *mjs*, *LMS*, and *Palindrome*, we leave the seed type unspecified, since the initial seeds provided in the AFLGo testsuite are empty files. We keep this setting to ensure consistency in seed types across these benchmarks.

**Table 2.** Summary of benchmark programs.

| Project | Program | Type | Seed Type | LoC | Version | CVE/Issue | CWE |
|---|---|---|---|---|---|---|---|
| jasper | jasper | Image | - | 45,554 | 1.900.1 | CVE-2015-5221 | CWE-416 |
| mjs | mjs-bin | Text/JSE | - | 44,233 | 1.25 | issues-57/issues-78 | CWE-190/CWE-416 |
| LMS | LMS | Text/DARPA | - | 9440 | - | - | - |
| Palindrome | Palindrome | Text/DARPA | - | 250 | - | - | - |
| jhead | jhead | Image | jpg | 5649 | 3.00 | CVE-2018-6612 | CWE-125, CWE-191 |
| xpdf | pdftotext | Text | pdf | 28,958 | 4.00 | CVE-2018-7175 | CWE-476 |
| Bento4 | mp42aac | Video | mp4 | 89,693 | 1.5.1-628 | CVE-2020-19721 | CWE-787 |
| mp3gain | mp3gain | Audio | mp3 | 12,140 | 1.5.2 | CVE-2017-14409 | CWE-787 |

Note: “JSE” denotes JavaScript Engine and “DARPA” denotes DARPA CGC Challenge.

#### 5.1.3. Experimental Settings

All experiments were executed within a Docker-based evaluation environment deployed on a workstation equipped with an Intel(R) Core(TM) i9-14900HX (2.20 GHz) CPU and 48 GB RAM, running Ubuntu 20.04 LTS as the host operating system. Each fuzzing campaign was repeated 15 times under identical container configurations, with a per-run time budget of 24 h. The fuzzing parameters strictly followed the default settings provided by the AFLGo testsuite [1] example scripts and the UniBench [34] benchmark suite, without any manual tuning. The initial seed corpus for AFLGo testsuite [1] was directly taken from its default seed set. For UniBench [34], we sorted the provided seeds by size in ascending order and selected the first five as the initial corpus. This choice is motivated by our observation that excessively large seeds can significantly reduce fuzzing throughput and thus adversely affect overall fuzzing efficiency. All experiments used identical seed configurations across different fuzzers. In addition, all fuzzing instrumentation settings remain unchanged and follow the default configurations of the corresponding tools. This setup ensures fair and reproducible comparisons across all evaluated approaches.

When reporting the time-based metrics, including Time-To-Reach (TTR) and Time-To-Expose (TTE), we exclude the time spent on compilation and preprocessing, and measure only the elapsed time from the start of fuzzing to reaching the target or triggering the vulnerability. This design is motivated by two considerations. On the one hand, compilation

and preprocessing do not constitute the main fuzzing process, and all compared fuzzers require target compilation before fuzzing, making this cost roughly comparable across tools. On the other hand, the preprocessing overhead of distance calculation is reported in Section 5.6. Our results show that the distance-calculation time of TriFuzz is usually below one minute, and even for the most complex programs it does not exceed five minutes. Compared with the overall fuzzing time budget, this overhead is negligible. Therefore, excluding compilation and preprocessing time allows TTR and TTE to more accurately reflect the effectiveness of different fuzzers.

### 5.2. Reaching Target Sites

To answer RQ1, we compared the **Time-To-Reach (TTR)** of TriFuzz against several representative directed and hybrid fuzzers, including AFLGo [1], TriFuzz-nSE (TriFuzz without symbolic execution), QSYM [21], SelectFuzz [33], and BEACON [36]. TTR measures the elapsed time from the start of fuzzing until the designated target basic block is first executed. Lower values indicate more efficient target-oriented exploration. Each experiment was repeated 15 times with a maximum time budget of 24 h, and the averaged results are summarized in Table 3. The notation “T.O.” indicates that the fuzzer fails to reach the target within the time limit.

**Table 3.** Time-To-Reach (TTR) and statistical significance ($p$-value) across benchmarks.

| Benchmark | TriFuzz | TriFuzz-Full | | TriFuzz-nSE | | AFLGo | | QSYM | | SelectFuzz | | BEACON | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\mu$TTR | $\mu$TTR | $p$(TriFuzz) | $\mu$TTR | $p$(AFLGo) | $\mu$TTR | $p$ | $\mu$TTR | $p$ | $\mu$TTR | $p$ | $\mu$TTR | $p$ |
| jasper | 6.3 | 6.55 | 0.385 | 17.29 | 0.320 | 18.55 | 0.0024 | 10.5 | 0.032 | 11.2 | 0.043 | 12.5 | 0.0396 |
| mjs-issues-78 | 4.1 | 4.3 | 0.445 | 4.4 | 0.471 | 4.9 | 0.385 | 4.3 | 0.429 | 5.3 | 0.338 | 4.2 | 0.395 |
| mjs-issues-57 | 12,387.8 | 14,127.3 | 0.068 | 37,091.2 | 0.332 | 38,709.3 | 0.00085 | 6377.2 | 0.0035 | 23,934.6 | 0.00032 | 25,718.3 | 0.00076 |
| LMS | 301.6 | 317.9 | 0.222 | 292.9 | 0.442 | 334.5 | 0.423 | 244.6 | 0.049 | 359.2 | 0.060 | 304.7 | 0.450 |
| Palindrome | 1.26 | 1.21 | 0.329 | 1.12 | 0.417 | 1.15 | 0.448 | 1.23 | 0.526 | 1.13 | 0.379 | 1.22 | 0.476 |
| jhead | 209.42 | 330.58 | 0.007 | 1823.41 | 0.00002 | 33,321.85 | 0.00003 | 1946.2 | 0.0054 | 3395.7 | 0.00012 | 10,923.5 | 0.00001 |
| xpdf (pdftotext) | T.O. | T.O. | - | T.O. | - | T.O. | - | T.O. | - | T.O. | - | T.O. | - |
| Bento4 (mp42aac) | T.O. | T.O. | - | T.O. | - | T.O. | - | T.O. | - | T.O. | - | T.O. | - |
| mp3gain | 60.1 | 370.5 | 0.0002 | 9234.1 | 0.00012 | 24,098.9 | 0.00001 | 119.5 | 0.0034 | 253.7 | 0.00037 | 58.2 | 0.398 |

Note: Underlined values indicate that the corresponding fuzzer outperforms the other compared fuzzers on that benchmark.

Overall, TriFuzz achieves competitive or superior performance across most benchmarks. On *jasper*, TriFuzz reaches the target in 6.3 s, outperforming all other fuzzers including AFLGo [1] (18.55 s), SelectFuzz [33] (11.2 s), and BEACON [36] (12.5 s). Similar improvements are observed on *jhead*, where TriFuzz reaches the target in 209.4 s, while AFLGo requires over 9 h and other fuzzers take significantly longer. On *mjs-issues-57*, although QSYM [21] reaches the target fastest due to aggressive symbolic solving, TriFuzz still reduces the reaching time by approximately threefold compared with AFLGo (12,387 s vs. 38,709 s). For lighter benchmarks such as *mjs-issues-78* and *Palindrome*, most fuzzers exhibit similar performance due to the shallow control-flow structure, yet TriFuzz still maintains competitive reaching speed.

For more complex applications such as *pdftotext* and *Bento4 (mp42aac)*, none of the fuzzers were able to reach the designated target locations within the 24-hour time budget. Manual inspection using *gdb* [46] showed that the selected target sites were located after the vulnerable statements. Specifically, we set breakpoints at all target locations and executed the programs with known crashing PoCs. The program crashed before reaching any of the breakpoints, indicating that the targets were not on the execution path leading to the crash. As a result, the program terminated immediately once the vulnerability was triggered, preventing further execution from reaching the target location. Since these target locations are directly taken from the official UniBench target files, this discrepancy should

be understood as a benchmark-specific limitation of the provided target definitions rather than an issue of our implementation.

