AFLGopher: Accelerating Directed Fuzzing via Feasibility-Aware Guidance

Weiheng Bai
bai00093@umn.edu
University of Minnesota - Twin Cities
Minneapolis, MN, USA

Kefu Wu
wu000380@umn.edu
University of Minnesota - Twin Cities
Minneapolis, MN, USA

Qiushi Wu
qiushi.wu@ibm.com
IBM T.J. Watson Research Center
Yorktown Height, NY, USA

Kangjie Lu
kjlu@umn.edu
University of Minnesota - Twin Cities
Minneapolis, MN, USA

Abstract

Directed fuzzing is a useful testing technique that aims to efficiently reach target code sites in a program. It has a wide range of applications, such as assessing the severity of vulnerabilities, confirming bugs found by static analysis, reproducing existing bugs, and testing code changes. The core of directed fuzzing is the guiding mechanism that directs the fuzzing to the specified target. A general guiding mechanism adopted in existing directed fuzzers is to calculate the control-flow distance between the current progress and the target, and use that as feedback to guide the directed fuzzing. A fundamental problem with the existing guiding mechanism is that the distance calculation is *feasibility-unaware*. For instance, it always assumes that the two branches of an `if` statement have equal feasibility, which is likely not true in real-world programs and would inevitably incur significant biases that mislead directed fuzzing.

In this work, we propose *feasibility-aware* directed fuzzing named *AFLGopher*. Our new feasibility-aware distance calculation provides pragmatic feedback to guide directed fuzzing to reach targets efficiently. We propose new techniques to address the challenges of feasibility prediction. Our new classification method allows us to predict the feasibility of all branches based on limited traces, and our runtime feasibility-updating mechanism gradually and efficiently improves the prediction precision. We implemented *AFLGopher* and compared *AFLGopher* with state-of-the-art directed fuzzers including AFLGo, enhanced AFLGo, WindRanger, BEACON and SelectFuzz. *AFLGopher* is 3.76×, 2.57×, 3.30×, 2.52× and 2.86× faster than AFLGo, BEACON, WindRanger, SelectFuzz and enhanced AFLGo, respectively, in reaching targets. *AFLGopher* is 5.60×, 5.20×, 4.98×, 4.52×, and 5.07× faster than AFLGo, BEACON, WindRanger, SelectFuzz and enhanced AFLGo, respectively, in triggering known vulnerabilities.

1 Introduction

Directed fuzzing aims to generate inputs that steer program execution toward specific target locations — such as newly patched code, suspicious functions, or frames in crash stack traces. It has become a foundational technique in modern software security testing, widely used for patch validation [24, 41], vulnerability reproduction [6], and exploitability assessment [38].

Unlike coverage-based fuzzers that explore programs indiscriminately, directed fuzzers focus on specific destinations. Their goal is two-fold: (1) *reach the target location as quickly and reliably as possible*, and (2) *once reached, fuzz around the target to expose potential vulnerabilities*. This **reach-then-trigger** model enables effective use of limited fuzzing resources and is critical in scenarios like crash reproduction and patch testing.

To guide the search, most directed fuzzers build on coverage-guided fuzzers like AFL [29], incorporating distance metrics to prioritize seeds closer to the target. AFLGo [34], the seminal work in this space, formulates the problem as a global optimization: it computes control-flow graph (CFG) distances from each basic block to the target and uses simulated annealing to allocate more energy to closer seeds. Hawkeye [3] and WindRanger [7] improve this approach by incorporating indirect-call resolution and data-flow sensitivity, respectively. However, all these systems share a critical limitation: they are *feasibility-unaware*. They treat CFG distance as a proxy for reachability and assume all branches are equally likely to be taken.

Why prior work falls short. In practice, control-flow branches vary greatly in their likelihood of being executed. Consider the example in Listing 1, where a vulnerability lies inside a function behind a rarely satisfied token check. A fuzzer like AFLGo may repeatedly attempt to reach this target because it is "close" in terms of graph distance, despite the branch being highly improbable. This misalignment causes wasted effort and poor performance.

```
void processRequest(char* token) {
    if (strcmp(token, "admin-token") != 0) {
        return;
        // Reject request if token is not an exact match
    }
    dangerousFunction();
    // Target Position: Function that needs to be tested

}
```

Figure 1: A rare condition guards access to a vulnerability.

Recent learning-based fuzzers attempt to address this differently. DeepGo [15] and HyperGo [14] learn mutation and path-transition probabilities to better guide input generation, but they do not model the feasibility of specific branches or generalize across structurally similar conditions. BEACON [11] prunes statically unreachable paths using symbolic reasoning, but lacks probabilistic prioritization and cannot adapt at runtime. These systems either treat feasibility as binary or focus only on mutation-level feedback.

Feasibility-Aware Directed Fuzzing. We propose ***AFLGopher***, the first directed fuzzer that augments traditional distance-based guidance with explicit *branch feasibility estimation*. Rather than assuming that all control-flow paths are equally likely to be taken, *AFLGopher* learns the empirical reachability of branches through runtime execution traces, semantic clustering, and sequence modeling. These feasibility scores are then embedded directly into the distance metric used to prioritize fuzzing seeds—shifting guidance from structural proximity to a more principled objective: **expected success in reaching the target**. This reformulates directed fuzzing from a static optimization problem over control-flow structure into a *probabilistic optimization problem* over executable paths, aligning fuzzing effort with real-world execution likelihood.

To realize feasibility-aware guidance, *AFLGopher* must determine the likelihood that a given control-flow edge is taken during fuzzing. In practice, not all program statements require this modeling. If a statement has only one successor (e.g., straight-line code), its outgoing edge has a feasibility of 100%. Hence, we focus exclusively on statements with multiple possible successors—referred to in this paper as *branch statements*.

Definition of branch statement. A branch statement is any program point where execution may proceed along more than one distinct path, depending on input or runtime state. In our work, we classify three types of branch statements: (1) **conditional statements** such as `if` and `switch`, (2) **indirect calls or jumps**, and (3) **return statements**. However, since the destination of a return is statically determined by the call site, its feasibility is trivially 100%. Therefore, *AFLGopher* focuses on estimating feasibility for the first two classes: conditional branches and indirect calls.

By modeling the likelihood of taking each branch at these control points, *AFLGopher* can accurately estimate the true cost of reaching a target, prioritize high-probability paths, and avoid wasting effort on structurally short but semantically unreachable ones.

Challenges. To realize *AFLGopher*, we face several challenges. To predict the feasibility of the branches, we need ground-truth data—the traces collected through dynamic analysis. However, a fundamental problem with dynamic analysis is that it cannot be comprehensive, with a very limited coverage rate. Most branches will not be covered and will not have any traces. It is challenging to predict the feasibility of all branches based on the limited traces (**C1**). Second, fuzzing is a dynamic process; to ensure precision, the prediction should keep improving efficiently as more traces become available. Without an optimization mechanism, the directed fuzzing will likely stall and fail to reach the targets. Optimization should be a runtime mechanism, so ensuring efficiency becomes challenging (**C2**).

Our techniques. We propose several new techniques to address the challenges mentioned above. Our first technique is a new classification method that allows us to calculate the feasibility of conditional-statement branches that do not have traces. The idea is to predict their feasibility based on the most similar conditional-statement branches that have traces, since semantically similar conditional statements are supposed to share similar feasibility distribution on their branches. Our second technique is a new deep-learning-based method that leverages the ability of the Long Short-Term Memory (LSTM) model: perfectly suitable for single-direction sequence-sensitive problems. This method allows us to take the traces as function-call sequences and predict the feasibility of each candidate's indirect-call target. Our third technique is a highly efficient runtime-updating mechanism that gradually improves the precision of feasibility prediction. The idea is that as the directed fuzzing progresses, we will have more and more traces that can be used to improve the prediction precision. To ensure efficiency, we further develop on-demand prediction-model updating and incremental deep learning, which triggers updating only when necessary and updates only the relevant parts without retraining the prediction model.

We implement *AFLGopher* and rigorously evaluate it against several state-of-the-art directed fuzzing tools, including BEACON, WindRanger, SelectFuzz, and AFLGo. Recognizing that AFLGo lacks support for indirect call edges, we also develop an enhanced variant, termed AFLGo-ICall, which extends its capabilities to handle indirect calls.

Our experimental results demonstrate that *AFLGopher* significantly outperforms all baseline fuzzers in reaching target code locations, achieving speedups of 3.76×, 2.57×, 3.30×, 2.52×, and 2.86× over AFLGo, BEACON, WindRanger, SelectFuzz, and AFLGo-ICall, respectively. Moreover, in reproducing known vulnerabilities, *AFLGopher* again leads with improvements of 5.60×, 5.20×, 4.98×, 4.52×, and 5.07× across the same set of baselines.

While direct comparison with DeepGo is not entirely equitable—due to the unavailability of its source code—our detailed evaluation and discussion in section 6 substantiate that *AFLGopher* consistently outperforms DeepGo across shared benchmarks. We attribute this performance gain to *AFLGopher*'s principled guidance mechanism that integrates dynamic feasibility estimation, enabling more efficient navigation of semantically reachable paths in the fuzzing process.

In this paper, we make the following contributions:

- **Feasibility-aware directed fuzzing.** We introduce the first feasibility-aware distance metric that integrates branch feasibility into directed fuzzing guidance. This enables more accurate feedback and significantly improves efficiency in reaching target code locations.
- **Techniques for feasibility prediction.** We develop three complementary methods: (1) a semantic classification approach that predicts the feasibility of conditional branches even without direct execution traces, (2) an LSTM-based model that estimates the feasibility of indirect-call targets from limited traces, and (3) a lightweight runtime update mechanism that incrementally refines predictions as fuzzing progresses.
- **Implementation and evaluation.** We design and implement *AFLGopher*, a feasibility-aware fuzzer that combines static and dynamic analysis. Our extensive evaluation across real-world benchmarks shows that *AFLGopher* reduces time-to-target and time-to-exposure by factors of 2.5–5× compared to state-of-the-art directed fuzzers.
- **Artifact availability.** To foster open science and reproducibility, we will release the full implementation of *AFLGopher* after acceptance.

2 Background

2.1 Directed Fuzzing: Design and Limitations

Directed fuzzing guides test generation toward a specified set of target locations in the program — such as recently patched code, a crash site, or a statically flagged vulnerability. Among existing strategies, distance-guided fuzzing has emerged as the dominant design, pioneered by AFLGo [34].

AFLGo models directed fuzzing as an optimization problem: it statically computes the control-flow graph (CFG) distance between basic blocks and the target, and uses this distance to rank seeds. The fuzzer allocates more energy to inputs that appear closer to the target, using simulated annealing to balance exploration and exploitation over time.

