RESEARCH-ARTICLE

Not All Paths Are Equal: Multi-path Optimization for Directed Hybrid Fuzzing

PEIHONG LIN, National University of Defense Technology China, Changsha, Hunan, China

PENGFEI WANG, National University of Defense Technology China, Changsha, Hunan, China

XU ZHOU, National University of Defense Technology China, Changsha, Hunan, China

WEI XIE, National University of Defense Technology China, Changsha, Hunan, China

GEN ZHANG, National University of Defense Technology China, Changsha, Hunan, China

KAI LU, National University of Defense Technology China, Changsha, Hunan, China

Accepted: 06 May 2025
Revised: 22 April 2025
Received: 21 October 2024

Not All Paths Are Equal: Multi-path Optimization for Directed Hybrid Fuzzing

PEIHONG LIN, National University of Defense Technology, China
PENGFEI WANG*, National University of Defense Technology, China
XU ZHOU, National University of Defense Technology, China
WEI XIE, National University of Defense Technology, China
GEN ZHANG, National University of Defense Technology, China
KAI LU, National University of Defense Technology, China

Directed grey-box fuzzing (DGF) can improve bug exposure efficiency by stressing bug-prone areas. Recent studies have modeled DGF as the problem of finding and optimizing paths to reach target sites. However, they still face the "*multi-path*" challenge. When a target site is reachable by multiple paths, it is crucial to comprehensively evaluate and effectively select these paths, as this affects the fuzzer's choice between reaching target sites via optimal paths and enhancing path diversity toward targets to expose hidden bugs in non-optimal paths. In this paper, we propose MultiGo, a directed hybrid fuzzer designed for multi-path optimization. First, we propose a new fitness metric called *path difficulty* to comprehensively evaluate the promising paths. This metric uses the Poisson distribution to estimate the probability of exploring basic blocks along execution paths based on statistical block frequency, distinguishing between optimal and challenging paths. With path difficulty as a key factor, a customized *Contextual Multi-Armed Bandit* (CMAB) model is employed to efficiently optimize path scheduling by comprehensively considering the impact of testing conditions on path scheduling. We introduce the concept of the *fuzzing context* to represent and evaluate testing conditions, which encompass factors such as path characteristics (e.g., path difficulty), the testing agent (e.g., fuzzing or symbolic execution), and the testing goal (e.g., path exploitation or exploration). Then, the CMAB model predicts the expected rewards for scheduling paths under different testing agents and goals, thereby optimizing path scheduling. By leveraging the CMAB model, MultiGo enhances DGF's capability to explore easier paths and symbolic execution's capacity to handle more complex ones, enabling efficient target reaching through optimal paths while ensuring sufficient coverage of non-optimal paths. MultiGo is evaluated on 136 target sites of 41 real-world programs from 3 benchmarks. The experimental results show that MultiGo outperforms the state-of-the-art directed fuzzers (AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL) and hybrid fuzzers (SymCC and SymGo) in reaching target sites and exposing known vulnerabilities. Moreover, MultiGo also discovered 14 undisclosed vulnerabilities.

CCS Concepts: • **Software and its engineering → Software testing and debugging**.

*Corresponding author: pfwang@nudt.edu.cn.
Authors' Contact Information: Peihong Lin, National University of Defense Technology, Changsha, China, phlin22@nudt.edu.cn; Pengfei Wang*, National University of Defense Technology, Changsha, China, pfwang@nudt.edu.cn; Xu Zhou, National University of Defense Technology, Changsha, China, zhouxu@nudt.edu.cn; Wei Xie, National University of Defense Technology, Changsha, China, xiewei@nudt.edu.cn; Gen Zhang, National University of Defense Technology, Changsha, China, zhanggen@nudt.edu.cn; Kai Lu, National University of Defense Technology, Changsha, China, kailu@nudt.edu.cn.

ACM 1557-7392/2025/5-ART
https://doi.org/10.1145/3735555

1 INTRODUCTION

Directed grey-box fuzzing [5, 7, 14] has gained significant attention in software security due to its efficiency in testing vulnerable code areas. Most directed grey-box fuzzers (e.g., AFLGo [4]) leverage lightweight compile-time instrumentation to drive the fuzzing toward pre-defined target sites. Based on the call graph and control-flow graph information of the program under test (PUT), DGF calculates the distance from basic blocks to target basic blocks (i.e., basic-block-level distance, BB distance for short) and uses this BB distance to compute the distance between inputs and targets (i.e., seed distance). The seed distance serves as the fitness metric for seed selection and energy assignment. Thus, DGF can prioritize the seeds that are more likely to reach the target sites (i.e., optimal seeds), which makes DGF outperform CGF in specific scenarios, such as patch testing [33], bug reproduction [22, 31], and potential buggy code verification [28, 32].