On *mjs-issues-57*, TriFuzz underperforms QSYM [21] in terms of TTR. We attribute this result mainly to two factors. First, the target location in this benchmark is relatively deep in the program, which makes efficient guidance more challenging. Second, our static analysis does not eliminate enough redundant functions in this case, leading to the generation of some redundant test cases during hybrid execution. Figure 6 shows the Selective Instrumentation Ratio (SIR) of all evaluated programs, which is defined as

$$\mathrm{SIR} = \frac{N_{\mathrm{Instr}}}{N_{\mathrm{All}}} \times 100\%$$

where $N_{\mathrm{Instr}}$ denotes the number of functions selected for instrumentation in the program, and $N_{\mathrm{All}}$ denotes the total number of functions in the program. We use SIR as an indicator of how broadly target-related functions are distributed in the program, and thus as an indirect reflection of the complexity of target-reaching control flow. As shown in Figure 6, the SIR of *mjs* remains very high, reaching 99.4%, which suggests that a large portion of functions are regarded as target-related and therefore preserved during selective instrumentation. We found that *mjs* has a single-file structure with relatively high coupling among functions. When the target code is located in a later part of the program, many functions may appear target-related and are therefore included in the target-relevant function set. As a result, selective instrumentation discards fewer irrelevant functions, and symbolic execution may still generate a considerable number of unrelated test cases, which ultimately reduces fuzzing efficiency.

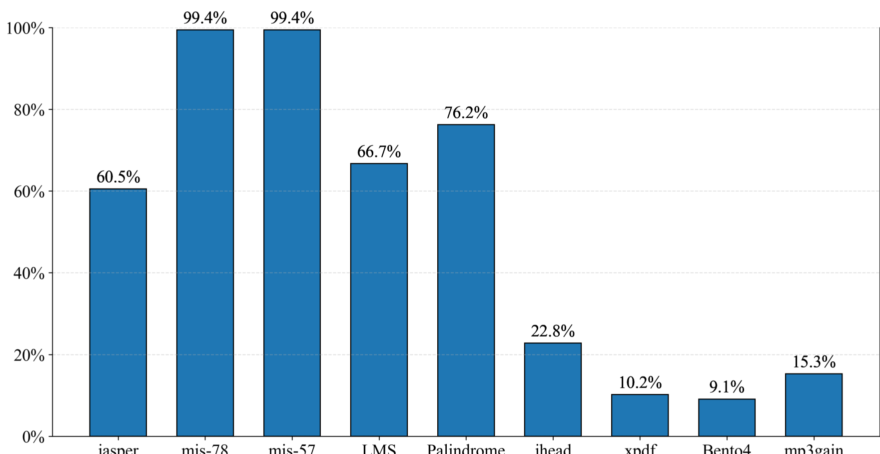


**Figure 6.** Selective Instrumentation Ratio (SIR) of all programs.

To further analyze the contribution of each component, we conducted an ablation study. We first compared TriFuzz-nSE with AFLGo [1], where TriFuzz-nSE corresponds to TriFuzz without symbolic execution and differs from AFLGo only in the distance metric. The results showed that the two methods achieve comparable performance on most benchmarks, while TriFuzz-nSE demonstrated a clear advantage on *jhead* and *mp3gain*, indicating that the proposed distance model can provide more effective guidance in certain cases. To evaluate the effect of selective symbolic instrumentation, we further compared TriFuzz with TriFuzz-Full. The results showed that TriFuzz achieves noticeable improvement only on *mp3gain*, suggesting that while selective instrumentation can reduce unnecessary overhead, its impact depends on the characteristics of the target program.

These findings answer **RQ1**: Overall, TriFuzz achieves favorable target-reaching performance in the evaluated programs. The results indicate that TriFuzz is particularly

effective when the Selective Instrumentation Ratio remains low, since most target-irrelevant paths can be filtered out and symbolic reasoning can be concentrated on target-related regions. When the instrumentation ratio is high, however, the selected region becomes close to full instrumentation, which may introduce redundant test cases and reduce the performance gap between TriFuzz and other hybrid fuzzing methods. For shallow targets, the TTR values of different fuzzers are often close, as such targets can be reached without requiring sophisticated distance guidance or symbolic assistance.

### *5.3. Exposing Target Vulnerabilities*

To answer RQ2, we evaluated the **Time-To-Expose (TTE)** of TriFuzz and compared it with several representative directed and hybrid fuzzers. TTE measures the elapsed time from the start of fuzzing until the first vulnerability-triggering input is generated. Lower values therefore indicate faster vulnerability discovery. Each experiment was repeated 15 times with a maximum time budget of 24 h, and the averaged results were summarized in Table 4. Table 5 further illustrates the relative performance improvement ratios for both TTR and TTE. In addition, we report the $p$-value computed using the Mann–Whitney U test to assess the statistical significance of the observed differences. Specifically, the $p$-values for TriFuzz-Full are computed against TriFuzz, those for TriFuzz-nSE are computed against AFLGo, and all other $p$-values are computed against TriFuzz.

**Table 4.** Time-To-Expose (TTE) and statistical significance ($p$-value) across benchmarks.

| Benchmark | TriFuzz | TriFuzz-Full | | TriFuzz-nSE | | AFLGo | | QSYM | | SelectFuzz | | BEACON | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $\mu$TTE | $\mu$TTE | $p$ (TriFuzz) | $\mu$TTE | $p$ (AFLGo) | $\mu$TTE | $p$ | $\mu$TTE | $p$ | $\mu$TTE | $p$ | $\mu$TTE | $p$ |
| jasper | 10.9 | 13.4 | 0.178 | 46.3 | 0.221 | 53.5 | 0.0011 | 20.1 | 0.028 | 30.1 | 0.017 | 35.7 | 0.015 |
| mjs-issues-78 | 1046.7 | 1210.8 | 0.013 | 5362.5 | 0.0023 | 8718.8 | 0.00005 | 2316.9 | 0.014 | 4427.9 | 0.0032 | 3892.1 | 0.0053 |
| mjs-issues-57 | 4427.1 | 6871.9 | 0.0047 | 12,930.2 | 0.0026 | 28,021.3 | 0.00003 | 3698.1 | 0.024 | 10,596.2 | 0.022 | 8915.5 | 0.015 |
| LMS | 5218.2 | 7293.6 | 0.017 | 7721.5 | 0.225 | 7564.2 | 0.034 | 5940.5 | 0.182 | 9501.4 | 0.025 | 4011.6 | 0.044 |
| Palindrome | 1.60 | 1.87 | 0.353 | 1.82 | 0.547 | 1.75 | 0.651 | 1.10 | 0.173 | 1.65 | 0.479 | 1.78 | 0.514 |
| jhead | 540.7 | 882.3 | 0.012 | 27,891.5 | 0.0003 | 47,612.2 | 0.00001 | 2398.2 | 0.0037 | 5273.5 | 0.0021 | 13,647.8 | 0.0005 |
| xpdf (pdftotext) | 2602 | 2934.25 | 0.151 | 4025.3 | 0.019 | 4893.4 | 0.0076 | 4216.2 | 0.0018 | 602.6 | 0.0066 | 5932.1 | 0.013 |
| Bento4 (mp42aac) | 392.5 | 416.1 | 0.149 | 412.9 | 0.182 | 420.3 | 0.287 | 490.1 | 0.031 | 448.2 | 0.251 | 149.6 | 0.030 |
| mp3gain | 84.45 | 280.7 | 0.0007 | 881.2 | 0.453 | 918.5 | 0.00002 | 190.7 | 0.013 | 523.1 | 0.0008 | 128.9 | 0.039 |

Note: Underlined values indicate that the corresponding fuzzer outperforms the other compared fuzzers on that benchmark.

