When Control Flows Deviate: Directed Grey-box Fuzzing with Probabilistic Reachability Analysis

Peihong Lin, Pengfei Wang✉, Xu Zhou, Wei Xie, Xin Ren, Kai Lu✉
National University of Defense Technology
{phlin22, pfwang, zhouxu, xiewei, renxin, kailu}@nudt.edu.cn

Abstract*—Directed grey-box fuzzing (DGF) steers testing toward high-value targets, but developing effective DGF for commercial off-the-shelf (COTS) binaries is challenging due to the lack of accurate structural information (e.g., control-flow graphs and call graphs), which can cause control flows to deviate and misguide DGF's reachability analysis. In this paper, we introduce BinGo, a tailored binary-level directed grey-box fuzzer, which can accommodate the flawed control-flow graphs (CFGs) of COTS binaries and enable accurate and efficient reachability analysis. First, to quantify the inevitable inaccuracies of uncovered indirect edges and analyze their impact on the reachability of basic blocks, we propose a Bayesian-based method. This method combines prior knowledge from static analysis with dynamic observations from fuzzing to estimate the confidence in correctly recovering indirect edges. Then, we present a new concept called a *region*, which redefines granularity for efficient reachability analysis by transforming the CFG into a region graph. Using the Bayesian results and region graph, we propose a custom fitness metric for binary-level DGF, termed *probabilistic reachability*. This metric, based on a dynamically updated region graph and reachability scores, is adaptive, lightweight, and accommodates inaccurate binary-level CFGs. We implemented a prototype tool, BinGo, and evaluated it on the CGC dataset, CVE-Benchmark, and UniBench benchmark. Experimental results show that BinGo surpasses baseline fuzzers (AFL++, AFLGo, PDGF, UAFuzz, and 1dVul) in reaching target locations and exposing known vulnerabilities. Additionally, BinGo discovered three new vulnerabilities in the real-world application *cscope-15.9*.

***Index Terms*—Software Security, Directed Grey-box Fuzzing, Region, Probabilistic Reachability**

I. Introduction

Directed Grey-box Fuzzing (DGF) [9], [11], [13], [23], [27] specializes in testing high-value target program locations. It has been widely used in patch testing [41], bug reproduction [20], [30], [39], and validation [38], [40]. Most existing DGF techniques are designed for open-source programs. They rely on accurate structural information, such as control-flow graphs (CFGs) and call graphs (CGs) from the program under test (PUT) to develop fitness metrics to guide the testing directions [11]. Recent DGF tools further analyze the CFG and CG to determine the reachability, i.e., *whether a basic block lies on the paths leading to the targets, allowing fuzzers to improve efficiency by focusing on the subset of reachable code*. For instance, Beacon [18] prunes unreachable paths, SelectFuzz [28] selectively instruments reachable basic blocks.

Commercial off-the-shelf (COTS) binaries have become the predominant form of software distribution, with Gartner's 2023 report indicating that over 70% of enterprise software purchases are COTS [1]. However, structural information critical for reachability analysis in DGF, such as accurate CFGs and CGs, is typically unavailable in COTS binaries. Existing reverse engineering tools, such as IDA Pro [4] and Ghidra [6], face substantial difficulties in resolving the targets of indirect calls and jumps in these binaries, which results in incomplete CFGs that miss many indirect edges. To recover indirect edges, state-of-the-art static analysis techniques, which primarily rely on limited propagation [34], pattern matching [36], and machine learning [45], have been developed. Despite these advancements, static approaches generally achieve less than 50% precision in recovering indirect edges on average [45], leading to inaccurate CFGs. Such inaccuracies make general fitness metrics (e.g., distance-based metrics) and strategy optimizations (e.g., path pruning [18]) unreliable for binary-level DGF. Since accurately recovering indirect edges at the binary level remains an open challenge, this paper aims to design a tailored directed grey-box fuzzer that can accommodate flawed CFGs of COTS binaries.

Challenge: How to perform an accurate and efficient reachability analysis for DGF with a flawed binary-level CFG? First, existing DGF fitness metrics do not apply to flawed CFGs of COTS binaries, as they generally assume accurate structural information without missing or incorrect indirect edges. This often misguides fuzzers, resulting in wasted effort on infeasible paths or missing critical paths. Second, some source code-based fuzzing techniques, such as ParmeSan [32] and PDGF [43], attempt to update the CFGs and reachability information at runtime after executing indirect edges. However, only a small portion (less than 10%, §V-C) of target-reachable indirect edges can be covered, and the impact of uncovered indirect edges cannot be quantitatively assessed. Furthermore, since most fuzzers rely on dynamic binary translation for accuracy and scalability, these runtime updates incur significant runtime overhead at the binary level. Although lighter methods such as static binary rewriting [14], [29] exist, they often struggle with complex dependencies or cross-platform binaries. Consequently, updating reachability for numerous basic blocks and paths in binary-level fuzzers remains costly. Third, DGF approaches [26], [31], [41] designed for binaries are mainly based on specific vulnerability characteristics (e.g., UAFuzz [31] for the use-after-free vulnerability) or program features (e.g., 1dFuzz [41] for patch features) to improve the efficiency of testing certain code locations or paths

Pengfei Wang✉ and Kai Lu✉ are the coresponding authors.

in binary-level DGF. They have not addressed the impact of binary-level CFG inaccuracies on the reachability analysis.

Key Insight. To enable accurate and efficient reachability guidance for binary-level DGF, our key insight is to design a novel reachability analysis method that remains effective even with flawed CFGs. Specifically, we first aim to design an indirect edge assessment method to quantify the inevitable inaccuracies when recovering indirect edges from COTS binaries, and then analyze how these uncertain edges affect the reachability of basic blocks. Based on this, we explicitly incorporate the quantified results and the analyzed impact of uncertain indirect edges into the analysis process, forming an efficient reachability analysis method. This method can leverage a customized fitness metric that accounts for binary-level CFG inaccuracies in reachability guidance and provides adaptive guidance for binary-level fuzzing.

Solutions. To realize our insight, we first use a static method to recover indirect edges and generate prior knowledge in the form of a matching score for each recovery. Second, we employ a Bayesian-based method, which combines prior knowledge from static analysis with dynamic statistics from fuzzing, to estimate the confidence (i.e., accuracy probability) in correctly recovering indirect edges. Third, we introduce a new concept called a **region**, which is defined as *a group of basic blocks that are influenced by the same set of indirect edges*, to enable efficient runtime updates. Regions are dynamically updated when fuzzing explores new indirect edges. Based on estimated accuracy probabilities and region abstraction, we design a region-based reachability analysis to efficiently update the reachability of basic blocks across different regions. Specifically, we propose intra-region depth to measure the difficulty of traversing within a region, and inter-region connectivity to measure the challenge of transitioning between regions. These two factors are combined to form our customized fitness metric, ***probabilistic reachability***. The metric not only guides seed prioritization but also optimizes both energy allocation and mutation strategies. As fuzzing progresses, both the Bayesian model and the region structure are dynamically updated, ensuring efficient and adaptive directed fuzzing despite flawed CFGs. In summary, we make the following contributions:

- We propose a ***Bayesian-based*** method, which combines prior knowledge from static analysis with runtime fuzzing information to quantify the inevitable inaccuracies in uncovered indirect edges.
- We introduce the concept of a ***region*** at the binary level, which captures the impact of uncovered indirect edges on basic blocks, thus enabling efficient reachability analysis.
- We design a customized fitness metric for binary-level DGF, called ***probabilistic reachability***, to accommodate flawed CFG in binary-level DGF.
- We implemented a tool named BinGo and evaluated BinGo on the CGC dataset, the UniBench benchmark, and the CVE-Benchmark, with a total of 194 targets. Experimental results show that BinGo reached more target locations, exposed known vulnerabilities faster, and achieved up to a 2.47× speedup over baseline fuzzers (AFL++, AFLGo, PDGF, UAFuzz, and 1dVul). Furthermore, BinGo discovered three new vulnerabilities within 24 hours.
- The artifact is available at https://anonymous.4open.science/r/BinGo-BC54.

II. Background and Motivation

A. Background

Directed Grey-box Fuzzing. Following AFLGo [11], most existing DGF works construct CGs and CFGs for PUTs to build interprocedural CFGs (ICFGs) during the static analysis phase. These works assume that the ICFG is accurate and build distance metrics upon it, which are then further refined using additional indicators [13], [15], [23] to optimize the fitness metric. A recent promising approach optimizes DGF through *reachability analysis*, focusing fuzzing efforts on the necessary program paths that trigger vulnerabilities [18], [28], [43], [44]. This is achieved by determining whether a program path exists between a given basic block and the target basic blocks based on an accurate ICFG. Only a limited number of studies have investigated DGF for COTS binaries, and most of them are designed for specific vulnerabilities or scenarios like use-after-free or patch testing [26], [31], [41]. Some works employ QEMU's dynamic translation [5], compiler-quality instrumentation [29], or binary rewriting techniques [14] to facilitate and accelerate binary-only fuzzing.

Bayesian Statistical Model. Bayesian statistics is a probabilistic inference method based on Bayes' theorem, which updates the probability distribution of a given event. It combines prior knowledge with data to update beliefs about unknown parameters, providing flexibility in managing uncertainty and incorporating external information. The theorem is expressed:

$$P(\theta|D) = \frac{P(D|\theta)P(\theta)}{P(D)} \tag{1}$$

where $P(\theta|D)$ is the posterior distribution of parameter θ given data D, $P(D|\theta)$ is the likelihood function, $P(\theta)$ is the prior distribution of θ, and $P(D)$ is the marginal likelihood.