Subsequent works, such as Hawkeye [3] and WindRanger [7], refine distance modeling by resolving indirect calls or incorporating data-flow dependencies. These systems remain grounded in the assumption that static proximity in the control-flow graph correlates with actual reachability. However, this assumption often breaks down in practice.

Critically, traditional distance-based fuzzers treat all branches and paths as equally feasible — ignoring that many are rarely or never taken under realistic inputs. As a result, fuzzers frequently prioritize syntactically short paths that are semantically unreachable. This gap between control-flow distance and true execution feasibility leads to ineffective exploration and wasted effort, particularly in programs with deep, input-sensitive logic.

2.2 The Need for Feasibility Awareness

While control-flow distance provides a convenient structural heuristic for prioritizing fuzzing inputs, it fails to capture the actual difficulty of reaching target code in practice. In many real-world programs, control-flow branches exhibit highly imbalanced feasibility: some are taken almost always, while others are rarely or never executed unless a specific input satisfies stringent conditions.

This discrepancy causes a critical failure mode in existing directed fuzzers. Because they treat all branches as equally viable, they often prioritize seeds that appear structurally close to the target but are routed through infeasible paths — resulting in wasted effort. Our empirical analysis reveals the severity of this problem: in our benchmarks we used in section 6, 96% of branch statements exhibit unequal feasibility across their outgoing edges. Nonetheless, existing directed fuzzers assign equal cost to both sides of a branch, ignoring this disparity.

The impact is substantial. According to the FuzzGuard study [43], over 91.7% of inputs generated by state-of-the-art directed fuzzers fail to reach the target location. To investigate this further, we manually analyzed execution traces of unreachable inputs and found that many were blocked by low-feasibility branches — such as `if` statements involving hard-coded tokens, size limits, or allocation failures.

These findings highlight a fundamental insight: *static proximity does not imply reachability*. A path with minimal CFG distance may, in reality, be effectively unreachable within the fuzzing time budget. To address this, we argue that directed fuzzers must shift from structural heuristics to **feasibility-aware guidance** — incorporating the likelihood that a path can actually be taken under mutation.

This motivates our design of *AFLGopher*, which integrates empirical feasibility into the core distance metric. In subsection 2.3, we present two key observations that ground this design: (1) path priority changes significantly when feasibility is considered, and (2) branch feasibility can be generalized across semantically similar structures.

2.3 Empirical Insights: Feasibility Influences Path Selection

To evaluate how branch feasibility affects fuzzing guidance, we conduct a detailed analysis of real-world programs and vulnerabilities. Our findings support two key insights: (1) incorporating feasibility changes which paths are prioritized, and (2) semantically similar branches tend to share feasibility characteristics, enabling generalization.

Feasibility alters prioritization. We analyze CVE-2020-11895 in `libming`, a real-world vulnerability that requires a specific path to reach the buggy function. Figure 2 shows three potential call traces leading to the target node T, with intermediate functions labeled A to E. Several functions contain conditional branches (A, B, C, E) or indirect calls (D), resulting in different levels of execution feasibility.

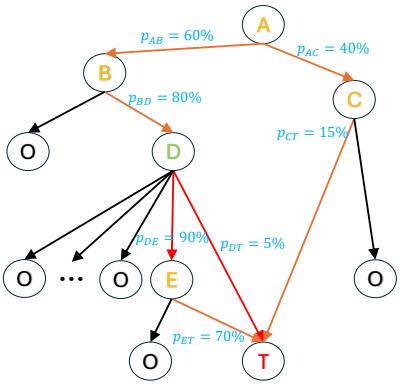

Figure 2: Call traces for exploiting the CVE-2020-11895. The node T represents the target function. The node $A - E$ represents the functions of the traces to the target. Node O represents all functions that are not included in the traces to the target. Node A, B, C, E has a IF statement that has two outgoing edges; node D indicates indirect calls and has several potential indirect call sites.

Using AFLGo's static CFG-based distance calculation, Trace 1 (`A` → `C` → `T`) appears shortest and is therefore prioritized. However, our feasibility-aware analysis reveals that Trace 3 (`A` → `B` → `D` → `E` → `T`)—though longer structurally—has much higher likelihood of successful execution due to more favorable branch conditions and indirect-call targets. By computing the empirical feasibility of each control-flow edge using AFL trace coverage, we quantify this as:

$$P_{ij} = \frac{\text{Hit}_j}{\text{Hit}_i} \tag{1}$$

Where Hit_i is the number of times node i is reached, and Hit_j is the number of times control flows from i to j. Using this metric, *AFLGopher* reprioritizes traces based on both path structure and actual feasibility, avoiding misleading shortcuts and focusing effort where success is more likely.

Semantically similar branches exhibit similar feasibility. Prior work has demonstrated that semantically similar code fragments tend to exhibit similar behavior or functionality. For instance, Pei et al.[23] and Ming et al.[20] show that structurally and semantically similar functions yield comparable execution traces, while Tufano et al. [30] leverage semantic embeddings to transfer program repair patterns across code segments. These findings suggest that semantic similarity can serve as a proxy for functional similarity, even across lexically diverse code.

Motivated by this insight, we hypothesize that *semantically similar conditional branches tend to share similar execution feasibility* — that is, the likelihood of triggering a particular branch direction during fuzzing. For example, branches guarding resource allocation failures (e.g., `malloc` or `xmlAlloc`) or dictionary initialization typically exhibit low feasibility on their failure side, while accepting branches are taken more frequently. We posit that this asymmetry is not unique to specific branches but consistent across semantically similar ones throughout the program.

However, as shown in Listing 3, we find that surface-level semantics of the condition itself (e.g., `if(!inbuffer)` or `if(out == NULL)`) are insufficient to accurately determine similarity. Although these two branches perform equivalent checks, the variables involved—`inbuffer` and `out`—do not immediately reveal their semantics. To address this, *AFLGopher* introduces *context-sensitive backward analysis* (subsection 4.1) to trace each condition variable to its source, capturing more meaningful semantic context, such as whether it originates from a memory allocator.

```
void foo1(){
        ...
        inbuffer = malloc(insize);
        ...
        if(!inbuffer){...}  /*if-1*/
        ...
        Buffer out = (Buffer)malloc(BUFFER_SIZE);
        ...
        if(out == NULL){...}  /*if-2*/
    }

```

Figure 3: Two semantically similar branches guarded by memory allocation checks.

To validate this, we conducted an empirical study across seven diverse open-source programs spanning domains such as parsing, compression, memory management, and system utilities. From these programs, we curated a stratified benchmark of 1,000 conditional branch statements. These findings are intended to provide empirical, directional guidance for the feasibility generalization approach employed in *AFLGopher*. A detailed description of our sampling methodology and validation strategy is provided in subsubsection 6.1.2.

To cluster the branches, we applied a semantic taxonomy based on three axes (detailed in subsection 4.1):

- *Condition variable role*, such as return-value checks (e.g., `if (malloc(...) == NULL)`), pointer checks, global state flags, and struct field access;
- *Comparison structure*, including null checks, integer thresholds, range bounds, and bitmask operations;
- *Usage context*, inferred from identifier names and API calls, such as memory allocation, file I/O, parsing, error propagation, and resource availability.

This clustering process resulted in 46 distinct semantic clusters. All branch labels were independently annotated and reviewed by three experienced graduate-level C/C++ developers (each with 5+ years of experience), and disagreements were resolved by majority vote to ensure consistency and quality.

Each program was fuzzed using AFL, and dynamic execution traces were collected to measure the feasibility of each branch — defined as the relative frequency of its outgoing edges being taken. To assess whether semantic similarity aligns with feasibility, we compared our manual semantic clusters with feasibility-based groupings using the Adjusted Rand Index (ARI). Our clustering achieved an ARI of 0.91, indicating excellent alignment. In contrast, random clusterings of the same data yielded an average ARI of 0.29.

These results strongly support our hypothesis: **branches with similar semantics tend to share similar feasibility**. This empirical observation validates the use of semantic clustering in *AFLGopher* to generalize feasibility estimates across semantically similar but previously unexplored branches, enabling more informed and efficient fuzzing guidance.

3 Overview of *AFLGopher*

The primary goal of directed fuzzing is to reach a specified target location in the program as efficiently as possible. Traditional distance-guided fuzzers such as AFLGo prioritize inputs based on static control-flow distance. However, these approaches fail to account for whether the paths are actually feasible under fuzzing conditions, often wasting resources on unreachable branches. *AFLGopher* addresses this limitation by estimating the empirical *feasibility* of branches and integrating this information into its guidance logic.

Figure 4 illustrates the architecture of *AFLGopher*, which is structured into three main phases: (1) *static conditional-statement classification*, (2) *runtime branch-feasibility prediction*, and (3) *runtime branch-feasibility update*. These phases operate as a closed feedback loop to progressively guide fuzzing toward feasible paths leading to the target.

Phase 1: Static Analysis and Conditional Statement Classification. Given a target program, *AFLGopher* first generates its LLVM IR and performs backward analysis to extract the source values of all conditional branch statements. These statements are grouped into semantically similar clusters based on attributes such as the role of the condition variable (e.g., pointer validity, return value checks), the comparison structure (e.g., null checks, threshold comparisons), and the usage context (e.g., memory allocation, I/O). An interprocedural control-flow graph (CFG) is also constructed to enable downstream distance computation. The resulting semantic clusters and CFG form the static inputs to the fuzzer.

Phase 2: Runtime Branch-Feasibility Prediction. During fuzzing, *AFLGopher* instruments and executes the program on input seeds.

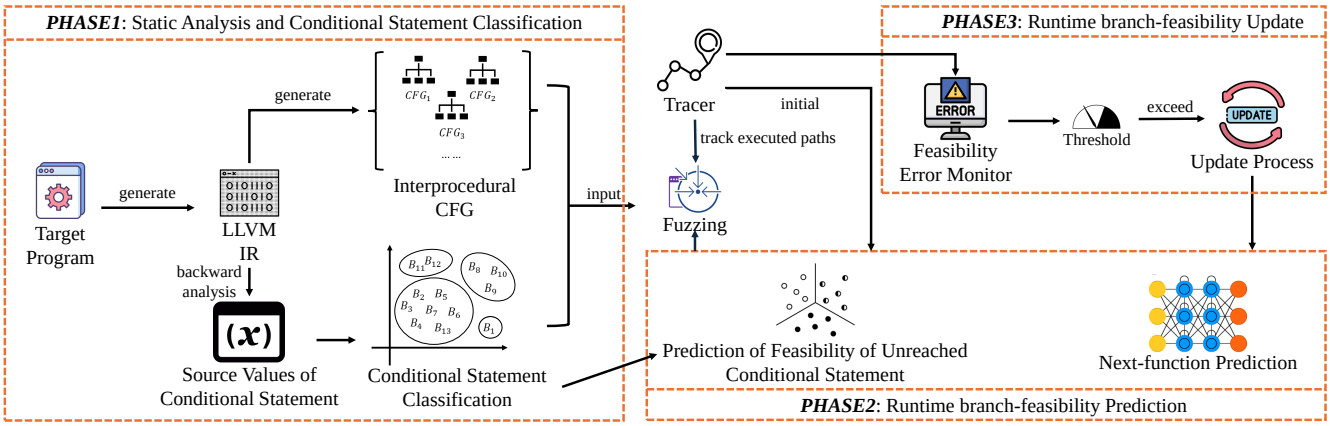

Figure 4: Overview of *AFLGopher*.