**Table 5.** Relative improvement ratios (TTR and TTE).

| Metric | Comparison | jasper | mjs-78 | mjs-57 | LMS | Palindrome | jhead | pdftotext | mp42aac | mp3gain | All |
|---|---|---|---|---|---|---|---|---|---|---|---|
| TTR | TriFuzz & TriFuzz-Full | 1.0× | 1.0× | 1.1× | 1.1× | 1.0× | 1.6× | - | - | 6.2× | 1.9× |
| | TriFuzz-nSE & AFLGo | 1.1× | 1.1× | 1.0× | 1.1× | 1.0× | 18.3× | - | - | 2.6× | 3.8× |
| | QSYM | 1.7× | 1.0× | 0.5× | 0.8× | 1.0× | 9.3× | - | - | 2.0× | 2.3× |
| | SelectFuzz | 1.8× | 1.3× | 1.9× | 1.2× | 0.9× | 16.2× | - | - | 4.2× | 3.9× |
| | BEACON | 2.0× | 1.0× | 2.1× | 1.0× | 1.0× | 52.2× | - | - | 1.0× | 8.6× |
| TTE | TriFuzz & TriFuzz-Full | 1.2× | 1.2× | 1.6× | 1.4× | 1.2× | 1.6× | 1.1× | 1.1× | 3.3× | 1.5× |
| | TriFuzz-nSE & AFLGo | 1.2× | 1.6× | 2.2× | 1.0× | 1.0× | 1.7× | 1.2× | 1.0× | 1.0× | 1.3× |
| | QSYM | 1.8× | 2.2× | 0.8× | 1.1× | 0.7× | 4.4× | 1.6× | 1.2× | 2.3× | 1.8× |
| | SelectFuzz | 2.8× | 4.2× | 2.4× | 1.8× | 1.0× | 9.8× | 0.2× | 1.1× | 6.2× | 3.3× |
| | BEACON | 3.3× | 3.7× | 2.0× | 0.8× | 1.1× | 25.2× | 2.3× | 0.4× | 1.5× | 4.5× |

Overall, TriFuzz achieves the fastest vulnerability exposure on several benchmarks and remains competitive on most others. For instance, on *jasper*, TriFuzz triggers the vulnerability in 10.9 s, substantially outperforming AFLGo [1] (53.5 s), SelectFuzz [33] (30.1 s), and BEACON [36] (35.7 s). Similar improvements are observed on *mjs-issues-*

*78*, where TriFuzz exposes the vulnerability in 1046.7 s, while AFLGo [1] requires over 8700 s and other directed fuzzers take significantly longer. We observed that all fuzzers achieve similar performance in terms of TTR on this benchmark, indicating that reaching the target location is not the main challenge. Instead, vulnerability triggering requires the program to be in a specific state when reaching the target. TriFuzz benefits from its selective symbolic instrumentation guided by data-flow dependencies, which enables it to more effectively satisfy such conditions and reach vulnerability-triggering states faster. Another particularly notable case is *jhead*, where TriFuzz reduces the exposure time from 47,612 s for AFLGo to only 540.7 s, achieving nearly two orders of magnitude improvement. For more challenging benchmarks such as *mjs-issues-57*, QSYM [21] triggers the vulnerability slightly faster due to its aggressive symbolic solving strategy. Nevertheless, TriFuzz still significantly outperforms AFLGo and other directed fuzzers, reducing the exposure time from 28,021 s to 4427 s. On lightweight benchmarks such as *Palindrome*, most fuzzers expose the vulnerability within a few seconds due to the shallow execution paths, resulting in only marginal performance differences.

Interestingly, hybrid execution alone does not always guarantee optimal performance. We observe that, for some complex programs, vulnerability triggering requires not only reaching the target location but also satisfying specific program states, which are difficult to obtain purely through constraint solving. For example, on *pdftotext*, SelectFuzz [33] achieves the fastest exposure time. We believe this is because vulnerability triggering in *pdftotext* depends not only on reaching target-related code, but also on satisfying specific parser states induced by structured PDF inputs. In such a scenario, SelectFuzz's lightweight selective path exploration is more effective at prioritizing target-adjacent paths with lower overhead, whereas the symbolic-execution component of TriFuzz may generate additional test cases that are locally feasible but do not efficiently lead to the exact vulnerability-triggering state. As a result, although TriFuzz still outperforms heavier hybrid baselines such as QSYM, it remains less effective than SelectFuzz on this benchmark in terms of TTE. Similarly, on *Bento4 (mp42aac)* and *LMS*, where vulnerability triggering is more challenging and execution overhead is higher, BEACON [36] outperforms our approach by pruning unreachable paths early, thereby reducing redundant exploration and improving efficiency. Nevertheless, TriFuzz still shows a clear advantage over QSYM [21] on these programs. Taking *pdftotext* as an example, Figure 6 shows that the SIR of TriFuzz is only 10.2%, indicating that selective instrumentation filters out a large portion of target-irrelevant functions while preserving those more closely related to the target. This allows TriFuzz to apply symbolic execution in a more focused manner. In contrast, QSYM adopts a full-instrumentation strategy, which is more likely to incur unnecessary symbolic overhead on such programs. As a result, although TriFuzz does not achieve the best performance on these benchmarks, it remains noticeably more efficient than QSYM.

To further quantify the statistical strength of these improvements, Table 6 also reports the Vargha–Delaney $\hat{A}_{12}$ effect size [45], a non-parametric measure widely used in fuzzing evaluation. $\hat{A}_{12}$ represents the probability that a random TTE sample from TriFuzz is smaller (i.e., better) than a random sample from another fuzzer. An $\hat{A}_{12}$ value greater than 0.5 indicates superiority, and values above 0.7 are typically interpreted as strong effects. As shown in the table, TriFuzz attains $\hat{A}_{12}$ values of 0.973 (jasper), 0.958 (mjs-issues-78), 0.897 (mjs-issues-57), and 0.990 (jhead) when compared with AFLGo [1], providing strong statistical evidence that the reductions in TTE are not due to randomness but stem from consistently more efficient vulnerability discovery.

https://doi.org/10.3390/electronics15102049

**Table 6.** RTV and TTE/TTR $\hat{A}_{12}$ [45] results on the AFLGo test suite and UniBench.

| Prog. | RTV | | | | TTE $\hat{A}_{12}$ | | TTR $\hat{A}_{12}$ | |
|---|---|---|---|---|---|---|---|---|
| | TriFuzz | TriFuzz-Full | TriFuzz-nSE | AFLGo | $\hat{A}_{12}$ (TF, AFLGo) | $\hat{A}_{12}$ (TF, TFfull) | $\hat{A}_{12}$ (TF, AFLGo) | $\hat{A}_{12}$ (TF, TFfull) |
| jasper | 59.8 | 87.8 | 412.3 | 405.9 | 0.973 | 0.637 | 0.853 | 0.530 |
| mjs-issues-78 | $4.98 \times 10^5$ | $8.69 \times 10^5$ | $8.09 \times 10^6$ | $1.46 \times 10^7$ | 0.958 | 0.461 | 0.534 | 0.512 |
| mjs-issues-57 | $1.83 \times 10^7$ | $3.20 \times 10^7$ | $5.31 \times 10^8$ | $5.20 \times 10^8$ | 0.897 | 0.640 | 0.922 | 0.569 |
| LMS | $9.99 \times 6$ | $1.04 \times 10^7$ | $1.30 \times 10^7$ | $1.33 \times 10^7$ | 0.360 | 0.632 | 0.651 | 0.511 |
| Palindrome | 1.07 | 1.03 | 1.07 | 1.05 | 0.524 | 0.581 | 0.493 | 0.493 |
| jhead | $6.34 \times 4$ | $2.90 \times 10^5$ | $1.89 \times 10^6$ | $1.92 \times 10^6$ | 0.990 | 0.789 | 0.995 | 0.860 |
| xpdf (pdftotext) | $4.75 \times 5$ | $5.52 \times 10^5$ | $2.00 \times 10^6$ | $2.26 \times 10^6$ | 0.910 | 0.649 | - | - |
| Bento4 (mp42aac) | $1.66 \times 4$ | $1.68 \times 10^4$ | $2.01 \times 10^4$ | $1.70 \times 10^4$ | 0.564 | 0.555 | - | - |
| mp3gain | 235.9 | $4.03 \times 10^3$ | $7.03 \times 10^4$ | $7.80 \times 10^4$ | 0.988 | 0.482 | 0.814 | 0.975 |

Note: Underlined values indicate that the corresponding fuzzer outperforms the other compared fuzzers on that benchmark.