B. Motivation

Indirect edges, including *indirect calls* and *indirect jumps*, are highly prevalent in real-world binaries and play a critical role in determining program reachability. Our analysis of large-scale benchmarks (e.g., UniBench and CVE-Benchmark) shows that indirect edges constitute a significant proportion of control-flow transitions—64 out of 68 binaries in our benchmark contain thousands, and in some cases, even hundreds of thousands of such edges, which often guard access to security-critical targets. A typical challenge caused by indirect edges in binary-level DGF is that both static and dynamic analysis methods struggle to accurately resolve the true targets of indirect edges, leading to flawed CFGs. This, in turn, undermines the effectiveness of reachability analysis and guidance in binary-level DGF.

We illustrate this using a code snippet in Figure 1. In this example, the only feasible path ($path_1$) to the target

```c
void step2(){ target();}
void step1() {
  step2(); // To target
}

void unreachable() {
  void (*fp)() = mis_target();
  if (fp) fp(); // Static analysis incorrectly adds edge
  ↪  unreachable -> target
}

int main(int argc, char *argv[]) {
  int sel = atoi(argv[1]);
  if (sel == 0) step1(); // Path: main -> step1 -> target
  else unreachable(); // No path to target
}
```

Figure 1: Impact of inaccurate indirect edge recovery on DGF.

function `target` is `main` $\rightarrow$ `step1` $\rightarrow$ `step2` $\rightarrow$ `target`, involving three function calls. The function `unreachable` contains an indirect call that can only reach `mis_target`, which is unrelated to the fuzzing objective. However, state-of-the-art static analysis tools, such as CALLEE [45], BAP [21], and CodeSurfer [8], may resolve incorrect indirect call targets. Our empirical evaluation confirms that while state-of-the-art static methods can achieve high recall rates (over 90%), their precision rates are often below 50% [45]. This is mainly because they infer possible targets without accurate symbol and type information, which can lead to recovering non-existent edges. For instance, they may mistakenly resolve the indirect call in `unreachable` as potentially reaching `target`, introducing an incorrect edge (`unreachable`$\rightarrow$ `target`) in the recovered CFG. This misrepresentation directly impacts DGF. Guidance mechanisms, such as AFLGo's distance metrics, may then compute a path that appears shorter but is actually infeasible ($path_2$: `main` $\rightarrow$ `unreachable` $\rightarrow$ `target`) than the only real path $path_1$. As a result, the fuzzer may prioritize an unreachable path, wasting resources and missing the valid target path. Moreover, this problem cannot be addressed by existing dynamic fuzzing approaches (e.g., ParmeSan [32] and PDGF [43]) alone, since it only discovers indirect edges executed at runtime, leaving theoretically feasible but rarely triggered paths unexplored. For example, in our evaluation (§V-C), the binary-only version of PDGF only recovered 652 out of 7,462 target-reachable indirect edges, highlighting the severe limitations of dynamic discovery. For the large number of uncovered indirect edges, existing dynamic fuzzing approaches cannot determine whether these edges are correctly recovered, nor quantify their impact on the reachability of basic blocks affected by such indirect edges.

In summary, the inaccurate recovery of indirect edges at the binary level leads to persistent uncertainty in reachability analysis, fundamentally limiting the effectiveness of DGF. This motivates the need for new solutions that accommodate flawed CFG when guiding directed fuzzing in practice.

III. The Design of BinGo

A. Overview

In this paper, we design BinGo, a novel binary-level DGF approach for COTS binaries. We first apply the static method CALLEE [45] to resolve potential targets of indirect calls and generate prior knowledge that provides a matching score for each recovery. Then, we quantify the accuracy of recovered indirect edges by a Bayesian-based statistical model(§III-B). Then, we introduce the concept of ***region***, which serves as a new granularity for transforming the CFG into a region graph to facilitate efficient reachability analysis. Using the statistical results and region graph, we propose the region-based reachability analysis method (§III-C) and a new fitness metric called probabilistic reachability (§III-D), which maintains the accuracy of reachability analysis while minimizing overhead.

The overview of BinGo is depicted in Figure 2, which consists of two phases. In the static analysis phase, BinGo uses IDA Pro [4] and CALLEE [45] to construct CFG and recover indirect edges. Then, BinGo groups basic blocks into different regions and constructs the region graph. In the fuzzing phase, BinGo collects the execution frequency of nodes in the region graph (**Execution Info Collection Module**) and provides execution frequency to the **Indirect Edge Assessment Module** to estimate the confidence (i.e., accuracy probability) of accurately recovering indirect edges. Then, based on the region graph, BinGo continuously performs dynamic reachability analysis (**Region-based Reachability Analysis Module**). It updates reachability scores of all basic blocks (**Fitness Metric Calculation Module**) according to evolving accuracy probabilities, thereby optimizing the fuzzing process (**Fuzzing Optimization Module**).

B. Uncertainty-Aware Indirect Edge Assessment

We use the Bayesian statistical model to combine prior knowledge from static analysis with runtime fuzzing information to estimate the confidence in correctly recovering indirect edges. Compared to models like logistic regression or naive Bayes, the Bayesian approach is well-suited to fuzzing scenarios with sparse, incomplete, and unpredictable data. Its key advantage lies in continuously updating probability estimates as new fuzzing evidence emerges, flexibly incorporating CALLEE's static matching scores with dynamic observations. This adaptability allows maintaining accurate, up-to-date confidence measures for indirect edges with minimal computational overhead, even as runtime information evolves.

1) Formalization of the Bayesian-based Method: We begin by formalizing our Bayesian-based method:

- Let $G = (V, E)$ represent the CFG, where V is the set of basic blocks and E is the set of control-flow edges.
- Let $\mathcal{I}$ denote the set of recovered but not yet fuzz-covered indirect edges, and $e = (S, D)$ denote an indirect edge from source S to destination D in $\mathcal{I}$. We define $\phi : V \rightarrow 2^{\mathcal{I}}$ as the mapping assigning each block $v \in V$ to the set of uncovered indirect edges that affect its reachability to targets.
- Let $O = (n_s, n_d, obs = 0)$ represent the observation event, where n_s denotes the execution count of S, n_d denotes the execution count of D, and $obs = 0$ indicates that e has not been covered during fuzzing.
- Let event B represent the case where e truly exists, that is, e has been correctly recovered by static analysis. Conversely, event $\neg B$ indicates that e does not exist. Thus, $P(B)$ is

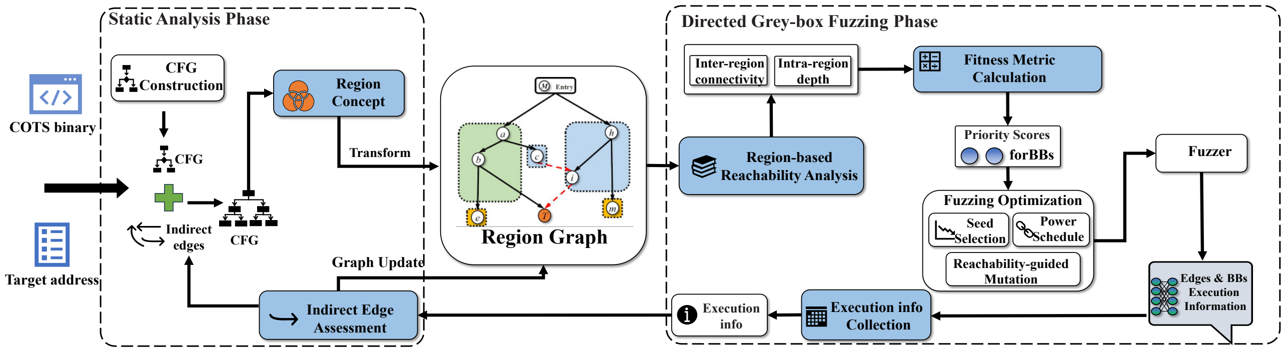


Figure 2: The overview of BinGo.

the prior probability that e is correct, $P(O \mid B)$ is the probability of observing $obs = 0$ given B is true, and $P(O \mid \neg B)$ is the probability of observing $obs = 0$ given B is false.

2) Bayesian Estimation of Accuracy Probability: Given the observation O, the posterior probability that e has been correctly recovered (i.e., the accuracy probability) can be updated using Bayes' theorem:

$$P(B \mid O) = \frac{P(O \mid B) \cdot P(B)}{P(O \mid B) \cdot P(B) + P(O \mid \neg B) \cdot (1 - P(B))} \tag{2}$$

Where $P(O \mid B)$ and $P(O \mid \neg B)$ are obtained via Maximum Likelihood Estimation (MLE):

$$\begin{aligned} P(O \mid B) &= P(obs = 0 \mid n_s, n_d, B) \cdot P(n_s, n_d \mid B) \\ P(O \mid \neg B) &= P(obs = 0 \mid n_s, n_d, \neg B) \cdot P(n_s, n_d \mid \neg B) \end{aligned} \tag{3}$$

Since n_s and n_d in O are observed facts rather than parameters to be inferred, we have $P(n_s, n_d \mid B) = 1$ and $P(n_s, n_d \mid \neg B) = 1$ as their deterministic occurrence. This simplifies Formula 3 to:

$$\begin{aligned} P(O \mid B) &= P(obs = 0 \mid n_s, n_d, B) \\ P(O \mid \neg B) &= P(obs = 0 \mid n_s, n_d, \neg B) \end{aligned} \tag{4}$$

Then, we approximate $P(obs = 0 \mid n_s, n_d, B)$ using MLE. Specifically, if e truly exists, and S and D are executed n_s and n_d times respectively without executing e, the probability of e being executed through random mutations during fuzzing can be approximated as:

$$p_{hit} = \frac{2}{n_s + n_d} \tag{5}$$

The denominator $n_s + n_d$ captures the case where both S and D are executed while e remains uncovered, whereas the numerator 2 applies Laplace smoothing to prevent zero-probability events and to account for the average contribution of S and D. This formulation estimates the edge-hit probability of e via MLE using only the execution counts of S and D, following the probabilistic approximation in prior work (e.g., DigFuzz). Such a simplification avoids the substantial computational cost of explicitly modeling complex factors that influence the execution of e, including path constraints and data dependencies.