A tracer records which branches are exercised in the generated execution traces. For uncovered branches, *AFLGopher* estimates feasibility using two mechanisms: (i) for conditional statements, feasibility is generalized from the observed behavior of semantically similar branches; and (ii) for indirect calls, *AFLGopher* applies a next-function prediction model based on LSTM to estimate the probability of reaching a given call target. These feasibility scores are then embedded into a **feasibility-aware distance metric**, which replaces static CFG distance for seed ranking.

Phase 3: Runtime Branch-Feasibility Update. As fuzzing proceeds, the system continuously monitors the prediction error between observed and expected branch behavior. If the error rate exceeds a predefined threshold, the *Feasibility Error Monitor* triggers the *Update Process*, which incrementally refines the feasibility scores—either by re-evaluating conditional clusters or updating the LSTM prediction model. This allows *AFLGopher* to adapt in real time as new execution paths are uncovered.

Fuzzing Guidance Loop. Throughout the fuzzing campaign, the feasibility-aware distance serves as the core signal for prioritizing seed inputs. Seeds that lie on paths with high feasibility and structural proximity to the target receive more mutation energy, as governed by the power schedule. This allocation strategy ensures that *AFLGopher* systematically prioritizes promising paths that are both reachable and close—significantly improving its ability to reach deep or guarded target locations with minimal wasted effort.

4 Design of *AFLGopher*

AFLGopher is designed to accelerate directed fuzzing by prioritizing execution paths that are not only structurally close to the target but also empirically feasible. This section describes the key components of *AFLGopher* and how it addresses two central challenges: (**C1**) how to predict the feasibility of unexplored branches, and (**C2**) how to refine these predictions at runtime as new execution traces become available.

To address (**C1**), *AFLGopher* introduces two core techniques: (1) *semantic clustering of conditional statements*, enabling generalization of feasibility across similar branches, and (2) *next-function prediction* for estimating the reachability of indirect call targets. To handle (**C2**), *AFLGopher* incorporates a lightweight runtime *feasibility updating mechanism* that adapts predictions as coverage improves.

4.1 Semantic Clustering of Conditional Statements

The feasibility of a conditional branch is often governed not just by its syntax, but by the semantics of the data it evaluates. Our analysis and manual study in subsection 2.3 across real-world software reveal that the behavior of conditional branches is heavily influenced by the *source* of the condition variable. To enable generalization of feasibility across uncovered code, *AFLGopher* performs static semantic clustering of conditional statements based on their source characteristics.

We design a lightweight yet expressive taxonomy to capture the most common sources of conditional variables that affect fuzzing feasibility. Through a *context-sensitive backward analysis* of LLVM IR across seven diverse open-source programs (including parsers, compressors, and system utilities), we identify five dominant categories that account for over 97% of condition variable sources:

- **Function return values** — e.g., `malloc()`, `fopen()`, `xmlParse()`; often influence error-handling paths with high feasibility asymmetry (success is common, failure is rare).
- **Global variables** — e.g., initialization flags, configuration toggles; typically require specific program state setup, making certain branches hard to reach through input mutation alone.
- **Function arguments** — passed directly from input, commonly used in range, size, or validity checks.
- **Struct or object fields** — e.g., `req->hdr.len`; prevalent in input parsers or network handlers where correct structure fields govern execution paths.
- **Locals derived from input** — local variables calculated from input arguments or structs, such as `offset + size`; frequently used in nested logic and boundary guards.

These categories were chosen to strike a balance between semantic expressiveness, generalization capability, and implementation simplicity. While additional cases (e.g., environment variables or

file descriptors) exist, they appear infrequently or are hard to analyze without dynamic context. Our taxonomy is extensible but empirically sufficient for broad applicability.

To extract these semantics, *AFLGopher* performs a *context-sensitive backward analysis* over LLVM IR to trace each condition variable to its origin. From the source, it extracts a semantic string that includes the variable type, variable name, and, if applicable, the associated function name or struct. These strings are then tokenized and embedded using a pre-trained Word2Vec model. Each conditional statement is represented as a concatenated embedding vector of its semantic tokens.

We cluster these semantic vectors using the DBSCAN algorithm. DBSCAN is chosen for its ability to find arbitrarily shaped clusters without requiring the number of clusters to be specified upfront—an important property for unsupervised classification over diverse programs. Additionally, DBSCAN is robust to noise and outliers, making it well-suited for handling imperfect static data.

The resulting clusters group conditional statements that are semantically similar in terms of how they guard program logic. These clusters form the foundation for *AFLGopher* 's feasibility generalization mechanism at runtime: when an uncovered branch belongs to a known cluster, *AFLGopher* estimates its feasibility based on the aggregated behavior of covered branches in the same group. This approach allows *AFLGopher* to extend feasibility knowledge to parts of the program that have not yet been executed, improving both guidance quality and fuzzing efficiency (detail in subsection 4.2).

4.2 Feasibility Prediction

At runtime, *AFLGopher* estimates the feasibility of both covered and uncovered branches using two distinct strategies:

Conditional Branches Feasibility Prediction. For conditional statements, *AFLGopher* collects execution traces and aggregates feasibility statistics within each semantic cluster. Specifically, it computes the average feasibility of all covered branches in a cluster and assigns this value to its uncovered members. This reflects our hypothesis that semantically similar conditionals—such as those checking the return value of memory allocation functions—are likely to exhibit similar branch-taking behavior. When a cluster contains no covered branches (e.g., early in fuzzing), we assign it the feasibility of the *nearest cluster* in vector space, measured by centroid distance. This fallback is motivated by the observation that clusters close in semantic embedding space often share similar operational intent. Compared to alternatives such as using a global average or uniform default, nearest-cluster inference preserves semantic locality and avoids diluting the guidance signal with unrelated behavior.

Indirect Call Feasibility Prediction. Unlike prior work such as CALLEE [40], which attempts to dynamically resolve indirect call targets during execution, *AFLGopher* does not predict *which* function will be invoked at runtime. Instead, it statically enumerates all possible indirect call targets and predicts the *feasibility* of exercising each edge—i.e., the feasibility (likelihood) that a given target will actually be taken under fuzzing.

To construct the candidate set of indirect call targets, *AFLGopher* integrates Multi-Layer Type Analysis (MLTA) [17]. MLTA is specifically chosen because it guarantees **zero false negatives**, ensuring that no legitimate targets are omitted. Compared to conventional pointer or alias analysis, MLTA provides a sound and efficient over-approximation of the indirect call graph. This results in a statically computed call graph where each indirect call site is mapped to a complete set of feasible target functions.

The core challenge, then, is not target discovery—but estimating *how likely* each of these statically identified targets is to be reached during fuzzing. This task presents three distinct difficulties: (1) each indirect call may dispatch to multiple targets; (2) the static context surrounding each indirect call target is often identical—since all targets are invoked from the same call site—making it ineffective to distinguish their feasibility using static features. As a result, we cannot apply the same semantic clustering strategy used for conditional statements, which relies on variation in static context to generalize feasibility.; and (3) early fuzzing coverage is sparse, limiting the training data available for target-specific learning.

To address this, *AFLGopher* introduces a technique we call *next-function feasibility prediction*, inspired by next-word prediction in natural language processing. Execution traces are collected and tokenized as sequences of function calls, where the goal is to predict the feasibility of each indirect call edge based on the preceding call chain. For example, given a trace like `main` → `parse` → `dispatch` → `?`, the model learns to assign probabilities to candidate targets observed at the final indirect call site.

We implement this prediction using a single-direction Long Short-Term Memory (LSTM) network. LSTM is selected over alternatives such as Bi-LSTM or Transformer-based models for several practical reasons. First, the function-call traces are inherently directional—only the forward context is relevant at the time of the indirect call. Bi-LSTM introduces backward dependencies that are semantically meaningless in this context. Second, LSTMs are more computationally efficient for the short sequences we typically observe (10–200 tokens), with $O(nd^2)$ complexity compared to $O(n^2d)$ for Transformer-based models. Finally, LSTMs support online and incremental learning, making them well-suited for the evolving nature of fuzzing where traces are collected continuously and model updates must be lightweight.

To overcome the initial lack of indirect call training data, *AFLGopher* employs *predominating functions*—functions that frequently appear in traces leading to indirect calls—as proxy labels. These allow early-stage training while preserving the directionality and structure of traces. As coverage improves and actual indirect call targets are reached, the model transitions to learning target-specific feasibility scores. The LSTM outputs a softmax distribution over candidate targets for each call site; these scores are used as feasibility estimates for edge weighting in the call graph.

Model updates are coordinated by *AFLGopher* 's runtime error monitoring framework (detail in subsection 4.3). When the prediction error exceeds a threshold θ_{IC}, the model is retrained incrementally using newly collected traces. The LSTM module is implemented in PyTorch and integrated with the main fuzzing loop via lightweight Inter-Process Communication (IPC), allowing predictions and updates to run asynchronously without degrading fuzzing throughput.

4.3 Runtime Feasibility Update

As fuzzing progresses, new execution traces continually reveal branches and indirect call targets that were previously uncovered. To ensure guidance remains accurate and effective, *AFLGopher* employs a dynamic runtime updating mechanism that refines feasibility predictions based on observed behavior. Without such adaptivity, static feasibility estimates risk becoming outdated or misleading—particularly in programs with highly skewed or input-sensitive control flow.

AFLGopher's update mechanism is driven by an error-monitoring system that evaluates the divergence between prior predictions and newly observed behavior. When prediction errors exceed predefined thresholds, the system triggers selective updates to restore model accuracy and improve future guidance. We separately monitor prediction error for (1) conditional statement clusters and (2) indirect call targets, using distinct but harmonized metrics.