For many benchmarks and comparisons, the reported $p$-values are below 0.05, indicating statistically significant differences in vulnerability discovery performance. This suggests that the observed improvements are unlikely to be caused by randomness in repeated fuzzing runs. However, for lightweight benchmarks or cases with small performance gaps, the differences are not always statistically significant.

To further understand the contribution of each component, we conducted an ablation study in the TTE setting. We first compared TriFuzz-nSE with AFLGo. The results show that TriFuzz-nSE consistently outperforms AFLGo on most benchmarks, with particularly notable improvements on *mjs* and *jhead*. This suggested that the proposed distance model, together with a more aggressive energy allocation strategy, can significantly improve fuzzing efficiency. We then compared TriFuzz with TriFuzz-Full to evaluate the effect of selective symbolic instrumentation. The results showed that TriFuzz achieves modest but consistent improvements over TriFuzz-Full on most benchmarks, which aligns with our expectation that selective symbolic instrumentation can reduce redundant test cases. However, in later fuzzing stages, TriFuzz-Full may still mitigate the impact of redundant inputs through energy allocation, thereby narrowing the performance gap. These observations indicated that the distance model provides substantial performance gains, while the benefit of selective symbolic instrumentation is relatively limited and scenario-dependent.

These findings answer **RQ2**: Overall, TriFuzz achieves favorable vulnerability-exposure performance on most evaluated benchmark programs. The ablation results further show that the proposed fine-grained probabilistic distance model provides more effective guidance than AFLGo's structural distance metric. Nevertheless, in some programs, vulnerability exposure depends not only on reaching the target site but also on satisfying specific program states at that site. Since TriFuzz mainly improves path reachability and branch-constraint solving, its advantage becomes less pronounced when vulnerability triggering is dominated by state-dependent conditions.

### *5.4. Impact of Selective Instrumentation*

To answer RQ3 and assess the effectiveness of selective compile-time instrumentation, we compared TriFuzz with a fully instrumented variant (TriFuzz–Full Instr.) across all benchmarks. The evaluation examines four dimensions: Time-To-Expose (TTE), Time-To-Reach (TTR), Redundant Test Case Variance (RTV) and Selective Instrumentation Ratio (SIR). The results are summarized in Tables 3, 4 and 6 and Figure 6. Furthermore, to quantitatively analyze the impact of selective symbolic instrumentation, we conducted an ablation study by comparing TriFuzz with TriFuzz-Full, and report the relative improvement ratios in Table 5.

TriFuzz achieves lower TTE and TTR values than the fully instrumented variant on most programs. Selectively instrumenting only the control- and data-flow-dependent regions substantially reduces symbolic execution overhead, leading to the earlier discovery of target-reaching paths and faster triggering of vulnerabilities. This improvement is most pronounced in benchmarks with complex control-flow structures where irrelevant symbolic paths constitute a large portion of the full instrumentation search space. By excluding these non-essential basic blocks, TriFuzz is able to consistently prioritize effective symbolic reasoning. However, we also observe that TriFuzz-nSE slightly outperforms TriFuzz on a small number of benchmarks. For example, in the TTR results on *LMS*, TriFuzz-nSE achieves 292.9 s, whereas TriFuzz requires 301.6 s. In this case, the SIR of TriFuzz on *LMS* reaches 66.7%, indicating that a relatively large portion of functions is still preserved for symbolic instrumentation. Our debugging analysis further shows that the test cases generated by symbolic execution do not effectively contribute to reaching the target in this benchmark. Instead, these additional test cases occupy part of the seed queue and introduce slight overhead, which ultimately leads to a minor degradation in fuzzing efficiency.

The advantage of selective symbolic instrumentation is further supported by the RTV reported in Table 6. TriFuzz produces markedly lower variance in TTE across repeated runs for nearly all benchmarks. This indicates a more stable and deterministic hybrid directed fuzzing process, with fewer redundant symbolic test cases and reduced susceptibility to randomness in execution order or search-space divergence. In contrast, full instrumentation suffers from larger RTV values, reflecting the noise introduced by exploring symbolic paths that do not contribute to reaching the target.

Combined with the experimental results and the SIR values reported in Figure 6, we observe that the instrumentation ratio has a direct impact on fuzzing performance. A larger SIR means that more functions are preserved for symbolic instrumentation, which in turn tends to generate more redundant test cases during hybrid execution and reduces overall fuzzing efficiency. In the extreme case where the SIR approaches 100%, the advantage of selective instrumentation largely disappears, and the performance of TriFuzz degrades toward that of TriFuzz-Full.

These findings answer **RQ3**: A low SIR enables TriFuzz to provide more precise path guidance by keeping the symbolically instrumented region compact and target-relevant. For example, on *pdftotext* and *jhead*, where the SIR is only 10.2% and 15.3%, respectively, TriFuzz can filter out many target-irrelevant paths, steer test cases toward the target region more efficiently, and expose vulnerabilities faster in benchmarks where the SIR is low. In contrast, a high SIR makes TriFuzz behave closer to TriFuzz-Full. On *mjs*, where the SIR reaches 99.4%, selective instrumentation provides limited pruning, introduces more redundant test cases, and consequently weakens the performance benefit of TriFuzz.

### 5.5. *Effectiveness of the Fine-Grained Probabilistic Distance Metric*

To answer RQ4 and assess the effectiveness of our fine-grained probabilistic distance metric, we conducted a comparative study between TriFuzz-nSE and the original AFLGo [1] across all benchmarks in the evaluation suite. The results, summarized in Tables 3 and 4, demonstrate that replacing AFLGo's heuristic structural distance with our probabilistic formulation yields substantial improvements in both TTE and TTR for most targets. Furthermore, to quantify the standalone contribution of the fine-grained probabilistic distance model, we report the relative improvement ratios in Table 5.

Across the majority of applications, TriFuzz-nSE consistently reduces the time to expose the target vulnerability (TTE) relative to AFLGo [1]. For instance, on *jasper*, TTE improves from 53.5 s (AFLGo) to 46.3 s (TriFuzz-nSE); on *mjs-issues-78*, the reduction is even more significant—from 8718.8 s to 5362.5 s. Highly structured programs such as

*jhead* and *mjs-issues-57* exhibit the most dramatic improvements: TriFuzz-nSE achieves speedups of 1.7× and 2.2×, respectively, highlighting the metric's ability to navigate complex control-flow regions more effectively. Even in lightweight benchmarks such as *Palindrome*, TriFuzz-nSE matches or slightly outperforms AFLGo, demonstrating that the new metric imposes no penalty on shallow targets. A similar trend is observed for TTR, where the probabilistic metric improves or closely matches AFLGo's [1] performance. For example, TriFuzz-nSE reaches the *jasper* target in 17.29 s, compared to AFLGo's [1] 18.55 s, and reduces TTR on *jhead* from 33,321.85 s to 1823.41 s, representing an improvement of over 18×. These results indicate that AFLGo's [1] original CFG-distance heuristic often misguides the fuzzer when the program contains loops, indirect calls, or unbalanced branching factors.

For shallow targets such as *Palindrome*, the two distance metrics exhibit comparable behavior, as expected. Conversely, for extremely deep targets such as *mp3gain* and *mjs-issues-57*, TriFuzz-nSE achieves consistently lower TTR and TTE values, showing that the probabilistic modeling of multi-path reachability is especially advantageous when navigating long dependency chains where structural distances alone are insufficient.