Assumption: We treat each execution of S as an independent Bernoulli trial for traversing e, with failure probability $(1 - p_{hit})$ per trial. n_s determines the number of trials, while n_d contributes only to p_{hit} because D may be reached through alternative paths not involving e.

Under this model, the estimation of $P(O \mid B)$ is:

$$P(O \mid B) = P(obs = 0 \mid n_s, n_d, B) = \left(1 - \frac{2}{n_s + n_d}\right)^{n_s} \tag{6}$$

For $P(obs = 0 \mid n_s, n_d, \neg B)$, since B is false (i.e., e is incorrectly recovered and does not exist), e can never be covered regardless of n_s or n_d. Thus, we have $P(O \mid \neg B) = P(obs = 0 \mid n_s, n_d, \neg B) = 1$.

Combining these likelihoods with Bayes' theorem yields:

$$P(B \mid O) = \frac{\left(1 - \frac{2}{n_s+n_d}\right)^{n_s} \cdot P(B)}{\left(1 - \frac{2}{n_s+n_d}\right)^{n_s} \cdot P(B) + 1 \cdot (1 - P(B))} \tag{7}$$

Based on Formula 7, BinGo updates the accuracy probability of each uncovered indirect edge dynamically after each fuzzing observation. By efficiently combining static priors with runtime data, BinGo quantifies the uncertainty introduced by uncovered indirect edges, enabling precise and adaptive reachability analysis and fuzzing guidance.

C. Region-based Reachability Analysis

To efficiently analyze and update the reachability of all basic blocks, we introduce the concept of a **region**, which defines a new granularity level at which the CFG is transformed into a **region graph**, thereby facilitating efficient reachability analysis. Within the region graph, we develop a **region-based reachability analysis method**. This method employs **intra-region depth** to measure the difficulty of reaching a specific basic block from the region's entry point, and uses **inter-region connectivity** to measure the difficulty of transitioning from one region to another that is reachable by a target.

1) Region Concept: We begin by formalizing the concept of a *region*.

Definition 1 (Region) A region $R = (V_R, E_R)$ is a subgraph of G where $V_R = \{v \in V \mid \phi(v) = E^*\}$, for some fixed indirect-edge set $E^* \subseteq \mathcal{I}$, and $E_R = \{(u, v) \in E \mid u \in V_R, v \in V_R, (u, v) \notin \mathcal{I}\}$, i.e., E_R contains all direct or resolved edges between nodes in V_R.

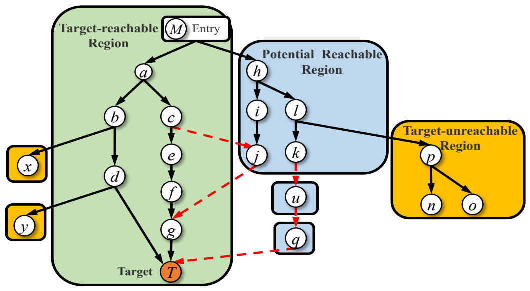


Figure 3: Reachability analysis on the region graph.

Based on this definition, we categorize regions into the following types, as illustrated in Figure 3:

- **Target-reachable region**: A region in which all basic blocks have at least one path to the target site, and none of these paths depend on any uncovered indirect edge. For instance, the green region in Figure 3 represents a target-reachable region where blocks (e.g., a and b) have paths to the target T, unaffected by indirect edges.
- **Target-unreachable region**: A region in which no basic block has any path to the target site, regardless of indirect edges. For instance, the yellow regions in Figure 3 represent target-unreachable regions, as blocks x and y have no paths to the target.
- **Potential reachable region:** A region in which all basic blocks have paths to the target site, but the reachability of these paths depends on one or more uncovered indirect edges. For instance, the blue regions in Figure 3 represent potential reachable regions. The reachability of blocks i and j both depends on the edge $<j, g>$, and removing $<j, g>$ would make both i and j unreachable.

2) Region Construction and Update: Algorithm 1 outlines the construction and online update process of the region graph.

Starting from the CFG $G = (V, E)$ and the set of target basic blocks V_T, we first remove the uncovered indirect edges $\mathcal{I}$ from E to obtain $E' = E \setminus \mathcal{I}$ and construct the reduced CFG $G' = (V, E')$ for initial region partitioning (Line 1). We then initialize empty region sets (Line 2). For each $v \in V$, if a path to any target basic block exists in G', v is classified to the target-reachable region R_t; otherwise, v is classified to the candidate region R_c (Lines 3–7). Then, we restore the uncovered indirect edges in $\mathcal{I}$ to recover the full CFG G (Line 8) and classify basic blocks in R_c. If v reaches a target in G, it is classified to the potential reachable region R_p; otherwise, it is classified to the unreachable region R_u (Lines 10-14). Finally, inter-region edges are added according to uncovered indirect-edge connections, resulting in a region graph RG that captures both deterministic intra-region control flow and probabilistic inter-region connections due to uncovered indirect edges.

During fuzzing, the region graph is updated online. When an uncovered indirect edge $e = (S, D) \in \mathcal{I}$ is executed, it is marked as covered and restored to E'. We then re-evaluate the reachability of S and update its region membership accordingly: (1) If S becomes target-reachable in G'

Algorithm 1: Region Construction and Updating

```
   Input: $G = (V, E)$, $V_T$ (Set of Target Nodes)
   Output: Region graph $RG$
 1 $E' \leftarrow E \setminus \mathcal{I}$ /* Remove uncovered indirect edges */
 2 Initialize: $R_t \leftarrow \emptyset$, $R_c \leftarrow \emptyset$
 3 foreach $v \in V$ do
 4     if HasPath($G' = (V, E'), v, V_T$) then
 5         $R_t \leftarrow R_t \cup \{v\}$
 6     else
 7         $R_c \leftarrow R_c \cup \{v\}$
 8 $E \leftarrow E' \cup \mathcal{I}$ /* Restore indirect edges */
 9 Initialize: $R_p \leftarrow \emptyset$, $R_u \leftarrow \emptyset$
10 foreach $v \in R_c$ do
11     if HasPath($G = (V, E), v, V_T$) then
12         $R_p \leftarrow R_p \cup \{v\}$
13     else
14         $R_u \leftarrow R_u \cup \{v\}$
   /* Online updating during fuzzing */
15 foreach $e = (S, D) \in \mathcal{I}$ when $e$ is executed do
16     $E' \leftarrow E' \cup \{e\}, \mathcal{I} \leftarrow \mathcal{I} \setminus \{e\}$ /* Mark $e$ as covered */
17     $\mathcal{R}(S, G') \triangleq \{ v \in V \mid \exists \text{ path}(S, v) \text{ in } G' \}$
18     $\mathcal{R}(S, G) \triangleq \{ v \in V \mid \exists \text{ path}(S, v) \text{ in } G \}$
19     if HasPath($G' = (V, E'), S, V_T$) and $R(S) \neq R_t$ then
20         $R_t \leftarrow R_t \cup \mathcal{R}(S, G') \cup S$
21     else if HasPath($G = (V, E), S, V_T$) and $R(S) \neq R_p$ then
22         $R_p \leftarrow R_p \cup \mathcal{R}(S, G) \cup S$
23     else if not HasPath($G = (V, E), S, V_T$) then
24         $R_u \leftarrow R_u \cup S$
25 return $RG$
```

and is not yet in R_t, we migrate S with all basic blocks in its reachable subgraph in G', i.e., $\{S\} \cup \mathcal{R}(S, G')$, to R_t (Lines 20-21); (2) If S becomes target-reachable in G but not in R_p, we migrate S with all basic blocks in its reachable subgraph in G, i.e., $\{S\} \cup \mathcal{R}(S, G)$, to R_p (Lines 22-23); (3) if S is target-unreachable in G, we move it to R_u (Lines 24–25). After each change, regions are merged or split as necessary, and inter-region edges are updated to ensure the region graph always reflects the latest reachability information and indirect-edge status.

3) Intra-region depth: As illustrated in Figure 3, a region R is a subgraph of the CFG that has a single entry block $root(R)$ and one or more exit nodes. For one basic block $m \in V_R$, let P_{root} denote all paths from $root(R)$ to m within R, the intra-region depth of m is the length of the longest path from the entry block $root(R)$ to m, denoted as: $d_m = \max_{p \in P_{\text{root} \to m}} |p|$, where $|p|$ denotes the number of basic blocks in path p.

For the *target-reachable region*, the exit node is the target basic block which has the maximum intra-region depth. For any basic block m within this region, a greater intra-region depth means it is farther from the entry basic block but closer to the target basic block, indicating higher reachability to the target basic block:

$$R_{depth}(m) = \frac{1}{d_{target} - d_m} \tag{8}$$

Formula 8 quantifies the reachability of m based on intra-region depth, where $R_{depth}(m)$ is the reachability, and d_{target} and d_m are the depths of the target and m, respectively.