Conditional Branch Monitoring. For conditional statements, *AFLGopher* tracks the prediction error for each semantic cluster. We define Err_{CSC} as the relative change in feasibility between two update cycles, calculated as

$$Err_{CSC} = \frac{|P' - P|}{P} \tag{2}$$

where P and P' are the old and new feasibility estimates for the cluster. If a cluster's error exceeds a threshold θ_{CSC}, it is marked as unstable. We then compute the overall group error rate,

$$Err_G = \frac{N'}{N} \tag{3}$$

where N' is the number of unstable clusters and N is the total number of clusters.

When $Err_G > \theta_G$, *AFLGopher* initiates an update: the average feasibility for each cluster is recalculated, and feasibility-aware distances are recomputed accordingly. This ensures that fuzzing energy remains focused on truly promising paths rather than stale estimations.

Indirect Call Monitoring. For indirect-call targets, *AFLGopher* leverages a prediction model trained on function-call sequences. We monitor the model's accuracy θ_{acc} before and after each training round. The prior error rate is defined as

$$Err_{IC} = 1 - \theta_{acc} \tag{4}$$

and the new error rate as Err'_{IC}. When

$$|Err_{IC} - Err'_{IC}| > \theta_{IC} \tag{5}$$

we trigger model retraining using the most recent execution traces.

Update Execution. Updates occur after each fuzzing cycle (as defined by AFL), but are only executed if thresholds are exceeded. Once triggered, the following steps occur: (1) feasibility scores for relevant edges are updated, (2) the dynamic distance table is refreshed with new weights, and (3) guidance rankings for seeds are recalculated. This ensures that updates remain lightweight and do not interfere with fuzzing throughput.

Threshold Selection. Optimal thresholds are determined empirically using the control variate method across nine real-world programs. We varied one threshold at a time and observed fuzzing effectiveness and update cost. Results indicate that setting $\theta_{CSC} = 10\%$ and $\theta_G = 3\%$ provides a good trade-off between update frequency and prediction accuracy. Lower thresholds resulted in excessive updates and performance overhead; higher thresholds led to outdated guidance. Users may adjust these parameters for domain-specific tuning (detail in subsection 6.6).

Dynamic Distance Table. To efficiently reflect updates in fuzzing guidance, *AFLGopher* replaces AFLGo's static distance instrumentation with a dynamic distance table. This table stores basic block and function edge weights in memory and is updated in-place when feasibility changes. This eliminates the need to recompile the target binary and significantly reduces guidance update latency.

Incremental Learning for Indirect Call Feasibility. To support timely and efficient updates to the next-function prediction model, *AFLGopher* employs incremental learning. New traces are used to retrain the LSTM model without discarding prior knowledge. This reduces retraining time and ensures the model adapts gracefully as new indirect call targets are discovered. The model is implemented in PyTorch and integrated via IPC with the fuzzer, allowing asynchronous prediction and retraining without largely impacting fuzzing speed.

5 Implementation

AFLGopher is implemented atop LLVM and AFLGo, with key components written in C++, Python, and PyTorch. The system consists of four primary modules: (1) static analysis and interprocedural graph construction, (2) normalization and classification of conditional statements, (3) feasibility-aware distance calculation, and (4) a dynamic runtime updating mechanism. Below, we detail the key implementation considerations for each module.

5.1 Constructing the Interprocedural CFG

Accurate interprocedural control-flow representation is critical for directed fuzzing, especially when indirect calls and nested function interactions are involved. *AFLGopher* constructs an interprocedural control-flow graph (Inter-CFG) by combining intra-procedural control-flow graphs (CFGs) with a statically resolved call graph (CG).

To resolve indirect calls, we integrate Multi-Layer Type Analysis (MLTA) [17], which provides a sound and efficient overapproximation with **zero false negatives**. This guarantees that all feasible indirect call targets are preserved. Compared to alias or pointer analysis, MLTA strikes an effective balance between precision and coverage. We integrate MLTA with LLVM's IR to build the Inter-CFG, which is then used for distance computations and feasibility-aware guidance throughout fuzzing.

5.2 Normalization of Conditional Statements

Prior to semantic clustering and feasibility prediction, conditional statements must be normalized to ensure consistency. Syntactically different but semantically equivalent conditions can lead to divergent feasibility estimates if not handled correctly. For example, consider `if (!x)` vs. `if (x == NULL)` — both check for nullity, but

without normalization, they may be assigned different feasibility labels.

To address this, *AFLGopher* treats conditions using ==, >, and < as standard forms. Conditions with !=, !, <=, and >= are transformed by inverting the branch feasibility assignment. For compound conditionals using && or ||, LLVM IR naturally decomposes them into atomic conditions, which *AFLGopher* handles as separate single-branch checks during analysis and modeling.

5.3 Feasibility-Aware Distance Calculation

AFLGopher extends AFLGo's distance computation methodology by modifying how edge weights are assigned. In AFLGo, all edges are weighted uniformly (typically with weight 1), which leads to feasibility-unaware guidance. In contrast, *AFLGopher* dynamically assigns edge weights based on empirical or predicted feasibility, with lower weights indicating higher likelihood of execution.

Basic Block Level. At the intra-procedural level, each edge between basic blocks is weighted using the reciprocal of its feasibility:

$$W_{\text{BB\_edges}} = \frac{1}{P_{\text{BB}}} \tag{6}$$

where P_{BB} is the empirical or predicted feasibility of the branch. High-feasibility edges thus have lower weights, biasing the distance metric toward paths that are both short and likely to be executed.

Function Level. Function-level edge weights are more nuanced, as function calls may be gated by one or more conditional branches. To account for this, we introduce the concept of a *Conditional Statement Chain (CSC)*—a series of conditional statements that dominate a particular function call site. We statically identify CSCs during control-flow analysis.

For each function call, we compute its feasibility P_F based on the enclosing CSCs:

$$P_F = \begin{cases} 1 & \text{if } FC \notin \text{any CSC}, \\ \max\limits_i \left(\prod\limits_{j=1}^{n} P_{ij} \right) & \text{if } FC \in \{CSC_i\}, \end{cases} \tag{7}$$

where P_{ij} is the feasibility of the j-th condition in CSC i. We take the maximum across CSCs to favor the most permissive path. The function-level edge weight is then computed as

$$W_{F\_edges} = \frac{1}{P_F} \tag{8}$$

, ensuring that paths through easily satisfiable conditions are prioritized.

5.4 Dynamic Distance Table

To support runtime feasibility updates, *AFLGopher* replaces AFLGo's static distance instrumentation with a dynamic in-memory distance table. This table stores weights for both basic block and function edges and is initialized at program startup. When new feasibility estimates are computed (via trace analysis or model updates), only the relevant entries in the table are updated. This avoids recompilation and allows the fuzzer to respond to new information in near real-time.

5.5 Runtime Prediction and Incremental Model Updates

For indirect-call edges, *AFLGopher* predicts edge feasibility using an LSTM-based model trained on dynamic function-call traces. The model is implemented in PyTorch and runs as a standalone module, interfacing with the fuzzer through a dedicated tracker. This tracker monitors execution coverage, enabling real-time updates of feasibility information.

During fuzzing, the fuzzer extracts the current function-call sequence and sends it to the prediction module. The LSTM processes this sequence and returns a softmax vector over the statically collected indirect call targets (from MLTA). Each entry in the vector represents the predicted feasibility of reaching a specific target. These scores are then translated into edge weights using the reciprocal transformation described in Equation 6, and used to update the feasibility-aware distance table in memory.

To ensure responsiveness, the LSTM model is trained incrementally. New execution traces are batched and used to update the model from its last checkpoint, avoiding costly retraining. Updates are triggered by *AFLGopher* 's error monitoring system (subsection 4.3) whenever prediction error exceeds a defined threshold. Because both training and inference operate asynchronously from the main fuzzing loop, they introduce no measurable slowdown in fuzzing throughput.

This integration of on-the-fly model refinement with in-place edge weight updates allows *AFLGopher* to continuously adapt its guidance strategy as new execution paths are discovered. It ensures that feasibility predictions remain accurate throughout the fuzzing campaign and that distance metrics reflect the evolving program state.

6 Evaluation

Directed fuzzing aims to reach and reproduce user-specified target locations—such as patched code or known vulnerabilities—as efficiently as possible. To evaluate the effectiveness of *AFLGopher* in fulfilling this objective, we conduct a large-scale experimental study. This section answers the following research questions:
RQ1: Does *AFLGopher* reach predefined target sites more effectively than state-of-the-art directed fuzzers?
RQ2: Is *AFLGopher* more effective in reproducing known bugs than competing approaches?
RQ3: How much do *AFLGopher* 's individual design components contribute to its overall effectiveness?

6.1 Evaluation Setup

All experiments are performed on a virtual machine running Ubuntu 18.04 with 4GB RAM and 50GB disk space, hosted on a 64-bit Windows 11 Pro system with an Intel Core i9-12900K @ 3.20GHz, 128GB of RAM, and a 24GB Nvidia RTX 3090Ti GPU. The fuzzing environment aligns with prior fuzzing research such as Healer [28].

6.1.1 Evaluation Baselines. We compare *AFLGopher* against a diverse set of baseline fuzzers:
• **AFLGo** [2] pioneered the concept of distance-guided directed fuzzing and remains a standard benchmark.

• **AFLGo-ICall** is our enhanced version that extends AFLGo to support indirect call edges in the call graph.
• **WindRanger** [7], which steers fuzzing toward deviation basic blocks;
• **BEACON** [11], which employs path pruning to exclude infeasible branches;
• **SelectFuzz** [19], which selectively explores code regions relevant to the target site.
• **DeepGo** [15], which claims to be open source, but we were unable to obtain or reproduce their source code at the time of writing. As such, we will compare with them based on the result provided in their paper.

6.1.2 Evaluation Metrics and Methodology. We follow prior work [2, 3, 7, 22, 43] in adopting two key **metrics**:
• **Time-to-Target (TTT)**: Time required for a fuzzer to first *reach* a predefined target location in the binary.
• **Time-to-Exposure (TTE)**: Time required to first *trigger* a known bug associated with the target.
Experimental Methodology. For each target site, we run each fuzzer 10 times with a fixed 24-hour time budget per run to reduce randomness and ensure statistical robustness. If a fuzzer fails to reach the target or expose the bug in all runs, we mark the result as *Timeout (T.O.)*. Meanwhile, for some target programs, the baseline fuzzer cannot compile, we will mark as **N/A**, which will exclude from the comparison of the performance.