These findings answer **RQ4**: The proposed fine-grained probabilistic distance metric provides a more accurate and informative guidance signal than AFLGo's structural distance metric. By better distinguishing basic blocks with different target-reachability probabilities, the proposed metric helps the fuzzer prioritize more promising seeds, thereby improving both target reaching and vulnerability exposure. These results validate the effectiveness of the distance model as a core component of TriFuzz.

### 5.6. Distance-Calculation Overhead Optimization

To evaluate the effectiveness of the proposed distance-calculation optimizations, we measured execution time and peak memory consumption (Max RSS) before and after optimization. Table 7 summarizes the results, where T.O. denotes runs that failed due to memory exhaustion. In addition, we also include AFLGo [1] and its optimized distance-calculation version, AFLGo-fast, where the optimized version adopts a faster and more lightweight design. We compare TriFuzz with AFLGo [1] to show that, although our method provides substantially improved distance-calculation precision, it does not introduce significant performance overhead. While the optimization has negligible impact on small programs, it yields substantial improvements for large and structurally complex applications.

**Table 7.** Execution time and memory consumption of distance calculation for TriFuzz and AFLGo.

| Prog. | Time (s) | | | | Max RSS (KB) | | | |
|---|---|---|---|---|---|---|---|---|
| | TriFuzz | TriFuzz (Optimized) | AFLGo | AFLGo (Fast) | TriFuzz | TriFuzz (Optimized) | AFLGo | AFLGo (Fast) |
| jasper | 0.44 | **0.41** | 107.875 | 2.105 | **78,408** | 79,064 | 112,588 | 112,964 |
| mjs-issues-78 | 4.74 | 4.85 | 40.728 | **1.053** | **45,004** | 45,604 | 100,932 | 101,116 |
| mjs-issues-57 | 4.85 | 5.05 | 40.220 | **1.055** | **45,564** | 45,732 | 101,420 | 101,160 |
| Palindrome | ≈**0** | ≈**0** | 2.953 | 0.251 | 30,976 | 30,464 | 78,260 | 77,292 |
| jhead | 52.84 | 0.52 | 6.437 | **0.170** | 585,372 | **54,528** | 77,568 | 77,824 |
| Bento4 (mp42aac) | **15.15** | 15.25 | 405.230 | 37.450 | **106,488** | 108,912 | 299,260 | 183,232 |
| mujs | 46.69 | **0.24** | 13.200 | 1.340 | 349,952 | **48,128** | 83,029 | 84,231 |
| cflow | 0.14 | **0.13** | 45.425 | 1.178 | **45,968** | 46,140 | 75,008 | 69,836 |
| xpdf (pdftotext) | T.O. | **91.72** | 546.434 | 156.340 | T.O. | **246,444** | 362,744 | 326,836 |
| objdump | T.O. | 186.98 | 1301.534 | **18.830** | T.O. | 1,625,368 | 2,439,868 | **501,452** |
| SQLite | 113.78 | 124.93 | 389.309 | **7.569** | **147,084** | 207,864 | 302,808 | 289,060 |

Note: "T.O." indicates that the distance-calculation process failed due to excessive memory consumption, so the execution time could not be measured. Bold values indicate that the corresponding distance calculation algorithm outperforms the other compared ones on that benchmark.

For lightweight benchmarks such as *jasper*, *Palindrome*, and *cflow*, both versions show comparable performance, as these programs contain few conflict basic blocks and limited opportunities for reuse. Accordingly, runtime and memory usage remain largely unchanged. In contrast, for applications with large, deeply nested, or highly cyclic CFGs—such as *jhead*, *mujs*, *pdftotext*, and *objdump*—the benefits are pronounced. Figure 7 further shows the scale of these programs, measured by both lines of code and disk size (KB). We observe that larger programs generally require longer static-analysis time, which is consistent with the increased complexity of their control-flow structure and inter-procedural dependencies. These programs contain thousands of overlapping conflict trees, which the baseline algorithm repeatedly reconstructs via recursive DFS, leading to prohibitive memory consumption (e.g., over 1.6 GB for *objdump* and 246 MB for *pdftotext*).

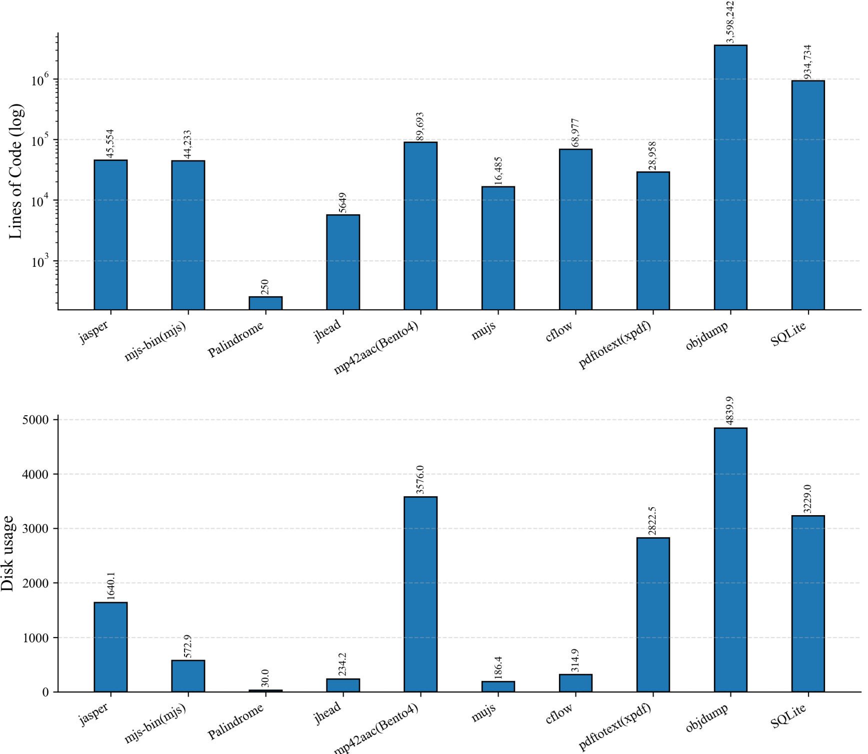


**Figure 7.** Program size.

With conflict-tree reuse, distance calculation becomes orders of magnitude more efficient. For example, *jhead* improves from 52.84 s and 585 MB to 0.52 s and 54 MB, while *mujs* drops from 46.69 s and 350 MB to 0.24 s and 48 MB. Moreover, the previous issue that caused distance calculation for *pdftotext* and *objdump* to fail due to excessive memory consumption is also resolved, and both programs can now complete static analysis within 5 min. These reductions confirm that eliminating redundant tree traversals effectively prevents exponential growth in time and space. More importantly, the results show that for most programs, TriFuzz can complete distance calculation within one minute. Only a small number of particularly complex programs require more time, and even in those cases, the computation still finishes within five minutes. Compared with the overall fuzzing campaigns, which typically run for many hours, this overhead is negligible. Therefore, the proposed distance-calculation algorithm has only a limited impact on the end-to-end fuzzing process.

Overall, the results confirm that conflict-tree reuse dramatically improves the scalability of probabilistic distance calculation, reducing both runtime and memory consumption and enabling large-scale real-world analysis.

### 5.7. Vulnerability Triggering Diversity Analysis

To further understand the effectiveness of TriFuzz in triggering target vulnerabilities, we analyzed the crashing test cases using *gdb* [46] and examined their execution backtraces. We group crashing inputs based on their normalized backtrace patterns and use the number of distinct backtraces as a proxy for failure-path diversity.

Figure 8 shows the number of unique crashes generated by TriFuzz and AFLGo on each benchmark. Overall, TriFuzz produces more unique crashes than AFLGo on most benchmarks, indicating that it is able to explore a broader set of failure-triggering executions. The advantage is particularly pronounced on *mjs-issues-78*, *mjs-issues-57*, *Palindrome*, and *jhead*, where TriFuzz yields substantially more unique crashes. Although AFLGo achieves a slightly higher count on *LMS*, the overall trend still supports that TriFuzz can generate more diverse crashing behaviors on most evaluated programs.