For the *potential reachable region*, exit nodes (e.g., nodes j and k) are the source basic blocks of uncovered indirect edges connecting this region to either a target-reachable region or another potential reachable region (e.g., nodes g and u). For a basic block m, greater intra-region depth suggests proximity to exit nodes, increasing the likelihood of reaching the target-reachable region via indirect edges:

$$R_{depth}(m) = \frac{\sum\limits_{exit \in \eta} \frac{1}{d_{exit} - d_m}}{|\eta|} \tag{9}$$

Formula 9 calculates the reachability of m based on the depth difference with exit basic blocks in η, where η is the set of exit basic blocks that can be reached by m.

For the *target-unreachable region*, reachability is not analyzed as all basic blocks are unreachable. Notably, in both Formula 8 and Formula 9, R_{depth} is bounded by 1. When m is neither a target nor an exit basic block, either d_{target} or $d_{\text{exit}(e)}$ exceeds d_m, resulting in $R_{\text{depth}}(m) \leq 1$. When m is a target or an exit basic block, we directly set $R_{\text{depth}}(m) = 1$.

4) Inter-region connectivity: If a basic block (e.g., h in Figure 3) can reach the target basic block via a path that contains uncovered indirect edges, it must first reach an exit node within its own region and then traverse one or more inter-region transitions (e.g., $<j, g>$ or $<k, u, q, T>$) to reach the target. The difficulty of these inter-region transitions is related to the accuracy probability of the indirect edges, as a lower probability suggests a higher likelihood of incorrect recovery, potentially disrupting paths between regions and hindering transitions. To quantify the challenge of reaching the target through inter-region transitions, we introduce the concept of *inter-region connectivity*.

In graph theory, the connectivity between two regions depends on the length and number of inter-region paths. An inter-region path is defined as the sequence of regions traversed to move from one region to another (e.g., path $<k, u, q, T>$ includes four regions, giving it a length of 4). The connectivity should adhere to two principles: (1) it is inversely related to the path length, as longer paths present more obstacles; and (2) it is directly related to the number of paths between two regions, as more paths increase the likelihood of successful traversal through paths.

For any two adjacent regions r_i and r_j connected by n uncovered indirect edges, the connectivity is calculated as:

$$c_{i,j} = 1 - \prod_{k=1}^{n} (1 - P_k(i, j)) \tag{10}$$

Where $P_k(i, j)$ denotes the accuracy probability of the k^{th} indirect edge connecting r_i and r_j. The term $\prod\limits_{k=1}^{n} (1 - P_k(i, j))$ reflects the probability that all indirect edges are incorrectly recovered. Formula 10 indicates that even if only one edge is correctly recovered, r_i and r_j are still considered connected.

Formula 11 uses connectivity to quantify reachability between region r_m, containing basic block m, and the target-reachable region r_t, containing the target block, via inter-region paths:

$$R_{cnct}(r_m, r_t) = \sum_{p \in Q(r_m, r_t)} \prod_{(i,j) \in p} c_{i,j} \tag{11}$$

Where $Q(r_m, r_t)$ denotes the set of all possible inter-region paths from r_m to r_t, and p denotes a specific path within this set. Each path requires at least one indirect edge between adjacent regions to ensure connectivity. The connectivity between r_m and r_t along a path is the product of connectivities between adjacent regions. Reachability also depends on the number of inter-region paths; thus, we sum the connectivities of all paths from r_m to r_t. As accuracy probabilities are updated, inter-region connectivity is continuously adjusted.

D. Fitness Metric and Optimization

Based on the reachability analysis, we propose a new fitness metric called *probabilistic reachability* to measure and score the reachability of basic blocks.

1) Calculation of probabilistic reachability: The probabilistic reachability score of a basic block m is determined by two key factors: intra-region depth and inter-region connectivity. The calculation is categorized as follows:

(1) **Basic block m in the target-unreachable region.** Since no basic block in this region can reach the target, its probabilistic reachability score is set to 0.

(2) **Basic block m in the potential reachable region.** To reach the target, m must first traverse one of the exit basic blocks of its own region and then follow an inter-region path to a target-reachable region. In this case, inter-region connectivity is the more dominant factor, because it requires satisfying uncovered indirect edge conditions and successfully navigating multiple inter-region transitions. Accordingly, we place greater emphasis on inter-region connectivity:

$$score(m) = R_{depth}(m) + \frac{|E|}{|\mathcal{I}|} \cdot R_{cnct} \tag{12}$$

Where $score(m)$ denotes the reachability score of m, and $\frac{|E|}{|\mathcal{I}|}$ is an adaptive weight that balances term contributions across programs, avoiding manual parameter tuning and improving robustness. First, we do not normalize R_{depth} and R_{cnct} since they are already scale-comparable ($R_{depth} \leq 1$, typically $R_{cnct} \leq 3$), thus avoiding unnecessary computation. Second, the adaptive weight $\frac{|E|}{|\mathcal{I}|}$ reflects the granularity difference between intra-region and inter-region transitions. A lower proportion of inter-region edges indicates increasing difficulty of executing indirect edges and performing inter-region transitions, thus placing greater weight on inter-region transitions.

(3) **Basic block m in the target-reachable region.** When m belongs to a target-reachable region, m has maximum inter-region connectivity $\max \mathrm{R}_{\mathrm{cnct}}$, and its probabilistic reachability score is given by:

$$score(m) = R_{depth}(m) + \frac{|E|}{|\mathcal{I}|} \cdot \max \mathrm{R}_{\mathrm{cnct}} \tag{13}$$

Based on the reachability scores of basic blocks, the **reachability score for each seed** is calculated as:

$$score(s) = \frac{\sum_{m \in \xi(s)} score(m)}{|\xi(s)|} \tag{14}$$

Where $\xi(s)$ denotes the set of all basic blocks along the path of s, and $|\xi(s)|$ denotes the number of basic blocks in $\xi(s)$.

2) Reachability-guided fuzzing optimization: With the reachability scores of basic blocks and seeds, we optimize the input prioritization, power scheduling, and mutation strategy to prioritize covering basic blocks with high reachability (i.e., high-reachability basic blocks).

Input prioritization. BinGo first prioritizes inputs that uncover new edges between high-reachability basic blocks. If new edges are not found, BinGo re-mutates previously fuzzed seeds, focusing on inputs whose execution paths include high-reachability blocks within these regions.

Power scheduling. Based on the reachability scores of seeds and AFL's method for basic energy assignment, BinGo adjusts the energy assigned to each seed:

$$\hat{p}(s) = p_{afl} \cdot \frac{score(s)}{\widetilde{score}} \tag{15}$$

Seeds with higher scores receive more energy, increasing the chance of overcoming path constraints and reaching targets. We allocate more energy to seeds with above-average reachability scores, enhancing the focus on promising paths.

Mutation strategy. We define ***promising basic blocks*** as those that, if target-reachable (e.g., block m), have successor blocks (e.g., l) that are also target-reachable but not yet covered. During seed mutation, we log covered blocks and identify promising ones. We then check if mutations affect variable values in these blocks, marking the positions of bytes causing such changes as ***rough promising bytes***. These bytes are refined into ***fine-grained promising bytes*** by intersection. Finally, we dynamically adjust the selection probabilities of fine-grained promising bytes based on mutation effectiveness compared to other bytes:

$$P(bytes_{fp}) = N_{pbb(s)}/N_s \tag{16}$$

Where $bytes_{fp}$ denotes fine-grained promising bytes, $N_{pbb}(s)$ denotes the number of times variable changes occur in promising basic blocks within seed s, and N_s denotes the total number of mutations on seed s.

IV. Implementation

The implementation of BinGo is based on QEMU-AFL [5]. Specifically, BinGo integrates IDA Pro and CALLEE to recover indirect edges and construct CFGs from COTS binaries. Our prototype of BinGo comprises approximately 1800 lines of C/C++ code and 2300 lines of Python code. The C/C++ code is dedicated to the fuzzing component, which collects runtime information, such as edge and basic block executions. It also calculates reachability scores for seeds to optimize input prioritization and power scheduling. Python code is used for the static analysis component. It performs structural information enhancement and region classification and supports the fuzzing process by conducting reachability analysis.

V. Evaluation

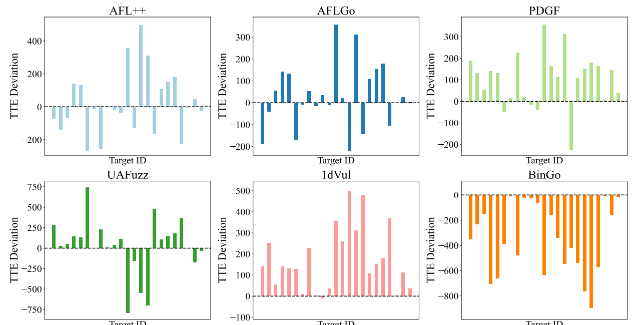
To evaluate BinGo, we conducted extensive experiments aiming to answer the following research questions:

- **RQ1:** How does BinGo perform in reaching targets?
- **RQ2:** How effective is BinGo in exposing known vulnerabilities?
- **RQ3:** How does each component of BinGo contribute to its overall performance?
- **RQ4:** What is the overhead introduced by each component of BinGo?
- **RQ5:** How does BinGo perform in terms of discovering new vulnerabilities?