To assess statistical significance, we apply the Mann–Whitney U test (p-value) [35]. We consider results statistically significant when $p < 0.05$, allowing us to reject the null hypothesis. Additionally, we use the Vargha–Delaney $\hat{A}_{12}$ effect size [31] to quantify whether one technique outperforms another. A value of $\hat{A}_{12} > 0.71$ is interpreted as a large performance difference.
Data sampling for conditional statement classification verification. To validate the consistency of feasibility across semantically similar conditional statements, we perform manual analysis on a representative benchmark dataset collected by us. Following practices from recent program analysis and fuzzing studies [1, 9, 10, 18, 32, 36], which typically analyze 40–400 samples, we conservatively select 1,000 conditional statements for inspection based on pre-defined rules (see subsection 2.3). All sampled conditionals are cross-validated by three developers with more than five years of C/C++ experience. This process ensures that our classification and feasibility evaluations are unbiased, statistically grounded, and reproducible. According to empirical sampling theory [5], this sample size yields a margin of error of approximately ±5%, offering strong statistical confidence for our consistency checks.

6.1.3 Benchmark Dataset. To ensure fairness, reproducibility, and comparability with prior work, we evaluate *AFLGopher* using the same benchmark datasets adopted by AFLGo [2] and widely used in recent directed fuzzing research [3, 7, 11, 15, 19, 22].
• **UniBench** [13] contains real-world programs from diverse domains along with curated seed corpora. This dataset has been adopted by several recent directed fuzzers, including WindRanger [7] and DeepGo [15]. Although DeepGo's implementation is not publicly available, we use the same target programs and positions to support a theoretical comparison under RQ1.
• **AFLGo Test Suite (ATS)**[1] was introduced in AFLGo's original work [2] to measure a fuzzer's ability to reach known vulnerabilities. It includes ground truth crash sites and target locations, making it well-suited for evaluating bug reproduction capabilities. To answer RQ2, we adopt the AFLGo test suite as a benchmark to assess how effectively each fuzzer can expose known vulnerabilities under time-constrained conditions.

By using these two well-established benchmarks, we ensure that our results are directly comparable to those of prior work, and that both target-reaching and bug-triggering aspects of directed fuzzing are thoroughly evaluated.

6.2 Effectiveness on Reaching Target Sites

To evaluate the effectiveness of *AFLGopher* in reaching predefined target sites, we tested 20 programs from the UniBench benchmark suite, covering a total of 80 unique target locations. For each target, we measured the Time-to-Target (TTT), defined as the time required for a fuzzer to reach the designated code location. Each experiment was executed with a 24-hour timeout threshold per run, and results were averaged over 10 independent runs. Full results are summarized in Table 2 (Due to font restriction, full results appear in the Appendix).

AFLGopher successfully reached 77 out of 80 target sites, substantially outperforming all baselines: AFLGo (7/80), BEACON (10/80), WindRanger (15/80), SelectFuzz (27/80), and AFLGo-ICall (29/80). Furthermore, *AFLGopher* achieved the shortest average TTT on 71 out of 80 targets, demonstrating its consistent superiority across a wide range of program types and target patterns.

In terms of speed, *AFLGopher* delivers substantial improvements, achieving 3.76×, 2.57×, 3.30×, and 2.52× average speedup in TTT compared to AFLGo, BEACON, WindRanger, and SelectFuzz, respectively. When compared to AFLGo-ICall, *AFLGopher* achieves a 2.86× speedup. Although DeepGo [15] is not publicly available for direct comparison, we approximate its relative performance based on the TTT results it reported. When normalized via AFLGo, BEACON, and WindRanger as anchor baselines, *AFLGopher* demonstrates estimated TTT improvements of 1.17×, 1.49×, and 1.82× over DeepGo, respectively.

To ensure statistical rigor, we conducted both the Mann–Whitney U test and the Vargha–Delaney effect size test. All $p - values$ were below 0.05, indicating statistically significant improvements. The mean $\hat{A}_{12}$ values of *AFLGopher* versus AFLGo, BEACON, WindRanger, SelectFuzz, and AFLGo-ICall were 0.78, 0.81, 0.86, 0.82, and 0.79, respectively—reflecting large effect sizes according to standard interpretation [31].

Conclusion: Across 80 diverse targets, *AFLGopher* consistently reaches more targets and does so significantly faster than all evaluated baselines, validating its advantage in target-oriented fuzzing.

6.3 Effectiveness of Vulnerability Reproduction

To evaluate how effectively *AFLGopher* exposes known vulnerabilities, we follow the same setup as BEACON [11], WindRanger [7], and DeepGo [15] by using the AFLGo test suite, which defines 20 real-world CVE-labeled vulnerabilities as target sites. These target

[1] https://github.com/aflgo/aflgo/tree/master/examples

Table 1: The Time-to-Exposure Results on Programs from ATS

Program	CVE-ID	AFLGo	BEACON	WindRanger	SelectFuzz	AFLGo-ICall	AFLGopher-C	AFLGopher-N	AFLGopher-U	AFLGopher
binutils2.26	2016-4487	0.16h	0.05h	0.14h	0.18h	0.17h	0.35h	0.42h	0.33h	0.33h
	2016-4488	0.4h	3.38h	0.26h	0.53h	0.39h	1.19h	1.21h	0.58h	0.57h
	2016-4489	0.27h	0.31h	0.42h	0.3h	0.3h	0.44h	0.75h	0.73h	0.72h
	2016-4490	0.1h	0.17h	0.28h	0.39h	0.17h	0.33h	0.47h	0.3h	0.3h
	2016-4491	*T.O.*	*T.O.*	*T.O.*	18.26h	23.24h	8.24h	*T.O.*	7.95h	4.11h
	2016-4492	0.9h	0.79h	0.16h	1.89h	0.92h	1.11h	1.64h	1.05h	0.45h
	2016-6131	*T.O.*	*T.O.*	21.4h	19.25h	*T.O.*	12.46h	*T.O.*	11.55h	4.25h
libming4.48	2018-8807	*T.O.*	11.86h	13.88h	18.18h	21.39h	16.4h	22.42h	11.3h	4.4h
	2018-8962	19.36h	8.12h	13.2h	14.42h	17.87h	13.9h	12.21h	7.73h	3.49h
	2018-11095	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	10.68h	4.78h
	2018-11225	*T.O.*	17.34h	*T.O.*	20.41h	22.99h	19.33h	*T.O.*	16.57h	4.72h
LibPNG1.5.1	2011-2501	0.79h	N/A	0.73h	1.84h	1.24h	1.95h	1.67h	1.32h	0.36h
	2011-3328	7.7h	N/A	4.32h	7.49h	7.64h	4.08h	3.61h	2.8h	1.35h
	2015-8540	0.06h	N/A	0.11h	1.97h	0.11h	0.4h	0.4h	0.4h	0.4h
xmllint2.9.4	2017-9047	*T.O.*	*T.O.*	*T.O.*	18.39h	*T.O.*	9.12h	12.48h	8.74h	4.27h
	2017-9048	*T.O.*	*T.O.*	*T.O.*	2.51h	2.27h	3.05h	*T.O.*	1.73h	0.57h
	2017-9049	*T.O.*	*T.O.*	*T.O.*	21.41h	*T.O.*	17.98h	*T.O.*	6.93h	4.4h
	2017-9050	*T.O.*	*T.O.*	*T.O.*	21.09h	*T.O.*	19.99h	*T.O.*	13.19h	4.17h
Lrzip0.631	2017-8846	*T.O.*	11.05h	17.74h	21.1h	*T.O.*	12.13h	*T.O.*	11.64h	4.49h
	2018-11496	22.46h	9.3h	19.11h	22.31h	21.91h	11.99h	9.62h	7.93h	4.01h
speedup		**5.60x**	**5.20x**	**4.98x**	**4.52x**	**5.07x**	**3.42x**	**4.97x**	**2.37x**	-
mean *p-values*		**0.03**	**0.002**	**0.004**	**0.003**	**0.006**	**0.014**	**0.011**	**0.017**	-
mean $\hat{A}_{12}$		**0.76**	**0.83**	**0.89**	**0.80**	**0.81**	**0.79**	**0.86**	**0.88**	-

locations and the corresponding Time-to-Exposure (TTE) results are presented in Table 1.

Among all evaluated fuzzers, *AFLGopher* uncovered the most vulnerabilities (20/20), outperforming AFLGo (11), BEACON (10), WindRanger (13), SelectFuzz (19), and AFLGo-ICall (16). Moreover, *AFLGopher* achieved the shortest TTE on 15 out of 20 targets, demonstrating its consistent efficiency across a diverse set of real-world programs.

In terms of mean TTE, *AFLGopher* delivers substantial speedups over all baselines: 5.60× over AFLGo, 5.20× over BEACON, 4.98× over WindRanger, 4.52× over SelectFuzz, and 5.07× over AFLGo-ICall. All differences are statistically significant ($p < 0.05$), and the corresponding mean $\hat{A}_{12}$ effect sizes are 0.76, 0.83, 0.89, 0.80, and 0.81, respectively—indicating large performance improvements [31].

Although DeepGo is not publicly available for direct evaluation, we estimate relative performance based on its published TTE results. Using AFLGo, BEACON, and WindRanger as normalization anchors, we estimate that *AFLGopher* achieves 2.14×, 1.57×, and 2.05× speedups over DeepGo, respectively. These indirect comparisons further suggest that *AFLGopher* advances the state of the art in vulnerability exposure.

Conclusion: *AFLGopher* consistently exposes known vulnerabilities faster than all evaluated baselines, and shows promising improvements over reported state-of-the-art systems, validating its effectiveness in vulnerability-triggering scenarios under realistic fuzzing constraints.

6.4 Effectiveness and Accuracy of Conditional Statement Classification (CSC)

To assess the contribution of our conditional statement classification (CSC) module, we evaluate its impact on both Time-to-Target (TTT) and Time-to-Exposure (TTE) across two benchmark suites: UniBench and the AFLGo Test Suite (ATS). For this purpose, we isolate the CSC component in *AFLGopher*, referred to as *AFLGopher*-C, and compare its performance against all baseline fuzzers and the full version of *AFLGopher*.

As shown in Table 2 and Table 1, *AFLGopher*-C achieves consistent improvements over AFLGo and other baselines, though it does not match the full capabilities of *AFLGopher*. On average, *AFLGopher*-C achieves speedups of 1.52×, 1.05×, 1.34×, 1.03×, and 1.16× over AFLGo, BEACON, WindRanger, SelectFuzz, and AFLGo-ICall on TTT. On TTE, *AFLGopher*-C delivers even more significant gains—1.64×, 1.53×, 1.46×, 1.33×, and 1.49× against the same fuzzers, respectively. These results demonstrate that CSC alone contributes meaningfully to the overall efficiency of *AFLGopher*.

Clustering Accuracy. To validate the accuracy of CSC, we perform both internal and external evaluations.