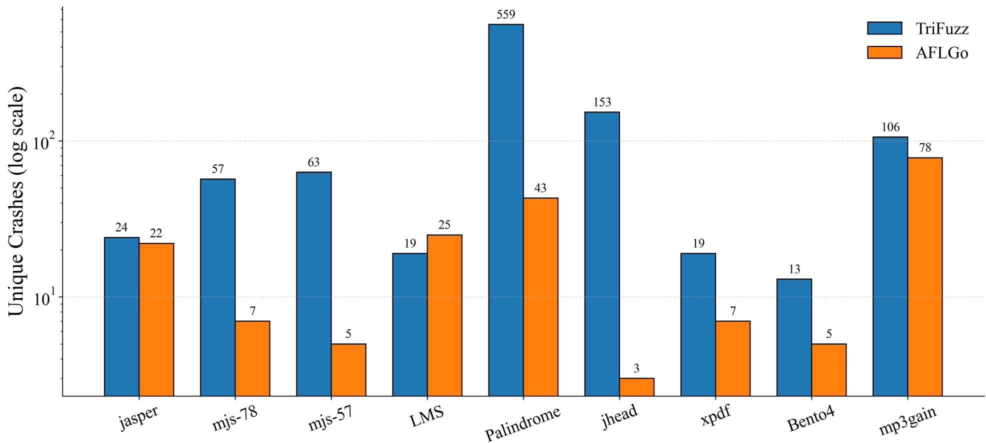


**Figure 8.** Comparison of unique crashes.

For *LMS*, however, TriFuzz shows a different behavior. Although it achieves a relatively clear advantage in TTE, its performance in TTR is less remarkable, which also helps explain why its crash diversity is lower than that of AFLGo on this benchmark. Our debugging analysis suggests that the vulnerability in *LMS* is triggered through a relatively narrow set of target-related program states. As a result, once TriFuzz is guided into this region, its selective symbolic execution tends to repeatedly generate inputs that satisfy similar constraints and thus converge to a limited number of crash patterns. In contrast, AFLGo [1] relies more heavily on unguided mutation, which may explore a broader set of nearby executions and accidentally expose slightly more diverse crashes, even though its overall vulnerability-triggering efficiency remains lower.

The advantage of TriFuzz in vulnerability-path diversity can be attributed to the integration of selective symbolic execution: once TriFuzz identifies a crashing input, SymCC [18] is invoked to explore alternative feasible paths by solving nearby branch constraints. Even small variations in symbolic inputs can lead to different execution paths that still trigger the same bug. However, when the vulnerability can only be triggered through a relatively narrow set of execution paths, mutation-based strategies may exhibit a greater advantage, since random mutations can more naturally explore local input variations without being biased toward a small set of similar symbolic solutions.

We note that this analysis is based on backtrace patterns as a lightweight proxy and does not fully capture all aspects of path diversity, but it provides practical insight into the diversity of failure-inducing inputs.

## 6. Discussion

### 6.1. Limitations

Despite the significant improvements achieved by TriFuzz over AFLGo [1] and other baselines, several limitations remain. First, as discussed in Section 5.1.2, our evaluation on AFLGo's testsuite [1] and UniBench [34] necessarily excludes a subset of real-world programs. This restriction mainly stems from the limitations of SymCC [18]. In practice, SymCC only provides wrappers for a subset of important C standard library functions, rather than fully modeling all *libc* functions or complex system-level library behaviors. When execution flows into uninstrumented code or library functions without symbolic wrappers, SymCC [18] treats the corresponding results as concrete values, which may break the symbolic propagation chain and prevent further constraint solving. As a result, the symbolic execution component of TriFuzz may become ineffective for programs that heavily depend on such unsupported libraries. This design reflects a practical trade-off in SymCC [18]: by performing symbolic instrumentation at compile time, it achieves substantially lower runtime overhead than dynamic-binary-translation-based systems such as QSYM [21], but its applicability is correspondingly constrained by library support. Therefore, the generality of our current implementation is primarily limited by the incomplete support of SymCC [18] for external library functions. Nevertheless, since SymCC [18] still supports a large portion of commonly used functions in practice, this limitation does not fundamentally undermine the feasibility of our approach.

Second, although we introduce a fine-grained probability-based distance metric, the current formulation assigns uniform probabilities to successor branches rather than estimating branch likelihood based on semantic constraints. For instance, under the predicate `a > 0`, assigning equal probabilities to the true and false branches is reasonable. However, for predicates such as `a == 0xDEADBEEF`, the true branch is exceedingly unlikely to be taken during mutation-based fuzzing. Our current model cannot differentiate between these cases and thus treats both successors identically, resulting in a distance signal that is structurally meaningful but not fully reflective of real mutation likelihoods. Incorporating branch-condition-aware probability estimation remains an important direction for future work.

Finally, the effectiveness of selective symbolic instrumentation depends on both program complexity and the precision of the underlying inter-procedural static analysis. TriFuzz currently relies on conservative pointer analysis, data-flow analysis, and indirect-call recovery to identify target-related functions and basic blocks. For large-scale programs or C++-heavy codebases with extensive virtual calls, callbacks, and function pointers, such analysis may over-approximate possible callees, data dependencies, and call-chain relationships. As a result, selective symbolic instrumentation may conservatively include a broader set of control-flow- or data-flow-related functions, which enlarges the selected instrumentation region and increases the Selective Instrumentation Ratio (SIR). When SIR becomes high, the selected region gradually approaches full instrumentation, and the benefit of selection is weakened because SymCC may still generate many redundant test cases from target-irrelevant or weakly relevant paths. This limitation is consistent with our RQ3 observation that TriFuzz is most effective when SIR remains low, while its advantage becomes less pronounced when the selected region becomes large. Improving pointer-analysis precision, indirect-call resolution, and demand-driven refinement is therefore a promising direction for reducing such over-approximation and further improving the effectiveness of selective symbolic instrumentation.

Nevertheless, our experimental results show that TriFuzz still performs well on many large-scale programs. In several such programs, the Selective Instrumentation Ratio (SIR) remains low, indicating that our method is already able to approximate target-related

execution paths reasonably well. This allows TriFuzz to progressively steer fuzzing toward the target code and eventually trigger the vulnerability with good effectiveness.

### *6.2. Future Work*

Several promising directions remain for extending TriFuzz beyond its current design. The probability-based distance metric can be further refined through more accurate modeling of branch predicates. As discussed earlier, the current formulation assigns uniform probabilities to successor branches due to the absence of semantic reasoning about branch conditions. Incorporating branch-condition analysis to derive more realistic probability values would allow the model to better capture the likelihood of different execution paths under mutation-based fuzzing. This is particularly relevant for skewed predicates, such as equality checks and constant comparisons, and would improve both the stability and precision of distance estimation.

The experimental results also suggest that seed mutation quality plays a critical role in reaching target locations and triggering vulnerabilities. While TriFuzz enhances guidance through probabilistic distance modeling and selective symbolic execution, the mutation strategy largely follows AFLGo [1]. Exploring more target-aware mutation strategies that focus on input bytes influencing control-flow-relevant conditions may further improve performance. Potential directions include gradient-inspired feedback, adaptive byte-level importance estimation, and symbolic-assisted mutation synthesis, which can be naturally integrated into the current hybrid fuzzing framework.

Another important direction is a more comprehensive comparison with closely related hybrid fuzzing approaches, such as HyperGo [30]. Although HyperGo [30] also leverages probability-based guidance and an Optimized Symbolic Execution Complementary (OSEC) scheme, differences in implementation details and evaluation configurations make it non-trivial to ensure a fair and reproducible comparison under a unified setup. Establishing such a controlled evaluation environment would enable a clearer understanding of the trade-offs between different probability-guided hybrid fuzzing designs.