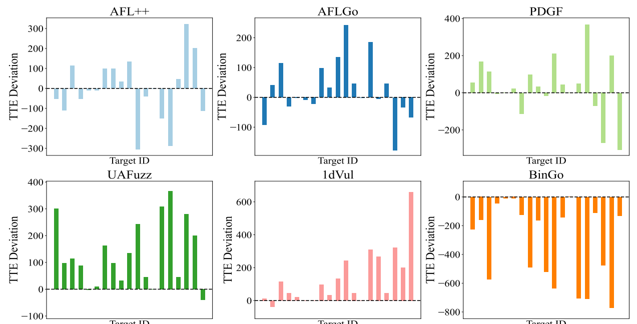
A. Evaluation Setup

1) Comparison targets: We use six state-of-the-art fuzzers as baselines, including AFL++ (QEMU mode, shortened to AFL++ in the evaluation) [16], AFLGo [11], PDGF [43], UAFUZZ [31], and 1dVul [33]. Among them, AFL++ is a binary-level coverage-guided fuzzer, aiming to determine whether similar performance gains of BinGo can be achieved with an efficient binary-level coverage-guided fuzzer. The other four baseline fuzzers are directed fuzzers. First, we selected AFLGo, a well-known DGF tool, and created a binary-only version (AFLGo$_{cots}$) based on their paper. We also used the source-code-level version of AFLGo (AFLGo$_{open}$) as the ground truth to analyze the accuracy of reachability analysis in BinGo and AFLGo$_{cots}$. Second, among DGF approaches that dynamically supplement CFGs and update reachability (e.g., ParmeSan [32], PDGF [43], Seive [35], and FishFuzz [44]), we chose PDGF because it does not require sanitizers for pre-analysis and allows pre-defined targets. Since PDGF is designed at the source-code level, we implemented a binary-only version following their paper's description. Finally, since 1dVul is not open-sourced, we also reproduced it according to its design described in the paper.

2) Evaluation benchmarks: We selected **one dataset and two benchmarks** for our evaluation, covering a diverse range of programs, with a total of **194 targets**. (1) **DARPA Cyber Grand Challenge (CGC) dataset:** The CGC dataset binaries contain various obstacles to validate binary analysis and dynamic testing capabilities. To ensure fairness, we selected the same 126 single-object applications from CGC as those used by 1dVul [33]. (2) **CVE-Benchmark:** A set of real-world programs containing diverse types of known vulnerabilities, which have been evaluated in previous works, e.g., Beacon [18], DAFL [22], SelectFuzz [28], and 1dFuzz [41]. (3) **UniBench benchmark:** This benchmark includes various real-world programs and has been used in prior studies [15], [27], [43]. To ensure consistency, we used the same 16 programs with 30 targets as used by PDGF.



(a) TTE deviation to the average on CVE-Benchmark



(b) TTE deviation to the average on UniBench.

Figure 4: TTE deviation to the average on CVE-Benchmark and UniBench.

3) Experiment settings: Experiments were conducted in a Docker environment on a 64-bit Ubuntu machine equipped with 1 CPU core (Intel Xeon(R) Gold 6133 CPU @2.50GHz) and 8 GB of memory. All fuzzers were launched using one CPU core in the Docker environment. For each fuzzing campaign, we used the initial seeds provided in the dataset if available; otherwise, we used a file containing the string "hello" as the initial seed. Each baseline fuzzer was configured as recommended in its respective paper. For statistical analysis, we employed the Mann-Whitney U test to compute p-values and determine significance. Experiments on the CGC dataset were repeated three times with an 8-hour time budget per run, consistent with the 1dVul setup. For the CVE-Benchmark and UniBench, experiments were repeated five times, each with a 24-hour time budget. In total, we used approximately **42,000 CPU hours** to evaluate BinGo's effectiveness.

B. Reaching Target Locations (RQ1)

Target selection. We compared BinGo with the baseline fuzzers on reaching the target locations in the CGC dataset. The CGC dataset includes 126 applications along with their corresponding patched versions, which contain fixes for the identified vulnerabilities. We utilized BinDiff [3] to perform binary diffing between these 126 applications and their patched versions to identify the differing basic blocks. From these differing basic blocks, we randomly selected one basic block as the fuzzing target for each application. Consequently, our evaluation included 126 targets, and we used the **Time-to-Reach (TTR)** metric to assess the time required to generate the first input whose path covers the target basic block.

TTR results. BinGo can reach the most (77/126) target sites compared to AFL++ (69/126), AFLGo$_{cots}$ (64/126), PDGF (59/126), UAFuzz (60/126), and 1dVul (70/126) within the time budget. For the mean TTR, BinGo demonstrated 1.23×, 1.37×, 1.78×, 1.53×, and 1.26× speedup compared to AFL++, AFLGo$_{cots}$, PDGF, UAFuzz, and 1dVul, respectively. P-values were below 0.05, indicating that the improvements achieved by BinGo in TTR are statistically significant.

C. Exposing Known Vulnerabilities (RQ2)

To answer **RQ2**, we compared BinGo with the baseline fuzzers in exposing known vulnerabilities in the CVE-Benchmark and UniBench [25]. We used the **Time-to-Exposure (TTE)** metric to assess the time required to expose the vulnerability at the target site. A crash observed at the target site indicates that the fuzzer has successfully exposed the vulnerability. We present the TTE results using two methods. First, the raw TTE results for CVE-Benchmark and UniBench are shown in Table I, and Table II. Additionally, to provide a more straightforward comparison, bar charts in Figure 4 visualize these results. In these figures, the x-axis represents the sequential target numbers, and the y-axis shows the TTE deviation from the average. For each target, we first calculate the average TTE across all fuzzers, and subtract this average from each fuzzer's TTE. A negative TTE deviation indicates superior vulnerability-exposure performance compared to other fuzzers. Furthermore, targets that timed out for all fuzzers were excluded from the analysis.

Comparison with fuzzers in CVE-Benchmark. The de-

TABLE I: The TTE results for the CVE-Benchmark.

No	Program	CVE	AFL++	AFLGo$_{cots}$	PDGF	UAFuzz	1dVul	BinGo$_I$	BinGo$_R$	BinGo
1	cxxfilt	2016-4491	582.3m	464.4m	842.4m	935.1m	794.3m	398.1m	422.6m	**301.4m**
2		2016-6131	255.7m	352.8m	523.1m	423.1m	644.3m	273.2m	340.9m	**159.4m**
3	objdump	2017-8392	1321.3m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1233.5m**
4		2017-8396	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
5		2017-8398	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
6		2017-16828	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
7		2018-17360	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
8	objcopy	2017-7303	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
9		2017-8393	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
10		2017-8394	T.O.	T.O.	T.O.	T.O.	T.O.	983.1m	T.O.	**592.8m**
11		2017-8395	T.O.	T.O.	T.O.	T.O.	T.O.	1044.7m	T.O.	**648.2m**
12	cjpeg	2018-14498	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
13		2020-13790	427.4m	526.4m	647.2m	T.O.	823.4m	454.7m	499.2m	**308.2m**
14	swftophp	2016-9827	**16.9m**	21.3m	42.4m	34.2m	39.6m	17.2m	23.9m	18.4m
15		2016-9829	954.8m	1264.8m	T.O.	T.O.	T.O.	916.2m	1042.6m	**736.1m**
16		2016-9831	79.2m	63.7m	98.2m	86.4m	81.6m	61.4m	74.9m	**57.2m**
17		2017-7578	43.5m	98.4m	48.2m	104.3m	53.2m	44.2m	85.1m	**39.2m**
18		2017-9988	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
19		2017-11728	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
20		2017-11729	102.5m	128.4m	98.4m	253.6m	173.6m	110.8m	92.4m	**75.3m**
22		2018-8807	404.2m	554.9m	696.3m	377.6m	792.4m	459.4m	398.1m	**374.7m**
23		2018-8962	T.O.	725.8m	1058.6m	**395.3m**	T.O.	688.3m	703.4m	603.5m
24		2018-11095	T.O.	T.O.	T.O.	**427.5m**	T.O.	T.O.	T.O.	583.2m
25		2018-11225	799.1m	818.4m	734.6m	T.O.	T.O.	794.3m	649.3m	**542.1m**
26		2018-11226	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
27		2018-20427	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**792.3m**
28		2019-9114	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**523.1m**
29		2019-12982	T.O.	T.O.	T.O.	T.O.	T.O.	941.6m	T.O.	**364.5m**
30		2020-6628	843.2m	968.1m	1235.8m	T.O.	T.O.	T.O.	T.O.	**502.4m**
31	xmllint	2017-5969	24.6m	28.4m	34.6m	27.1m	31.9m	27.5m	29.1m	**22.5m**
32		2017-9047	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
33		2017-9048	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
34		2017-9049	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
35	lrzip	2017-8846	309.1m	288.4m	406.8m	**89.3m**	375.9m	207.4m	241.9m	107.2m
36		2018-11496	116.4m	142.6m	178.4m	**104.6m**	175.4m	138.3m	157.2m	122.4m
37	pngimage	2018-13785	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
38	avaconv	2018-18829	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
	speedup		**1.85×**	**1.95×**	**2.21×**	**2.07×**	**2.39×**	**1.68×**	**1.70×**	-
	mean p-values		**0.012**	**0.010**	**0.005**	**0.007**	**0.003**	**0.015**	**0.018**	-

T.O. means that the fuzzers cannot expose known vulnerabilities within 24 hours; the text in bold represents the best TTE performance.

TABLE II: The TTE results on programs from UniBench.