(1) Internal validation. We manually labeled 1,000 conditional statements drawn from our benchmark dataset (see subsection 2.3 and subsubsection 6.1.2) and use the Adjusted Rand Index (ARI) [27] to compare our semantic clusters against ground truth labels. Our clustering yields an ARI score of 0.91, indicating near-perfect alignment. In contrast, random clusterings on the same dataset average an ARI of 0.29, underscoring the effectiveness of our clustering method.

(2) External validation. To further assess clustering quality, we use the Silhouette Coefficient (SC) [25], a standard unsupervised clustering metric that evaluates cohesion and separation. SC values range from -1 to 1, where values close to 1 indicate well-separated clusters, values near 0 suggest overlapping boundaries, and negative values indicate likely misclassifications. Across all semantic clusters generated by CSC, we observe an average SC of 0.82, with

all cluster values exceeding 0.8—strong evidence that our clusters are both compact and well-separated.

Together, these quantitative results confirm that our conditional statement classification is not only semantically meaningful but also empirically precise, contributing directly to *AFLGopher*'s performance advantage.

6.5 Effectiveness and Accuracy of Next Function Prediction (NFP)

To assess the contribution of the Next Function Prediction (NFP) component, we evaluate its impact on both Time-to-Target (TTT) and Time-to-Exposure (TTE). For this analysis, we isolate NFP from *AFLGopher* —referred to as *AFLGopher*-N—and compare its performance to all baseline fuzzers as well as the full version of *AFLGopher*.

As shown in Table 2, *AFLGopher*-N achieves speedups of 1.40×, 0.96×, 1.22×, 0.94×, and 1.07× over AFLGo, BEACON, WindRanger, SelectFuzz, and AFLGo-ICall, respectively, in terms of average TTT. Similarly, in Table 1, *AFLGopher*-N achieves TTE speedups of 1.13×, 1.05×, 1.01×, 0.91×, and 1.03× against the same set of fuzzers.

While NFP yields improvements in several scenarios, its standalone impact is less pronounced than that of CSC. This is largely attributable to the distribution of indirect calls across the evaluated programs. For example, LIBMING contains over 300 indirect call sites, providing ample opportunity for NFP to optimize guidance. In such programs, NFP significantly enhances the feasibility estimation of indirect-call edges. However, in programs with relatively few indirect calls, the contribution of NFP is minimal. In contrast, conditional statements are pervasive in all programs, making CSC generally more influential when used in isolation.

These observations underscore that NFP and CSC are complementary: each addresses a different dimension of path feasibility. Their combined use is critical to *AFLGopher* 's overall effectiveness.

Prediction Accuracy. To evaluate the accuracy of the NFP model, we define precision using the following formula:

$$\text{Accuracy} = 1 - \frac{1}{n}\sum_{i=1}^{n}\left|F_{p_i} - F_{t_i}\right|$$

where n is the total number of indirect-call edges, F_{p_i} is the predicted feasibility score, and F_{t_i} is the ground truth feasibility from execution traces. Using this metric, NFP achieves a high average prediction accuracy of 95.11%, indicating that the model is able to estimate feasibility with strong alignment to observed behavior.

6.6 Effectiveness and Accuracy of Runtime Update Process

To assess the contribution of the runtime update process, we evaluate its impact on both Time-to-Target (TTT) and Time-to-Exposure (TTE). For this analysis, we isolate this process from *AFLGopher* —referred to as *AFLGopher*-U—and compare its performance to all baseline fuzzers as well as the full version of *AFLGopher*.

As shown in Table 2, *AFLGopher*-U achieves speedups of 1.68×, 1.15×, 1.48×, 1.14×, and 1.29× over AFLGo, BEACON, WindRanger, SelectFuzz, and AFLGo-ICall, respectively, in terms of average TTT. Similarly, in Table 1, *AFLGopher*-N achieves TTE speedups of 2.38×, 2.20×, 2.11×, 1.91×, and 2.14× against the same set of fuzzers.

These results corroborate that combining CSC and NFP outperforms either component alone. Although *AFLGopher*-U improves over existing baselines, it still underperforms the full *AFLGopher* because, without the update process, it cannot incorporate feasibility information from new traces during fuzzing.

Parameter Sensitivity: Thresholds for Triggering Updates. The error thresholds θ_{CSC} and θ_G serve as key hyperparameters for *AFLGopher* 's runtime update mechanism, determining when to retrain the prediction model and refresh feasibility-aware distances. To identify effective values, we apply a control variate methodology based on average Time-to-Target (TTT), varying one parameter while keeping the other fixed to isolate its individual impact.

We evaluate θ_{CSC} over the range [0%, 20%] in 2.5% increments, and θ_G over [0%, 10%] in 0.5% increments. Each configuration is tested using the UniBench benchmark suite, where we measure the average Time-to-Target (TTT) across all target programs. As shown in Figure 5, the configuration $\theta_{CSC} = 10\%$ and $\theta_G = 3\%$ consistently achieves the best trade-off — minimizing unnecessary updates while remaining responsive to prediction drift. This setting yields the lowest average TTT, balancing guidance quality with computational efficiency.

Thresholds below this range lead to overly frequent updates, which increase computational cost and reduce effective fuzzing throughput. Conversely, higher thresholds delay necessary updates, causing stale feasibility scores that misguide the fuzzer. While the defaults provide a robust general-purpose configuration, *AFLGopher* also exposes both parameters for domain-specific tuning, allowing users to optimize the trade-off between accuracy and resource efficiency based on workload characteristics.

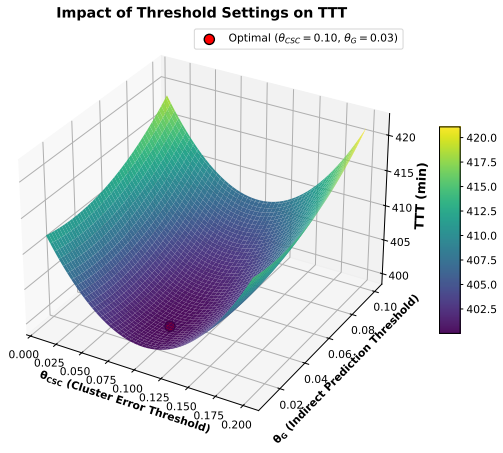

Figure 5: The impact of error threshold settings on runtime update process.

7 Limitations

While *AFLGopher* demonstrates substantial improvements over prior directed fuzzing techniques, it also has several limitations that merit further exploration. We outline three primary limitations observed during our evaluation.

Runtime Overhead for Feasibility Assignment. AFLGopher introduces a modest runtime overhead during the initial phase of fuzzing, stemming from the need to collect execution traces and compute feasibility-aware distance metrics. This initialization phase typically lasts approximately five minutes before feasibility-informed guidance becomes effective. While this overhead is negligible for long-running fuzzing campaigns, it may hinder responsiveness in time-sensitive settings or programs with short execution windows. For instance, in the UniBench targets CFLOW:PARSER.C:1223 and CFLOW:PARSER.C:108, as well as CVE-2016-4487 from the ATS suite, we observed a measurable delay in reaching the target due to this "warm-up" latency. Importantly, this initialization cost is incurred post-fuzzing launch and is fully accounted for in the evaluation metrics, ensuring that all reported speedups reflect end-to-end performance. Despite this delay, AFLGopher consistently outperforms state-of-the-art baselines across the majority of benchmarks. Nonetheless, optimizing early trace collection and reducing cold-start latency remain promising directions for future work, particularly to enhance applicability in real-time fuzzing pipelines or continuous integration environments.

Resource Cost of Runtime Updates. To maintain adaptive guidance, *AFLGopher* periodically evaluates prediction accuracy and updates feasibility models. Although these updates are conducted asynchronously, they still consume memory and CPU resources. In environments with constrained resources or high mutation throughput, this can lead to slight performance degradation. For example, in case MP3GAIN:LAYER3.C:1116 from Table 2, *AFLGopher* underperforms compared to SelectFuzz, which aggressively explores all reachable paths without feasibility modeling. While *AFLGopher* 's several updates in this case enhanced model precision, they also incurred non-trivial overhead. Currently, update frequency is controlled via error thresholds (see subsection 6.6); more intelligent scheduling strategies or lightweight approximations may offer improvements in future versions.

Summary. Despite these limitations, *AFLGopher* consistently delivers superior performance across a wide range of targets. Moreover, each limitation presents a clear path toward further enhancement—through trace optimization, feasibility-exploitability fusion, or smarter update scheduling—reinforcing the system's extensibility and relevance to future work in directed fuzzing.

8 Related Work

Directed Grey-Box Fuzzing. Directed grey-box fuzzing (DGF) has gained increasing traction for its ability to efficiently test specific program regions, particularly those associated with vulnerabilities or patches. AFLGo [2] introduced the foundational concept of DGF by using control-flow distances to prioritize seeds closer to target locations. Hawkeye [3] builds upon this approach by improving seed selection and incorporating indirect-call targets, resulting in more complete path coverage.

Several other systems have proposed alternative methods to direct fuzzing toward program hotspots. Liang et al. [8] presented a lightweight sequence-coverage–based technique for user-defined statement targeting. Wüstholz et al. [37] employed online static analysis to refine directionality. UAFuzz [21] focuses on use-after-free vulnerabilities at the binary level, becoming the first directed fuzzer specialized for such a class.

Recent state-of-the-art tools include WindRanger [7], which uses deviation blocks to guide distance computation; Beacon [11], which performs lightweight static analysis for path pruning; and SelectFuzz [19], which selectively instruments paths most relevant to the target site. Other noteworthy systems like DeepGo [15], HyperGo [14], and DAFL [12] also contribute meaningful advances, particularly in enhancing guidance through learned metrics. However, DeepGo and HyperGo are unavailable for reproduction, and DAFL's integration presented practical challenges. As a result, we evaluate against WindRanger, Beacon, and SelectFuzz as representative, open-source baselines.

To the best of our knowledge, none of these systems incorporates feasibility-aware guidance. In contrast, *AFLGopher* is the first to leverage both branch and indirect-call feasibility—learned dynamically and semantically grouped—to improve the precision of directed fuzzing.

Machine Learning–Assisted Fuzzing. Recent years have also seen a rise in machine learning (ML) techniques applied to various aspects of fuzzing. Fuzzguard [43] uses deep learning to predict input reachability, filtering out low-value seeds before execution. DeFuzz [42] identifies vulnerable code regions via pre-trained models and directs fuzzing accordingly. DeepFuzz [16] employs a sequence-to-sequence generator to create well-formed C programs for compiler fuzzing.

Neuzz [26] applies surrogate neural models to smooth program branches and facilitate gradient-guided input generation. Angora [4] similarly relies on gradient descent to solve path constraints, improving branch coverage. Suzzer [39] prioritizes paths statistically more likely to contain bugs. Meanwhile, RL-based approaches such as Wang et al. [33] introduce hierarchical schedulers for coverage-maximizing seed selection.