Finally, the diversity of paths and inputs produced by TriFuzz suggests potential applications beyond vulnerability detection. In particular, diverse vulnerability-triggering test cases may benefit downstream security tasks such as exploit development and automated exploit generation (AEG) [47–52]. Since many AEG techniques rely on symbolic execution [18,21,24,25] and taint analysis [53,54] to construct viable attack chains, providing multiple semantically distinct inputs that trigger the same vulnerability may improve both effectiveness and robustness. Exploring this integration would move TriFuzz toward a more complete vulnerability discovery and exploitation pipeline.

## 7. Conclusions

In this paper, we present **TriFuzz**, a hybrid directed fuzzing framework that improves target-oriented vulnerability discovery. Our approach enhances directed fuzzing by providing more effective guidance toward specific locations. Experimental results on standard directed fuzzing benchmarks show that TriFuzz consistently accelerates both target reachability and vulnerability triggering compared with existing approaches. In addition, TriFuzz demonstrates improved robustness across different programs and produces more diverse failure-inducing inputs. These results indicate that combining refined guidance with symbolic execution and introducing fine-grained probability-based distance calculation can effectively improve the performance of directed fuzzing in practice.

**Author Contributions:** Conceptualization, Y.L. and Y.W.; methodology, Y.W.; software, Y.W.; validation, Y.W. and W.Q.; formal analysis, Y.W.; investigation, Y.W. and W.Q.; resources, Y.W.; data curation, Y.W.; writing—original draft preparation, Y.W.; writing—review and editing, R.F. and J.L.;

visualization, Y.W.; supervision, R.F.; project administration, Y.L.; funding acquisition, W.Q. All authors have read and agreed to the published version of the manuscript.

**Funding:** This research was funded by the Natural Science Foundation of Jiangsu Province (No. BK20230134) and Shanghai Automotive Industry Science and Technology Development Foundation (No. 2511).

**Institutional Review Board Statement:** Not applicable.

**Informed Consent Statement:** Not applicable.

**Data Availability Statement:** Dataset available on request from the authors.

**Conflicts of Interest:** The authors declare no conflicts of interest.

## References

1. Böhme, M.; Pham, V.-T.; Nguyen, M.-D.; Roychoudhury, A. Directed Greybox Fuzzing. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017; pp. 2329–2344. [CrossRef]
2. Canakci, S.; Matyunin, N.; Graffi, K.; Joshi, A.; Egele, M. Targetfuzz: Using darts to guide directed greybox fuzzers. In Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security, Nagasaki, Japan, 30 May–3 June 2022; pp. 561–573. [CrossRef]
3. Tan, X.; Zhang, Y.; Lu, J.; Xiong, X.; Liu, Z.; Yang, M. Syzdirect: Directed greybox fuzzing for linux kernel. In Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, Copenhagen, Denmark, 26–30 November 2023; pp. 1630–1644. [CrossRef]
4. Wang, P.; Zhou, X.; Yue, T.; Lin, P.; Liu, Y.; Lu, K. The progress, challenges, and perspectives of directed greybox fuzzing. *Softw. Test. Verif. Reliab.* **2024**, *34*, e1869. [CrossRef]
5. Geretto, E.; Jemmett, A.; Giuffrida, C.; Bos, H. LibAFLGo: Evaluating and Advancing Directed Greybox Fuzzing. In Proceedings of the 2025 IEEE 10th European Symposium on Security and Privacy (EuroS&P), Venice, Italy, 30 June–4 July 2025; pp. 355–373. [CrossRef]
6. Kim, T.; Choi, J.; Heo, K.; Cha, S. DAFL: Directed Grey-box Fuzzing guided by Data Dependency. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 4931–4948.
7. Cao, S.; He, B.; Sun, X.; Ouyang, Y.; Zhang, C.; Wu, X.; Su, T.; Bo, L.; Li, B.; Ma, C.; et al. Oddfuzz: Discovering java deserialization vulnerabilities via structure-aware directed greybox fuzzing. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2023; pp. 2726–2743. [CrossRef]
8. Bao, A.; Zhao, W.; Wang, Y.; Cheng, Y.; McCamant, S.; Yew, P. From Alarms to Real Bugs: Multi-target Multi-step Directed Greybox Fuzzing for Static Analysis Result Verification. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 6977–6997.
9. Lin, Z.; Zhang, Y.; Dai, J.; Huang, X.; Xiang, B.; Yang, G.; Yuan, L.; Zhang, L.; Chen, T.; Yang, M. Effective directed fuzzing with hierarchical scheduling for web vulnerability detection. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 8349–8366.
10. Deng, P.; Zhang, L.; Meng, Y.; Yang, Z.; Zhang, Y. ChainFuzz: Exploiting Upstream Vulnerabilities in Open-Source Supply Chains. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 6199–6218.
11. Liang, H.; Yu, X.; Cheng, X.; Liu, J.; Li, J. Multiple targets directed greybox fuzzing. *IEEE Trans. Dependable Secur. Comput.* **2024**, *21*, 325–339. [CrossRef]
12. Xiang, Y.; Zhang, X.; Liu, P.; Ji, S.; Liang, H.; Xu, J.; Wang, W. Critical code guided directed greybox fuzzing for commits. In Proceedings of the 33rd USENIX Security Symposium (USENIX Security 24), Philadelphia, PA, USA, 14–16 August 2024; pp. 2459–2474.
13. Österlund, S.; Razavi, K.; Bos, H.; Giuffrida, C. ParmeSan: Sanitizer-guided greybox fuzzing. In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 2289–2306.
14. Fang, H.; Zhang, K.; Yu, D.; Zhang, Y. DDGF: Dynamic Directed Greybox Fuzzing with Path Profiling. In Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis, Vienna, Austria, 16–20 September 2024; pp. 832–843. [CrossRef]
15. Li, R.; Liang, H.; Liu, L.; Ma, X.; Qu, R.; Yan, J.; Zhang, J. GTFuzz: Guard token directed grey-box fuzzing. In Proceedings of the 2020 IEEE 25th Pacific Rim International Symposium on Dependable Computing (PRDC), Perth, Australia, 1–4 December 2020; pp. 160–170. [CrossRef]