No.	Program	Target sites	AFL++	AFLGo$_{cots}$	PDGF	UAFuzz	1dVul	BinGo$_I$	BinGo$_R$	BinGo
1	cflow	parser.c:1284	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
2	mp42aac	Factory.cpp:89	289.4m	248.6m	395.3m	642.3m	353.4m	196.4m	159.4m	**117.6m**
3	mp42aac	Factory.cpp:105	244.2m	394.2m	523.4m	451.2m	317.2m	253.2m	322.5m	**193.4m**
4	mp42aac	Buffer.cpp:175	T.O.	T.O.	T.O.	T.O.	T.O.	1128.4m	T.O.	**753.7m**
5	jhead	jpgqguess.c:108	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
6	mp3gain	gain analysis.c:195	**76.2m**	98.4m	121.4m	217.2m	174.6m	77.2m	97.5m	82.4m
7	mp3gain	interface.c:188	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
8	lame	util.c:608	8.3m	14.6m	18.3m	15.2m	36.9m	**7.3m**	13.5m	9.2m
9	lame	util.c:606	10.3m	11.2m	42.1m	29.6m	16.8m	10.9m	11.8m	**8.6m**
10	lame	get_audio.c:865	287.4m	166.7m	76.3m	352.7m	186.4m	109.5m	125.3m	**65.4m**
11	imginfo	jpc_dec.c:1297	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
12	imginfo	jpc_dec.c:516	T.O.	T.O.	T.O.	T.O.	T.O.	1337.8m	T.O.	**851.9m**
13	imginfo	jpc_t1cod.c:144	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
14	pixdata	io-tga.c:441	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
15	pixdata	pixops.c:1214	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1244.6m**
16	pixdata	io-ico.c:59	T.O.	T.O.	1288.5m	T.O.	T.O.	T.O.	T.O.	**785.4m**
17	pixdata	io-pcx.c:528	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
18	tcpdump	in_cksum.c:108	892.6m	T.O.	1410m	T.O.	T.O.	993.5m	T.O.	**562.9m**
19	tcpdump	print-isakmp.c:2502	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
20	tic	name_match.c:102	1353.9m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**1252.3m**
21	flvmeta	dump_xml.c:151	**2.3m**	3.4m	6.4m	4.2m	5.4m	8.4m	4.7m	5.8m
22	pdftotext	FoFiType1.cc:206	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
23	pdftotext	OutputDev.cc:3065	982.7m	1317.2m	1182.4m	T.O.	T.O.	847.5m	1027.4m	**428.3m**
24	exiv2	basicio.cpp:1003	785.9m	1068m	T.O.	T.O.	1341.2m	931.7m	519.4m	**364.3m**
25	objdump	libbfd.c:618	T.O.	T.O.	1322.4m	T.O.	T.O.	T.O.	T.O.	**1284.6m**
26	objdump	dwarf2.c:2553	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
27	mujs	jsdtoa.c:725	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
28	wav2swf	wav.c:281	T.O.	941.7m	847.3m	1399.4m	T.O.	1065.6m	935.7m	**644.7m**
29	tiffcp	tiffcp.c:1423	T.O.	1205.1m	T.O.	T.O.	T.O.	836.4m	1031.7m	**469.3m**
30	tiffcp	tiffcp.c:1596	342.9m	388.3m	**149.3m**	415.2m	1116m	351.8m	463.9m	324.3m
		speedup	**1.72×**	**1.79×**	**2.07×**	**2.47×**	**1.82×**	**1.30×**	**2.30×**	-
		mean p-values	**0.029**	**0.018**	**0.007**	**0.006**	**0.012**	**0.033**	**0.008**	-

tailed TTE results on real-world programs from the CVE-Benchmark are presented in Table I. Among the fuzzers, BinGo can expose the most vulnerabilities (23) compared to AFL++(15), AFLGo$_{cots}$ (15), PDGF (14), UAFuzz (13), and 1dVul (11). For the mean TTE, BinGo demonstrated 1.85×, 1.95×, 2.21×, 2.07×, and 2.39× speedup compared to AFL++, AFLGo$_{cots}$, PDGF, UAFuzz, and 1dVul, respectively. Moreover, as Figure 4(a) shows, for almost all targets, BinGo's TTE deviations are negative. Most of BinGo's TTEs are significantly lower than the average, indicating that BinGo's TTE performance consistently surpasses the average performance of all other fuzzers. Based on the above analysis, we can conclude that **BinGo can expose known vulnerabilities faster than baseline fuzzers in the CVE-Benchmark**.

Comparison with fuzzers in UniBench The detailed TTE results on real-world programs from UniBench are presented in Table II. Among the fuzzers, BinGo exposed the most vulnerabilities (19) compared to AFL++ (12), AFLGo$_{cots}$ (12), PDGF (13), UAFuzz (9), and 1dVul (9). For the mean TTE, BinGo demonstrated 1.72×, 1.79×, 2.07×, 2.47×, and 2.30× speedup compared to AFL++, AFLGo$_{cots}$, PDGF, UAFuzz, and 1dVul, respectively. Moreover, as Figure 4(b) shows, for almost all targets, BinGo's TTE deviations are negative. Most of BinGo's TTEs are significantly lower than the average, indicating that BinGo's TTE performance in the UniBench consistently surpasses the average performance of all other fuzzers. Based on the above analysis, we can conclude that **BinGo can expose known vulnerabilities faster than the baseline fuzzers in the UniBench benchmark**.

D. Component-wise Analysis (RQ3)

To address **RQ3**, we conducted an ablation study to evaluate the impact of Indirect Edge Assessment (i.e., *IE*) and Region-based Guidance (i.e., *RG*) optimizations on the overall performance of BinGo. The *RG* optimizations include the region-based reachability analysis and optimizations in §III-C

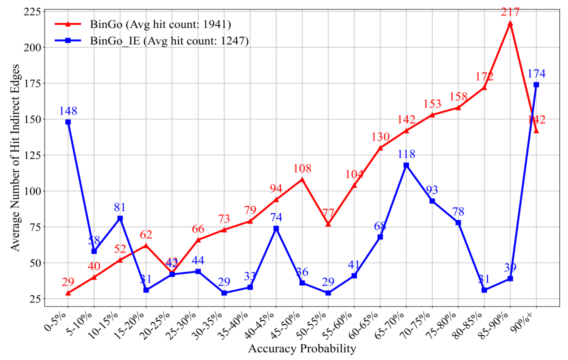
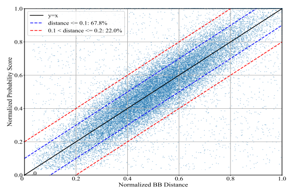


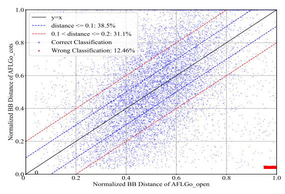
Figure 5: Distribution of accuracy probability and the number of hit indirect edges.

and §III-D. First, we disabled the *IE* module from BinGo to form BinGo$_I$, where indirect edges are recovered solely by CALLEE with fixed accuracy probabilities from CALLEE's matching similarity [45]. Then, we replaced our region-based guidance with AFLGo's distance-based strategy from BinGo to form BinGo$_R$, updating distances periodically based on indirect edge accuracy probabilities. We evaluated these variants on both the CVE-Benchmark and UniBench, with detailed TTE results in Table I and Table II. In the CVE-Benchmark, BinGo exposed the most vulnerabilities (21), compared to BinGo$_I$ (17) and BinGo$_R$ (17), with mean TTE speedups of 1.68× and 1.70×, respectively. In the UniBench benchmark, BinGo exposed 19 vulnerabilities, versus 14 for BinGo$_I$ and 12 for BinGo$_R$, with mean TTE speedups of 1.30× and 2.30×. From these TTE analysis results, **we can conclude that both *IE* and *RG* contribute significantly to reducing TTE in BinGo.** Additionally, to further validate the impact of each component, we conducted the following experiments.

Accuracy of the *IE* module. We claim that the dynamically updated accuracy probabilities from the *IE* module reliably reflect the likelihood of correctly recovering indirect edges. To validate this, we analyzed the correlation between edge hit counts and their accuracy probabilities. Theoretically, if the accuracy probability is reliable, edges with higher probabilities should be hit more frequently, indicating correct recovery and accurate reachability analysis. We categorized indirect edges into 19 probability intervals (e.g., 0-5%, 5-10%, ..., 85%-90%, 90%+). For each category, we recorded two statistics: (1) the average number of indirect edges hit per test, and (2) the observed rates of correct versus incorrect edge recovery. The accuracy probability of each edge is dynamically updated during fuzzing: it decreases only when the indirect edge remains unhit while its source or destination blocks are frequently executed (see Formula 7). This mechanism ensures that edges with low accuracy probabilities have typically undergone extensive exploration, yet remain unhit, indicating a higher likelihood of incorrect recovery. Conversely, edges with high accuracy probabilities are easily exercised, suggesting a greater likelihood of correct recovery. Thus, a positive correlation between hit count and accuracy probability demonstrates the reliability of our approach, with higher probabilities empirically linked



(a) Deviation between reachability and distance scores.



(b) Deviation between distances in two AFLGo versions.

Figure 6: Deviation between different fitness metrics in BinGo and AFLGo$_{cots}$.

to a greater likelihood of correct recovery.

Figure 5 presents the distribution of accuracy probabilities and the average number of hit indirect edges per category. The x-axis shows the accuracy probability intervals, and the y-axis indicates the average hit count. The red and blue lines represent BinGo and BinGo$_I$, respectively. For BinGo, hit counts generally rise with higher accuracy probabilities, except in the 90%+ range, where the decline is due to the initially small number of edges with very high static probabilities, which decrease as fuzzing progresses. In contrast, BinGo$_I$ shows no clear correlation between accuracy probability and hit counts. These results confirm that indirect edges with higher accuracy probabilities are more likely to be correctly recovered and hit, supporting more effective target reachability. Furthermore, BinGo also achieves significantly more hit target-reachable indirect edges (**1941**) compared to BinGo$_I$ (**1247**) and the binary-only version of PDGF (**652**) per program on average, indicating enhanced CFG coverage and more accurate reachability analysis for binary-level DGF.