These systems predominantly focus on improving input mutation strategies or identifying coverage-critical paths. In contrast, *AFLGopher* contributes a novel dimension to ML-assisted fuzzing by: (1) clustering conditional statements via unsupervised learning to support generalizable feasibility estimation, and (2) using an LSTM-based model to predict the runtime reachability of indirect-call targets. Rather than enhancing input generation, *AFLGopher* enhances guidance precision, thereby improving the fuzzing process at the semantic and structural levels.

9 Conclusion

Directed grey-box fuzzing often struggles with inefficiencies due to feasibility-unaware distance calculations. In this paper, we introduce a feasibility-aware approach with *AFLGopher*, a tool that predicts the feasibility of branch statements and incorporates this feedback into the fuzzing. *AFLGopher* combines branch-statement classification with an efficient model update mechanism to ensure precision and efficiency. In performance evaluations using benchmark datasets, *AFLGopher* significantly outperformed state-of-the-art directed fuzzers in both Time-To-Target and Time-To-Exposure metrics. These results confirm the superior effectiveness of *AFLGopher* over contemporary directed fuzzers.

References

[1] Rohan Bavishi, Hiroaki Yoshida, and Mukul R Prasad. 2019. Phoenix: Automated data-driven synthesis of repairs for static analysis violations. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 613–624.

[2] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. 2017. Directed greybox fuzzing. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*. 2329–2344.

[3] Hongxu Chen, Yinxing Xue, Yuekang Li, Bihuan Chen, Xiaofei Xie, Xiuheng Wu, and Yang Liu. 2018. Hawkeye: Towards a desired directed grey-box fuzzer. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2095–2108.

[4] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725.

[5] Ronán Conroy. 2015. Sample size: A rough guide. *Retreived from http://www.beaumontethics. ie/docs/application/samplesizecalculation. pdf* (2015).

[6] Xiaoning Du, Bihuan Chen, Yuekang Li, Jianmin Guo, Yaqin Zhou, Yang Liu, and Yu Jiang. 2019. Leopard: Identifying vulnerable code for vulnerability assessment through program metrics. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 60–71.

[7] Du, Zhengjie and Li, Yuekang and Liu, Yang and Mao, Bing. 2022. WindRanger: a directed greybox fuzzer driven by deviation basic blocks. In *Proceedings of the 44th International Conference on Software Engineering*. 2440–2451.

[8] M Eddington. 2008. Peach Fuzzer. URL:https://peachtech.gitlab.io/peach-fuzzer-community/(visitedon04/10/2021)

[9] Lirong Fu, Shouling Ji, Kangjie Lu, Peiyu Liu, Xuhong Zhang, Yuxuan Duan, Zihui Zhang, Wenzhi Chen, and Yanjun Wu. 2021. Cpscan: Detecting bugs caused by code pruning in iot kernels. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 794–810.

[10] David Gens, Simon Schmitt, Lucas Davi, and Ahmad-Reza Sadeghi. 2018. K-Miner: Uncovering Memory Corruption in Linux.. In *NDSS*.

[11] Heqing Huang, Yiyuan Guo, Qingkai Shi, Peisen Yao, Rongxin Wu, and Charles Zhang. 2022. Beacon: Directed grey-box fuzzing with provable path pruning. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 36–50.

[12] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. {DAFL}: Directed Grey-box Fuzzing guided by Data Dependency. In *32nd USENIX Security Symposium (USENIX Security 23)*. 4931–4948.

[13] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, et al. 2021. {UNIFUZZ}: A holistic and pragmatic {Metrics-Driven} platform for evaluating fuzzers. In *30th USENIX Security Symposium (USENIX Security 21)*. 2777–2794.

[14] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Kai Lu, and Gen Zhang. 2023. HyperGo: Probability-based Directed Hybrid Fuzzing. *arXiv preprint arXiv:2307.07815* (2023).

[15] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. 2024. DeepGo: Predictive Directed Greybox Fuzzing. In *2024 Network and Distributed System Security Symposium (NDSS 2024)*.

[16] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. 2019. Deepfuzz: Automatic generation of syntax valid c programs for fuzz testing. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 1044–1051.

[17] Kangjie Lu and Hong Hu. 2019. Where does it go? refining indirect-call targets with multi-layer type analysis. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1867–1881.

[18] Kangjie Lu, Aditya Pakki, and Qiushi Wu. 2019. Detecting {Missing-Check} bugs via semantic-and {Context-Aware} criticalness and constraints inferences. In *28th USENIX Security Symposium (USENIX Security 19)*. 1769–1786.

[19] Changhua Luo, Wei Meng, and Penghui Li. 2023. SELECTFUZZ: Efficient Directed Fuzzing with Selective Path Exploration. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2693–2707.

[20] Lannan Luo, Jun Ming, Dinghao Wu, Peng Liu, and Shengjian Zhu. 2019. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. *IEEE Transactions on Dependable and Secure Computing* 16, 3 (2019), 473–487. doi:10.1109/TDSC.2017.2754490

[21] Manh-Dung Nguyen, Sébastien Bardin, Richard Bonichon, Roland Groz, and Matthieu Lemerre. 2020. Binary-level Directed Fuzzing for {Use-After-Free} Vulnerabilities. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*. 47–62.

[22] Sebastian Österlund, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2020. {ParmeSan}: Sanitizer-guided greybox fuzzing. In *29th USENIX Security Symposium (USENIX Security 20)*. 2289–2306.

[23] Kexin Pei, Zhiqiang Xuan, Junfeng Yang, and Suman Jana. 2022. Learning approximate execution semantics from traces for binary function similarity. *IEEE Transactions on Dependable and Secure Computing* (2022). doi:10.1109/TDSC.2022.3195252

[24] Van-Thuan Pham, Manh-Dung Nguyen, Quang-Trung Ta, Toby Murray, and Benjamin IP Rubinstein. 2021. Towards systematic and dynamic task allocation for collaborative parallel fuzzing. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1337–1341.

[25] Peter J Rousseeuw. 1987. Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics* 20 (1987), 53–65.

[26] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2019. Neuzz: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 803–817.

[27] Douglas Steinley. 2004. Properties of the hubert-arable adjusted rand index. *Psychological methods* 9, 3 (2004), 386.

[28] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. Healer: Relation learning guided kernel fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*. 344–358.

[29] Robert Swiecki. 2017. Honggfuzz: A general-purpose, easy-to-use fuzzer with interesting analysis options. *URl: https://github. com/google/honggfuzz (visited on 06/21/2017)* (2017).

[30] Michele Tufano, Martin White, Martin Martinez, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *Proceedings of the 26th IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 479–490. doi:10.1109/SANER.2019.8668043

[31] András Vargha and Harold D Delaney. 2000. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics* 25, 2 (2000), 101–132.

[32] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Harald C Gall, and Andy Zaidman. 2020. How developers engage with static analysis tools in different contexts. *Empirical Software Engineering* 25 (2020), 1419–1457.

[33] Jinghan Wang, Chengyu Song, and Heng Yin. 2021. Reinforcement Learning-based Hierarchical Seed Scheduling for Greybox Fuzzing. In *2021 Network and Distributed System Security Symposium*.

[34] Pengfei Wang, Xu Zhou, Kai Lu, Tai Yue, and Yingying Liu. 2020. Sok: The progress, challenges, and perspectives of directed greybox fuzzing. *Challenges, and Perspectives of Directed Greybox Fuzzing* (2020).

[35] Ronald L Wasserstein and Nicole A Lazar. 2016. The ASA statement on p-values: context, process, and purpose. 129–133 pages.

[36] Qiushi Wu, Zhongshu Gu, Hani Jamjoom, and Kangjie Lu. 2024. Gnnic: Finding long-lost sibling functions with abstract similarity. (2024).

[37] Valentin Wüstholz and Maria Christakis. 2020. Targeted greybox fuzzing with static lookahead analysis. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 789–800.

[38] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. 2022. Exploit the last straw that breaks android systems. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2230–2247.

[39] Yuyue Zhao, Yangyang Li, Tengfei Yang, and Haiyong Xie. 2020. Suzzer: A vulnerability-guided fuzzer based on deep learning. In *International Conference on Information Security and Cryptology*. Springer, 134–153.

[40] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. 2023. Callee: Recovering call graphs for binaries with transfer and contrastive learning. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2357–2374.

[41] Xiaogang Zhu and Marcel Böhme. 2021. Regression greybox fuzzing. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 2169–2182.

[42] Xiaogang Zhu, Shigang Liu, Xian Li, Sheng Wen, Jun Zhang, Camtepe Seyit, and Yang Xiang. 2020. Defuzz: Deep learning guided directed fuzzing. *arXiv preprint arXiv:2010.12149* (2020).

[43] Peiyuan Zong, Tao Lv, Dawei Wang, Zizhuang Deng, Ruigang Liang, and Kai Chen. 2020. {FuzzGuard}: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In *29th USENIX security symposium (USENIX security 20)*. 2255–2269.