16. Zalewski, M. American Fuzzy Lop (AFL) Fuzzer. 2015; p. 33. Available online: http://lcamtuf.coredump.cx/afl/ (accessed on 5 December 2024).
17. Fioraldi, A.; Maier, D.; Eißfeldt, H.; Heuse, M. AFL++: Combining incremental steps of fuzzing research. In Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT 20), Virtual Event, 11 August 2020.
18. Poeplau, S.; Francillon, A. Symbolic execution with SymCC: Don't interpret, compile! In Proceedings of the 29th USENIX Security Symposium (USENIX Security 20), Boston, MA, USA, 12–14 August 2020; pp. 181–198.
19. Cadar, C.; Dunbar, D.; Engler, D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI'08), San Diego, CA, USA, 8–10 December 2008; pp. 209–224.
20. Chipounov, V.; Georgescu, V.; Zamfir, C.; Candea, G. Selective symbolic execution. In Proceedings of the 5th Workshop on Hot Topics in System Dependability, Lisbon, Portugal, 29 June 2009.
21. Yun, I.; Lee, S.; Xu, M.; Jang, Y.; Kim, T. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Security Symposium (USENIX Security 18), Baltimore, MD, USA, 15–17 August 2018; pp. 745–761.
22. Shoshitaishvili, Y.; Wang, R.; Salls, C.; Stephens, N.; Polino, M.; Dutcher, A.; Grosen, J.; Feng, S.; Hauser, C.; Kruegel, C.; et al. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In Proceedings of the 2016 IEEE Symposium on Security and Privacy, San Jose, CA, USA, 22–26 May 2016; pp. 138–157. [CrossRef]
23. Baldoni, R.; Coppa, E.; D'elia, D.; Demetrescu, C.; Finocchi, I. A survey of symbolic execution techniques. *ACM Comput. Surv. (CSUR)* **2018**, *51*, 1–39. [CrossRef]
24. Sen, K. Concolic testing. In Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering, Atlanta, GA, USA, 5–9 November 2007; pp. 571–572. [CrossRef]
25. Chen, J.; Han, W.; Yin, M.; Zeng, H.; Song, C.; Lee, B.; Yin, H.; Shin, I. SYMSAN: Time and space efficient concolic execution via dynamic data-flow analysis. In Proceedings of the 31st USENIX Security Symposium (USENIX Security 22), Boston, MA, USA, 10–12 August 2022; pp. 2531–2548.
26. Stephens, N.; Grosen, J.; Salls, C.; Dutcher, A.; Wang, R.; Corbetta, J.; Shoshitaishvili, Y.; Kruegel, C.; Vigna, G. Driller: Augmenting fuzzing through selective symbolic execution. In Proceedings of the NDSS, San Diego, CA, USA, 21–24 February 2016; Volume 16, pp. 1–16. [CrossRef]
27. Zhao, L.; Duan, Y.; Xuan, J. Send hardest problems my way: Probabilistic path prioritization for hybrid fuzzing. In Proceedings of the Network and Distributed System Security Symposium (NDSS), San Diego, CA, USA, 24–27 February 2019. [CrossRef]
28. Kim, K.; Jeong, D.; Kim, C.; Jang, Y.; Shin, I.; Lee, B. HFL: Hybrid Fuzzing on the Linux Kernel. In Proceedings of the NDSS, San Diego, CA, USA, 23–26 February 2020.
29. Li, J.; Shen, J.; Su, Y.; Lyu, M. ColorGo: Directed Concolic Execution. *arXiv* **2025**, arXiv:2505.21130. [CrossRef]
30. Lin, P.; Wang, P.; Zhou, X.; Xie, W.; Lu, K.; Zhang, G. HyperGo: Probability-based directed hybrid fuzzing. *Comput. Secur.* **2024**, *142*, 103851. [CrossRef]
31. Yang, Y.; Yao, S.; Chen, J.; Lee, W. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 6299–6318.
32. Liang, H.; Jiang, L.; Ai, L.; Wei, J. Sequence directed hybrid fuzzing. In Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), London, ON, Canada, 18–21 February 2020; pp. 127–137. [CrossRef]
33. Luo, C.; Meng, W.; Li, P. Selectfuzz: Efficient directed fuzzing with selective path exploration. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2023; pp. 2693–2707. [CrossRef]
34. Li, Y.; Ji, S.; Chen, Y.; Liang, S.; Lee, W.; Chen, Y.; Lyu, C.; Wu, C.; Beyah, R.; Cheng, P.; et al. UNIFUZZ: A holistic and pragmatic Metrics-Driven platform for evaluating fuzzers. In Proceedings of the 30th USENIX Security Symposium (USENIX Security 21), Vancouver, BC, Canada, 11–13 August 2021; pp. 2777–2794.
35. Du, Z.; Li, Y.; Liu, Y.; Mao, B. Windranger: A directed greybox fuzzer driven by deviation basic blocks. In Proceedings of the 44th International Conference on Software Engineering, Pittsburgh, PA, USA, 25–27 May 2022; pp. 2440–2451. [CrossRef]
36. Huang, H.; Guo, Y.; Shi, Q.; Yao, P.; Wu, R.; Zhang, C. Beacon: Directed grey-box fuzzing with provable path pruning. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 36–50. [CrossRef]
37. Shah, A.; She, D.; Sadhu, S.; Singal, K.; Coffman, P.; Jana, S. Mc2: Rigorous and efficient directed greybox fuzzing. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, Los Angeles, CA, USA, 7–11 November 2022; pp. 2595–2609. [CrossRef]
38. Zheng, H.; Zhang, J.; Huang, Y.; Ren, Z.; Wang, H.; Cao, C.; Zhang, Y.; Toffalini, F.; Payer, M. FISHFUZZ: Catch deeper bugs by throwing larger nets. In Proceedings of the 32nd USENIX Security Symposium (USENIX Security 23), Anaheim, CA, USA, 9–11 August 2023; pp. 1343–1360.
39. Lin, P.; Wang, P.; Zhou, X.; Xie, W.; Zhang, G.; Lu, K. Deepgo: Predictive directed greybox fuzzing. *arXiv* **2025**, arXiv:2507.21952. [CrossRef]

40. Chen, Y.; Zhang, C.; Wang, L.; Zhu, W.; Luo, C.; Gui, N.; Ma, Z.; Zhang, X.; Su, B. IDFuzz: Intelligent Directed Grey-box Fuzzing. In Proceedings of the 34th USENIX Security Symposium (USENIX Security 25), Seattle, WA, USA, 13–15 August 2025; pp. 6219–6238.
41. Lattner, C.; Adve, V. LLVM: A compilation framework for lifelong program analysis & transformation. In Proceedings of the International Symposium on Code Generation and Optimization, 2004 (CGO 2004), San Jose, CA, USA, 20–24 March 2004; pp. 75–86. [CrossRef]
42. Chen, J.; Wang, J.; Song, C.; Yin, H. Jigsaw: Efficient and scalable path constraints fuzzing. In Proceedings of the 2022 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 22–26 May 2022; pp. 18–35. [CrossRef]
43. Peng, J.; Li, F.; Liu, B.; Xu, L.; Liu, B.; Chen, K.; Huo, W. 1dvul: Discovering 1-day vulnerabilities through binary patches. In Proceedings of the 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), Portland, OR, USA, 24–27 June 2019; pp. 605–616. [CrossRef]
44. Sui, Y.; Xue, J. SVF: Interprocedural static value-flow analysis in LLVM. In Proceedings of the 25th International Conference on Compiler Construction, Barcelona, Spain, 12–18 March 2016; pp. 265–266. [CrossRef]
45. Vargha, A.; Delaney, H. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *J. Educ. Behav. Stat.* **2000**, *25*, 101–132. [PubMed]
46. Stallman, R.; Pesch, R.; Shebs, S. *Debugging with GDB*; Free Software Foundation: Boston, MA, USA, 1988; 675p.
47. Avgerinos, T.; Cha, S.; Rebert, A.; Schwartz, E.; Woo, M.; Brumley, D. Automatic exploit generation. *Commun. ACM* **2014**, *57*, 74–84. [CrossRef]
48. Wu, Y.; Li, Y.; Zhu, H.; Zhang, Y. SAEG: Stateful Automatic Exploit Generation. In Proceedings of the European Symposium on Research in Computer Security, Bydgoszcz, Poland, 16–20 September 2024; pp. 127–145. [CrossRef]
49. Bui, Q.; Iannone, E.; Camporese, M.; Hinrichs, T.; Tony, C.; Tóth, L.; Palomba, F.; Hegedűs, P.; Massacci, F.; Scandariato, R. A Systematic Literature Review on Automated Exploit and Security Test Generation. *arXiv* **2025**, arXiv:2502.04953. [CrossRef]
50. Dixit, S.; Geethna, T.; Jayaraman, S.; Pavithran, V. Angerza: Automated exploit generation. In Proceedings of the 2021 12th International Conference on Computing Communication and Networking Technologies (ICCCNT), Kharagpur, India, 6–8 July 2021; pp. 1–6. [CrossRef]
51. Jin, L.; Cao, Y.; Chen, Y.; Zhang, D.; Campanoni, S. Exgen: Cross-platform, automated exploit generation for smart contract vulnerabilities. *IEEE Trans. Dependable Secur. Comput.* **2023**, *20*, 650–664. [CrossRef]
52. Alhuzali, A.; Eshete, B.; Gjomemo, R.; Venkatakrishnan, V. Chainsaw: Chained automated workflow-based exploit generation. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, 24–28 October 2016; pp. 641–652. [CrossRef]
53. Clause, J.; Li, W.; Orso, A. Dytan: A generic dynamic taint analysis framework. In Proceedings of the 2007 International Symposium on Software Testing and Analysis, London, UK, 9–12 July 2007; pp. 196–206. [CrossRef]
54. Newsome, J.; Song, D. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In Proceedings of the NDSS, San Diego, CA, USA, 3–4 February 2005; Volume 5, pp. 3–4.

**Disclaimer/Publisher's Note:** The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.