Accuracy of *RG* module. The effectiveness of the *RG* module depends on whether the probabilistic reachability scores accurately reflect the reachability of regions and basic blocks. To evaluate the accuracy of the probabilistic reachability scores, we measured the deviation between the probabilistic reachability scores and BB distances calculated by AFLGo$_{open}$, which served as the ground truth. First, we constructed the CFG using AFLGo$_{open}$ and LLVM-CFI, and calculated BB distances for all basic blocks. We then normalized both the probabilistic reachability scores and BB distances for comparability, assigning 0 to the minimum value and 1 to the maximum value for each metric. The deviation for each basic block was calculated as the absolute difference between its normalized reachability score and BB distance.

Figure 6(a) illustrates the deviations for all basic blocks: each dot represents a basic block, with the x-axis showing the normalized BB distance and the y-axis showing the normalized reachability score. The black line is the $y = x$ diagonal, while the blue and red dashed lines indicate deviations of 0.1 and 0.2, respectively. As shown in Figure 6(a), most data points are tightly clustered near the $y = x$, with 67.8% of blocks within 0.1 and 89.8% within 0.2 of the diagonal. The average deviation for BinGo is **0.0936**, indicating that the normalized probabilistic reachability scores closely approximate the ground-truth BB distances. For comparison, Figure 6(b) shows the deviation between the BB distances in AFLGo$_{cots}$ and AFLGo$_{open}$. Only 38.5% of dots lie within 0.1 of the $y = x$ diagonal and 69.6% within 0.2. The average deviation of AFLGo$_{cots}$ is **0.1637**, which is significantly larger than that of BinGo (0.0936), confirming that probabilistic reachability scores more accurately reflect basic block reachability in binary-level DGF than distance metrics. Additionally, we analyzed fuzzing resource allocation and found that BinGo assigns substantially more resources to high-reachability basic blocks (normalized BB distance above 0.5) compared to AFLGo$_{cots}$. High-reachability blocks account for 43% of total hits in BinGo, versus only 27% in AFLGo$_{cots}$. Thus, we can conclude that **the probabilistic reachability scores generated by the *RG* module reliably represent basic block reachability, effectively guiding DGF to high-reachability basic blocks**.

TABLE III: Fuzzing throughput of fuzzers ($exec/s$).

	AFL++	AFLGo	PDGF	UAFuzz	1dVul	BinGo$_I$	BinGo$_R$	BinGo
FT	937	893	649	766	673	872	703	821
INCR.	-	95.3%	69.3%	81.8%	71.8%	93.1%	75.0%	87.6%

E. Overhead Analysis (RQ4)

To answer **RQ4**, we measure the overhead introduced by different components of our approach by evaluating the reduction in fuzzing throughput (i.e., the number of inputs tested per unit time). We calculated the mean fuzzing throughput for AFL++, AFLGo$_{cots}$, PDGF, UAFuzz, 1dVul, BinGo$_I$, BinGo$_R$, and BinGo. As shown in Table III, #INCR. represents the fuzzing throughput of each fuzzer relative to AFL++, and the key observations on fuzzing throughput across configurations are as follows: First, BinGo$_I$ achieves a slightly higher throughput than BinGo, indicating that the *IE* module introduces minimal overhead and maintains fuzzing performance. Second, BinGo$_R$ exhibits sign**??**ificantly lower throughput compared to BinGo, suggesting that distance-based methods for updating BB distances incur substantially higher overhead than the reachability updating approach in the *RG* module. Third, we observe that PDGF, a source-code-level DGF tool, is not well-suited for binary-level DGF, showing the lowest throughput. Specifically, the high overhead from QEMU emulation of two binaries leads to an 18.3% lower throughput compared to BinGo.

F. Discovering New Vulnerabilities (RQ5)

To address **RQ5**, we applied BinGo to discover new vulnerabilities in real-world programs. In addition to UniBench, we included applications evaluated in recent fuzzing studies [18], [28], [39], such as cscope, LibJPEG, and LibPNG. We first identified vulnerable functions with assigned CVE-IDs in older versions of the tested programs. Next, we used BinDiff [3] to perform differential analysis between the old and new versions to locate newly added or modified code regions. Basic blocks from these differing regions were randomly selected as targets for directed fuzzing with BinGo to uncover new vulnerabilities. As a result, we identified three previously undiscovered vulnerabilities in cscope-15.9 [2], listed in Table IV. **All newly discovered vulnerabilities have been**

TABLE IV: New vulnerabilities detected by BinGo.

No	Prog	Bug Loc	Bug Type	Bug-ID
1	cscope-15.9	cscope+0x2ae05	heap-buffer-overflow	APPLYING
2	cscope-15.9	cscope+0x2fb7e	stack-buffer-overflow	APPLYING
3	cscope-15.9	cscope+0x2af53	heap-buffer-overflow	APPLYING

responsibly reported to vendors and are currently awaiting vulnerability ID assignment. These findings demonstrate that **BinGo can effectively collaborate with binary differencing tools to discover new vulnerabilities in COTS binaries.**

VI. Discussion

Necessity of the Bayesian-based method for indirect edge assessment. Existing dynamic fuzzing techniques can update coverage only for executed indirect edges, leaving the influence of unexecuted ones unassessed. To address this gap, we propose a Bayesian-based method that quantifies recovery uncertainty for such edges. The Bayesian model is well-suited for fuzzing's sparse, incomplete, and unpredictable runtime data, seamlessly combining static priors (e.g., CALLEE scores) with dynamic observations. As new evidence emerges, it incrementally refines confidence estimates, enabling accurate and adaptive reachability guidance even with incomplete coverage.

Novelty of the region concept in BinGo. Our region concept arises from the observation that unexecuted indirect edges affect the reachability of downstream basic blocks, distinguishing it from prior uses in LLVM [7], SESE-region [19], and TargetFuzz [12]. We define regions based on the reachability of blocks influenced by indirect edges, enabling us to capture this impact explicitly. This new abstraction transforms the CFG into a region graph, facilitating efficient reachability analysis. Leveraging the dynamically updated region graph and reachability scores, our fitness metric remains adaptive, lightweight, and robust to imperfect binary-level CFGs.

Impact of CFG and region accuracy on fuzzing effectiveness. The accuracy of CFG recovery and region classification critically affects directed fuzzing. Inaccurate indirect edges can misclassify basic blocks and regions as reachable or unreachable, skewing energy allocation and degrading guidance (§II-B). Such errors cascade to techniques relying on reachability, e.g., distance-based scoring and path pruning, further reducing effectiveness. Similar challenges appear in probabilistic symbolic execution, where analyses must reason over uncertain control flow. For instance, Symbolic PathFinder [17] augments symbolic execution with path-probability computation via constraint counting, prioritizing paths more likely to reach targets. Applying analogous probabilistic reasoning in fuzzing can mitigate CFG inaccuracies by aligning resource allocation with true reachability likelihoods.

VII. Related Work

Directed grey-box fuzzing. Source-code level DGF tools [11], [13], [15], [23], [43] have gained significant research attention in recent years. For example, SelectFuzz [28], DAFL [22], and SDFuzz [24] exclude codes irrelevant to the target sites, thereby improving the efficiency and effectiveness of DGF. TargetFuzz [12] organizes regions based on the coverage of seed-executed paths and focuses on those covered by close seeds, which differs from our region-based methods. However, only a few works investigate DGF for COTS binaries, and those that do are typically tailored for specific scenarios. For instance, V-Fuzz [26] employs a machine-learning approach to optimize power schedule, aiming to trigger vulnerable targets. UAFUZZ [31] utilizes operation sequence coverage to optimize seed selection, aiming to accelerate the discovery of use-after-free vulnerabilities. 1dFuzz [41] designs fitness metrics to reproduce 1-day vulnerabilities by leveraging unique features of patches. Unlike the aforementioned works, BinGo is customized for binary-level DGF, which can accommodate the flawed CFGs. Leveraging the new concept of region, BinGo can retain the accuracy of reachability analysis while minimizing overhead.

CFG construction for COTS binaries. Several reverse engineering tools, such as IDA Pro [4], Angr [34], and Ghidra [6], construct CFGs by decompiling COTS binaries but struggle to recover indirect edges from calls and jumps, leading to inaccurate CFGs. Static methods have been developed to identify indirect call and jump targets. Type-based approaches [37] infer function pointer types, matching them with address-taken functions, while CCFIR [42] scans code addresses in stripped binaries. Tools like BAP [10] and CodeSurfer [8] trace value sets for abstract locations, and CALLEE [45] uses transfer and contrastive learning to resolve indirect call targets. However, due to imprecise type data, these solutions often construct inaccurate CFGs for COTS binaries. To enhance CFGs for DGF's reachability analysis, methods like ParmeSan [32], PDGF [43], Seive [35], and FishFuzz [44] incrementally add indirect edges to CFGs and update reachability. However, these methods rely on source code to perform compile-time instrumentation, which is infeasible in our case of binary-only fuzzing. Additionally, PDGF only handles a limited number of indirect edges, which hinder its effectiveness in correcting reachability analysis errors. To support directed fuzzing, BinGo employs a heuristic method combined with a Bayesian-based approach to mitigate the impact of inaccurate indirect edges in dynamic fuzzing.

VIII. Conclusion

In this paper, we propose BinGo, a tailored directed greybox fuzzer for COTS binaries. BinGo optimizes the binary-level reachability analysis of basic blocks on the region graph and leverages a new fitness metric—probabilistic reachability to prioritize the blocks and seeds with high reachability. Our approach can retain the accuracy of reachability analysis while minimizing overhead at the binary level. Extensive experiments have demonstrated that BinGo can reach target locations and expose known vulnerabilities faster than baseline fuzzers (AFL++, AFLGo$_{cots}$, PDGF, UAFuzz, and 1dVul). BinGo also discovered three new real-world vulnerabilities.