Table 2: The Time-to-Target Results on Programs from UniBench

ID	Program	Version	Target Position	AFLGo	BEACON	WindRanger	SelectFuzz	AFLGo-ICall	AFLGopher-C	AFLGopher-N	AFLGopher-U	AFLGopher
1	cflow	1.6	parser.c:281	*T.O.*	3.08h	2.56h	2.33h	*T.O.*	*T.O.*	17.17h	6.22h	2.44h
2			c.c:1783	1.10h	0.15h	0.11h	0.11h	0.30h	0.48h	0.64h	0.55h	0.20h
3			parser.c:1223	0.12h	0.07h	0.11h	2.76h	0.13h	0.40h	0.44h	0.43h	0.16h
4			parser.c:108	1.30h	2.70h	0.43h	0.45h	1.20h	0.92h	1.27h	0.87h	0.34h
5	mp42aac	Bento4 1.5.1-628	Ap4AvccAtom.cpp:82	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	8.47h	10.10h	8.15h	3.49h
6			Ap4TrunAtom.cpp:139	*T.O.*	N/A	*T.O.*	9.22h	*T.O.*	17.28h	20.72h	18.71h	6.95h
7			Ap4SbgpAtom.cpp:181	*T.O.*	N/A	*T.O.*	0.80h	*T.O.*	21.21h	*T.O.*	21.49h	8.74h
8			Ap4AtomFactory.cpp:490	*T.O.*	N/A	*T.O.*	12.57h	*T.O.*	11.05h	12.72h	9.53h	4.31h
9	jhead	3.0.0	exif.c:1339	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	22.08h	22.93h	18.43h	7.95h
10			iptc.c:143	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	21.44h	23.65h	22.67h	7.88h
11			iptc.c:91	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	5.31h	*T.O.*	5.26h	2.18h
12			makernote.c:174	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	16.95h	19.97h	18.98h	6.78h
13	mp3gain	1.5.2	layer3.c:1116	23.80h	N/A	*T.O.*	6.76h	*T.O.*	20.73h	23.23h	17.13h	7.74h
14			mp3gain.c:602	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	5.52h	7.19h	5.03h	2.19h
15			interface.c:663	*T.O.*	N/A	*T.O.*	2.52h	*T.O.*	7.38h	*T.O.*	6.93h	2.72h
16			apetag.c:341	21.55h	N/A	8.10h	*T.O.*	17.63h	*T.O.*	*T.O.*	*T.O.*	14.66h
17	lame	3.99.5	bitstream.c:823	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	15.25h	18.61h	14.55h	6.09h
18			lame.c:2148	*T.O.*	N/A	*T.O.*	11.89h	18.73h	*T.O.*	*T.O.*	*T.O.*	11.13h
19			uantize_pvt.c:441	*T.O.*	N/A	14.15h	*T.O.*	*T.O.*	15.47h	17.81h	16.74h	6.20h
20			get_audio.c:1605	*T.O.*	N/A	*T.O.*	9.09h	*T.O.*	22.96h	*T.O.*	*T.O.*	9.21h
21	imginfo	jasper 2.0.1	jp2_cod.c:841	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	22.28h	10.10h
22			jp2_cod.c:636	*T.O.*	N/A	*T.O.*	2.97h	14.71h	18.08h	19.64h	15.29h	6.87h
23			jas_stream.c:823	*T.O.*	N/A	16.40h	*T.O.*	*T.O.*	12.34h	14.91h	12.38h	4.86h
24			jpc_dec.c:1393	*T.O.*	N/A	*T.O.*	14.50h	*T.O.*	12.28h	15.95h	12.63h	5.06h
25	gdk-pixbuf-pixdata	gdk-pixbuf 2.31.1	gdk-pixbuf-loader.c:387	*T.O.*	N/A	N/A	N/A	*T.O.*	10.18h	11.11h	9.07h	3.96h
26			io-qtif.c:511	*T.O.*	N/A	N/A	N/A	14.72h	5.10h	6.73h	4.97h	2.10h
27			io-jpeg.c:691	*T.O.*	N/A	N/A	N/A	*T.O.*	7.33h	9.40h	8.60h	2.95h
28			io-tga.c:360	21.29h	N/A	N/A	N/A	20.06h	19.89h	18.42h	0.96h	0.40h
29	jq	1.5	jv_dtoa.c:3122	*T.O.*	N/A	N/A	N/A	4.44h	17.85h	23.65h	18.19h	7.36h
30			jv_dtoa.c:2004	*T.O.*	N/A	N/A	N/A	*T.O.*	23.53h	*T.O.*	22.45h	8.75h
31			jv_dtoa.c:2518	*T.O.*	N/A	N/A	N/A	*T.O.*	11.86h	12.92h	11.64h	4.52h
32			jv_unicode.c:42	*T.O.*	N/A	N/A	N/A	4.60h	16.89h	19.61h	16.55h	6.73h
33	tcpdump	4.8.1	print-aodv.c:259	*T.O.*	N/A	16.05h	N/A	*T.O.*	12.69h	15.82h	11.48h	4.88h
34			print-ntp.c:412	*T.O.*	N/A	12.23h	N/A	*T.O.*	7.83h	8.48h	7.85h	2.89h
35			print-rsvp.c:1252	*T.O.*	N/A	14.07h	N/A	*T.O.*	5.90h	*T.O.*	5.15h	2.32h
36			print-l2tp.c:606	*T.O.*	N/A	*T.O.*	N/A	4.68h	11.50h	13.63h	12.44h	4.43h
37	tic	ncurses 6.1	captoinfo.c:189	*T.O.*	N/A	N/A	N/A	4.65h	7.13h	7.72h	5.89h	2.67h
38			alloc entry.c:141	*T.O.*	N/A	N/A	N/A	*T.O.*	14.42h	17.22h	12.51h	5.41h
39			name match.c:111	*T.O.*	N/A	N/A	N/A	*T.O.*	3.78h	4.61h	3.72h	1.45h
40			entries.c:78	*T.O.*	N/A	N/A	N/A	*T.O.*	9.81h	11.28h	8.90h	3.75h
41	flvmeta	1.2.1	json.c:1036	*T.O.*	N/A	*T.O.*	*T.O.*	4.45h	9.39h	12.07h	11.34h	3.85h
42			api.c:718	*T.O.*	N/A	*T.O.*	*T.O.*	4.41h	16.66h	21.72h	18.26h	6.69h
43			flvmeta.c:1023	*T.O.*	N/A	*T.O.*	5.67h	*T.O.*	*T.O.*	*T.O.*	23.29h	8.36h
44			check.c:769	*T.O.*	N/A	*T.O.*	*T.O.*	4.76h	9.39h	12.44h	8.99h	3.88h
45	tiffsplit	libtiff 3.9.7	tif_ojpeg.c:1277	*T.O.*	N/A	*T.O.*	2.54h	*T.O.*	7.33h	8.76h	6.74h	2.95h
46			tif_read.c:335	*T.O.*	N/A	*T.O.*	1.58h	*T.O.*	11.80h	13.56h	9.65h	4.38h
47			tif_jbig.c:277	*T.O.*	N/A	*T.O.*	11.70h	*T.O.*	*T.O.*	*T.O.*	*T.O.*	11.87h
48			tif_dirread.c:1977	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	13.62h
49	nm	binutils-5279478	tekhex.c:325	*T.O.*	23.46h	*T.O.*	*T.O.*	*T.O.*	7.72h	8.51h	6.87h	2.98h
50			elf.c:8793	*T.O.*	23.67h	*T.O.*	10.75h	4.63h	18.56h	23.27h	19.10h	7.29h
51			dwarf2.c:2378	*T.O.*	18.51h	*T.O.*	11.68h	*T.O.*	10.35h	12.14h	9.35h	3.76h
52			elf-properties.c:51	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	10.19h
53	pdftotext	4.00	XRef.cc:645	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	12.70h
54			GfxFont.cc:1337	*T.O.*	N/A	*T.O.*	*T.O.*	4.76h	12.95h	17.19h	11.84h	5.34h
55			Stream.cc:1004	*T.O.*	N/A	*T.O.*	9.54h	4.46h	*T.O.*	*T.O.*	54.03h	*T.O.*
56			GfxFont.cc:1643	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	*T.O.*	22.30h	*T.O.*	12.98h
57	sqlite3	SQLite 3.8.9	pager.c:5017	*T.O.*	N/A	N/A	*T.O.*	*T.O.*	8.07h	9.55h	6.93h	3.14h
58			func.c:1029	*T.O.*	N/A	N/A	*T.O.*	4.48h	11.03h	12.41h	10.35h	4.26h
59			insert.c:1498	*T.O.*	N/A	N/A	*T.O.*	4.46h	20.22h	22.47h	22.71h	7.98h
60			vdbe.c:1984	*T.O.*	16.40h	N/A	9.74h	*T.O.*	12.88h	14.06h	12.68h	4.65h
61	exiv2	0.26	tiffcomposite.cpp:82	*T.O.*	N/A	7.12h	7.35h	3.20h	7.62h	9.30h	6.75h	2.88h
62			XMPMeta-Parse.cpp:847	23.08h	N/A	2.45h	*T.O.*	4.56h	18.00h	21.47h	0.89h	0.32h
63			tiffvisitor.cpp:1044	*T.O.*	N/A	*T.O.*	11.53h	3.37h	*T.O.*	4.93h	4.37h	1.60h
64			XMPMeta-Parse.cpp:896	*T.O.*	N/A	21.47h	11.68h	3.56h	2.35h	2.66h	2.40h	0.94h
65	objdump	binutils-2.28	elf.c:9509	*T.O.*	20.92h	*T.O.*	8.08h	*T.O.*	16.28h	20.30h	14.72h	6.52h
66			section.c:936	*T.O.*	*T.O.*	*T.O.*	0.87h	*T.O.*	12.80h	14.61h	14.15h	4.72h
67			bfd.c:1108	*T.O.*	23.48h	*T.O.*	*T.O.*	*T.O.*	7.59h	8.42h	6.86h	2.81h
68			bfdio.c:262	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*	12.10h	15.88h	12.46h	4.95h
69	ffmpeg	4.0.1	rawdec.c:268	*T.O.*	N/A	*T.O.*	5.06h	4.51h	14.49h	17.18h	14.06h	5.99h
70			decode.c:557	*T.O.*	N/A	*T.O.*	13.66h	4.58h	*T.O.*	17.91h	3.52h	1.29h
71			dump.c:632	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	9.32h	11.30h	10.51h	3.72h
72			utils.c	*T.O.*	N/A	*T.O.*	11.46h	4.46h	*T.O.*	*T.O.*	*T.O.*	9.30h
73	mujs	1.0.2	jsrun.c:572	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	*T.O.*	19.85h	18.39h	6.58h
74			jsgc.c:47	*T.O.*	N/A	*T.O.*	*T.O.*	4.57h	*T.O.*	*T.O.*	*T.O.*	*T.O.*
75			jsdump.c:292	*T.O.*	N/A	10.67h	*T.O.*	*T.O.*	*T.O.*	*T.O.*	21.60h	9.78h
76			jsvalue.c:362	*T.O.*	N/A	*T.O.*	*T.O.*	*T.O.*	22.22h	3.62h	2.57h	1.16h
77	swftools	0.9.2	initcode.c:242	*T.O.*	N/A	18.53h	12.22h	*T.O.*	23.46h	*T.O.*	*T.O.*	11.55h
78			png.c:410	*T.O.*	N/A	11.31h	13.14h	*T.O.*	14.75h	8.29h	6.47h	2.60h
79			poly.c:137	*T.O.*	N/A	*T.O.*	3.21h	*T.O.*	*T.O.*	*T.O.*	*T.O.*	*T.O.*
80			jpeg2swf.c:257	*T.O.*	N/A	9.19h	8.32h	4.65h	20.43h	*T.O.*	19.27h	7.66h
		speedup		**3.76x**	**2.57x**	**3.30x**	**2.52x**	**2.86x**	**2.47x**	**2.69x**	**2.23x**	-
		mean *p-values*		**0.009**	**0.004**	**0.017**	**0.005**	**0.012**	**0.03**	**0.008**	**0.014**	-
		mean $\hat{A}_{12}$		**0.78**	**0.81**	**0.86**	**0.82**	**0.79**	**0.79**	**0.83**	**0.80**	-