ACKNOWLEDGMENTS

This work is carried out with the support of the Hunan Provincial Key Laboratory of Intelligent and Parallel Analysis for Software Security, and is partially supported by the Science and Technology Innovation Program of Hunan Province (2024RC3136), the National Natural Science Foundation China (62272472, U22B2005, 62306328), the National Key Research and Development Program of China under Grant No.2021YFB0300101, the National University of Defense Technology Research Project (ZK23-14), the HUNAN Province Natural Science Foundation (2021JJ40692), and the Research Project of Key Laboratory of the State Administration of Science, Technology and Industry for National Defense (WDZC20245250105).

REFERENCES

[1] "Global'2023 software buying trends." 2023, https://www.gartner.com/en/digital-markets/insights/2023-global-software-buying-trends.

[2] "cscope files," 2024, https://sourceforge.net/projects/cscope/files/cscope/v15.9/.

[3] "google/bindiff," 2024, https://github.com/google/bindiff.

[4] "hey-rays," Apr. 2024, https://hex-rays.com/download-center.

[5] "Qemu-afl," 2024, https://github.com/AFLplusplus/qemuafl.

[6] "ghidra," May. 2025, https://github.com/NationalSecurityAgency/Ghidra.

[7] "llvm," Apr. 2025, https://llvm.org/.

[8] G. Balakrishnan, R. Gruian, T. Reps, and T. Teitelbaum, "Codesurfer/x86—a platform for analyzing x86 executables," in *International conference on compiler construction*. Springer, 2005, pp. 250–254.

[9] M. BoHme, V. T. Pham, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Acm Sigsac Conference on Computer & Communications Security*, 2017, pp. 2329–2344.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "Bap: A binary analysis platform," in *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*. Springer, 2011, pp. 463–469.

[11] M. Böhme, "Directed greybox fuzzing with afl," 2023, https://github.com/ aflgo/aflgo.

[12] S. Canakci, N. Matyunin, K. Graffi, A. Joshi, and M. Egele, "Targetfuzz: Using darts to guide directed greybox fuzzers," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 561–573. [Online]. Available: https://doi.org/10.1145/3488932.3501276

[13] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, "Hawkeye: Towards a desired directed grey-box fuzzer," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS 2018, Toronto, ON, Canada, October 15-19, 2018*. ACM, 2018, pp. 2095–2108. [Online]. Available: https://doi.org/10.1145/3243734.3243849

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

[15] Z. Du, Y. Li, Y. Liu, and B. Mao, "Windranger: A directed greybox fuzzer driven by deviationbasic blocks," in *ICSE '22: 44st International Conference on Software Engineering*. ACM, 2022.

[16] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020. [Online]. Available: https://www.usenix.org/conference/woot20/presentation/fioraldi

[17] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *International Symposium on Software Testing and Analysis*, 2012. [Online]. Available: https://api.semanticscholar.org/CorpusID:14374373

[18] H. Huang, Y. Guo, Q. Shi, P. Yao, R. Wu, and C. Zhang, "Beacon: Directed grey-box fuzzing with provable path pruning," in *The 43rd IEEE Symposium on Security and Privacy(S&P'22)*, May 2022.

[19] R. Johnson, D. Pearson, and K. Pingali, "The program structure tree: computing control regions in linear time," *SIGPLAN Not.*, vol. 29, no. 6, p. 171–185, Jun. 1994. [Online]. Available: https://doi.org/10.1145/773473.178258

[20] J. Kim and J. Yun, "Poster: Directed hybrid fuzzing on binary code," in *the 2019 ACM SIGSAC Conference*, 2019.

[21] S. H. Kim, C. Sun, D. Zeng, and G. Tan, "Refining indirect call targets at the binary level," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021.

[22] T. E. Kim, J. Choi, K. Heo, and S. K. Cha, "Dafl: Directed grey-box fuzzing guided by data dependency," in *Proceedings of the 32nd USENIX Conference on Security Symposium*, ser. SEC '23. USA: USENIX Association, 2023.

[23] G. Lee, W. Shim, and B. Lee, "Constraint-guided directed greybox fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 3559–3576. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[24] P. Li, W. Meng, and C. Zhang, "Sdfuzz: Target states driven directed fuzzing," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024. [Online]. Available: https://www.usenix.org/conference/usenixsecurity24/presentation/li-penghui

[25] Y. Li, S. Ji, Y. Chen, S. Liang, W. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, K. Lu, and T. Wang, "UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers," in *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*. USENIX Association, 2021, pp. 2777–2794. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[26] Y. Li, S. Ji, C. Lyu, Y. Chen, J. Chen, Q. Gu, C. Wu, and R. Beyah, "V-fuzz: Vulnerability prediction-assisted evolutionary fuzzing for binary programs," *IEEE Transactions on Cybernetics*, vol. 52, no. 5, pp. 3745–3756, 2022.

[27] P. Lin, P. Wang, X. Zhou, W. Xie, G. Zhang, and K. Lu, "Deepgo: Predictive directed greybox fuzzing," in *Network and Distributed System Security (NDSS) Symposium 2024 26 February - 1 March 2024, San Diego, CA, U*. The Internet Society. [Online]. Available: https://www.ndss-symposium.org/wp-content/uploads/2024-514-paper.pdf

[28] C. Luo, W. Meng, and P. Li, "Selectfuzz: Efficient directed fuzzing with selective path exploration," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[29] S. Nagy, A. Nguyen-Tuong, J. D. Hiser, J. W. Davidson, and M. Hicks, "Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1683–1700. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/nagy

[30] M. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for use-after-free vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses, RAID 2020, San Sebastian, Spain, October 14-15, 2020*. USENIX Association, 2020, pp. 47–62. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/nguyen

[31] M.-D. Nguyen, S. Bardin, R. Bonichon, R. Groz, and M. Lemerre, "Binary-level directed fuzzing for Use-After-Free vulnerabilities," in *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. San Sebastian: USENIX Association, Oct. 2020, pp. 47–62. [Online]. Available: https://www.usenix.org/conference/raid2020/presentation/nguyen

[32] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "ParmeSan: Sanitizer-guided greybox fuzzing," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2020, pp. 2289–2306. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/osterlund

[33] J. Peng, F. Li, B. Liu, L. Xu, B. Liu, K. Chen, and W. Huo, "1dvul: Discovering 1-day vulnerabilities through binary patches," in *49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2019, Portland, OR, USA, June 24-27, 2019*. IEEE, 2019, pp. 605–616. [Online]. Available: https://doi.org/10.1109/DSN.2019.00066

[34] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “Sok: (state of) the art of war: Offensive techniques in binary analysis,” 2016.

[35] P. Srivastava, S. Nagy, M. Hicks, A. Bianchi, and M. Payer, “One fuzz doesn’t fit all: Optimizing directed fuzzing via target-tailored program state restriction,” in *Proceedings of the 38th Annual Computer Security Applications Conference*, ser. ACSAC ’22. New York, NY, USA: Association for Computing Machinery, 2022, p. 388–399. [Online]. Available: https://doi.org/10.1145/3564625.3564643

[36] V. van der Veen, D. Andriesse, E. Göktaş, B. Gras, L. Sambuc, A. Slowinska, H. Bos, and C. Giuffrida, “Practical context-sensitive cfi,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, New York, NY, USA, 2015.

[37] V. Van Der Veen, E. Göktas, M. Contag, A. Pawoloski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida, “A tough call: Mitigating advanced code-reuse attacks at the binary level,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 934–953.

[38] H. Wang, X. Xie, Y. Li, C. Wen, Y. Li, Y. Liu, S. Qin, H. Chen, and Y. Sui, “Typestate-guided fuzzer for discovering use-after-free vulnerabilities,” in *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 999–1010. [Online]. Available: https://doi.org/10.1145/3377811.3380386

[39] Y. Wang, X. Jia, Y. Liu, K. Zeng, and P. Su, “Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization,” in *Network and Distributed System Security Symposium*, 2020.

[40] C. Wen, H. Wang, Y. Li, S. Qin, Y. Liu, Z. Xu, H. Chen, X. Xie, G. Pu, and T. Liu, “Memlock: memory usage guided fuzzing,” in *ICSE ’20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 2020, pp. 765–777. [Online]. Available: https://doi.org/10.1145/3377811.3380396

[41] S. Yang, Y. He, K. Chen, Z. Ma, X. Luo, Y. Xie, J. Chen, and C. Zhang, “1dfuzz: Reproduce 1-day vulnerabilities with directed differential fuzzing,” ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 867–879. [Online]. Available: https://doi.org/10.1145/3597926.3598102

[42] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, “Practical control flow integrity and randomization for binary executables,” in *2013 IEEE symposium on security and privacy*. IEEE, 2013, pp. 559–573.

[43] Y. Zhang, Y. Liu, J. Xu, and Y. Wang, “Predecessor-aware directed greybox fuzzing,” in *2024 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, may 2024, pp. 40–40. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00040

[44] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, C. Cao, Y. Zhang, F. Toffalini, and M. Payer, “FISHFUZZ: Catch deeper bugs by throwing larger nets,” in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1343–1360. [Online]. Available: https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

[45] W. Zhu, Z. Feng, Z. Zhang, J. Chen, Z. Ou, M. Yang, and C. Zhang, “Callee: Recovering call graphs for binaries with transfer and contrastive learning,” in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 2357–2374.