Many DGF techniques have been proposed to accelerate the process of reaching targets. For example, recent works [5, 19, 20, 35] have modeled the process of DGF as finding paths to reach the target sites. Some prior works have improved the efficiency by refining the path reachability [19, 20, 35], while others [8, 19, 23] leverage symbolic execution to overcome complex constraints along paths toward targets, thereby accelerating target reaching and forming directed hybrid fuzzing (DHF). However, researchers still face the "***multi-path***" issue in directed fuzzing. In directed fuzzing, a target is often reachable by more than one path. Our observation shows that 94% (128/136) of the target sites we tested (§5.5) are reachable via multiple paths, with some targets having up to 16 different paths. While the primary goal of directed fuzzing is to quickly reach target sites to discover potential functional errors (e.g., overflow vulnerabilities), exploring alternative paths is also crucial, as certain non-functional errors require specific execution paths to be triggered. Therefore, for this "multi-path" issue, evaluating and handling multiple paths that are promising for reaching the targets (i.e., target-reachable paths) poses significant challenges.

Challenge 1: How to accurately evaluate and distinguish different paths in the multi-path scenarios? Existing directed fuzzing works typically rely on static analysis or fuzzing runtime information to design fitness metrics, which are then used to evaluate path reachability and distinguish paths. Static approaches, such as AFLGo [5], Hawkeye [7], DAFL [12], SelectFuzz [21], and SDFuzz [15], predominantly utilize BB distance, combined with other metrics [7, 9, 14], to evaluate the reachability of different basic blocks. These methods then design the fitness metrics based on the block reachability to distinguish paths during fuzzing. However, current static approaches struggle to accurately analyze the complexity of path constraints (§2.2). As a result, fitness metrics derived from current static methods fail to comprehensively assess the difficulty of reaching certain basic blocks and correctly distinguish paths in the multi-path scenarios. On the other hand, dynamic approaches, including Macro [10], SAVIOR [8], HyperGo [19], and DigFuzz [36], use frequency-based information and Markov-based approaches to evaluate the probability of covering different branches and paths. However, dynamic methods are prone to probability-based biases, such as favoring shorter paths, inaccurate evaluations due to insufficient hits on basic blocks and branches, and variable dependencies in constraints (§2.2). As a result, the fitness metrics designed by current dynamic methods would incorrectly prioritize paths and hinder directed fuzzing from exploring deeper code locations. Therefore, devising a more comprehensive fitness metric that effectively evaluates and distinguishes paths remains a significant challenge in multi-path scenarios.

Challenge 2: How to handle different paths in the multi-path scenarios? In directed fuzzing, effectively reaching target sites through optimal paths while ensuring the diversity of non-optimal paths explored toward targets presents a challenge that existing path scheduling optimizations fail to address. Traditional directed fuzzing methods, such as AFLGo, SelectFuzz, PDGF [35], and SDFuzz, often utilize simulated annealing algorithms to converge on optimal solutions, gradually neglecting non-optimal paths, leading to inadequate testing of those paths. Recent fuzzing methods, such as those based on Multi-Armed Bandit (MAB) models and Markov-based models, aim to optimize path allocation between fuzzing and symbolic execution in (directed) hybrid

fuzzing [8, 10] while balancing path exploration and exploitation [34, 37] to achieve optimized seed or path scheduling. However, traditional MAB models and optimization algorithms primarily set rewards based on simplistic metrics like coverage increase or target reaching, which are insufficient for addressing the multi-path issue since these metrics are sparse, and path scheduling is affected by various fuzzing contextual factors (i.e., testing conditions). For example, path characteristics (e.g., fitness metrics designed for path evaluation), the testing agent handling the path request (e.g., DGF or symbolic execution), and testing goals (e.g., path exploitation or exploration) dynamically change over time and significantly impact the effectiveness of path scheduling (§2.2). As a result, existing path scheduling approaches which rely solely on historical feedback (e.g., observed rewards) struggle to capture and handle the complex fuzzing context in multi-path scenarios. Therefore, designing a scheduling method that accounts for all relevant factors, handles complex runtime fuzzing context, converges quickly, and minimizes computational overhead remains a significant challenge.

To address the aforementioned challenges, we propose MultiGo, a directed hybrid fuzzer that combines symbolic execution and directed grey-box fuzzing for multi-path optimization.

For **Challenge 1**, we propose a new fitness metric called ***path difficulty***, which is designed to distinguish the optimal paths and more challenging ones. To mitigate the side effects of Markov-chain-based probabilistic methods, we utilize the **Poisson distribution** to estimate the probability of exploring basic blocks (i.e., block difficulty) based on statistical block frequency. The Poisson distribution does not rely on prior information and models discrete events with a constant rate, making it suitable for evaluating the likelihood that a fuzzer satisfies the path constraints of a basic block. Secondly, we combine runtime block difficulty with static metrics to form the new fitness metric, which comprehensively evaluates the fuzzer's difficulty in reaching one block through specific paths. Thirdly, we propose **the selective frequency and coverage instrumentation** method to identify and selectively instrument critical basic blocks, and design a lightweight approach to efficiently update path difficulty based on their execution frequency. Unlike static metrics, path difficulty dynamically updates based on the frequency of critical basic blocks, adapting to changing exploration of different paths for accurate path evaluation. Moreover, unlike existing dynamic approaches, the path difficulty based on the Poisson distribution eliminates variable dependencies and the bias toward shorter paths while providing accurate probability estimates even when basic block hits are sparse (§3.2.3). Based on the path difficulty, MultiGo can comprehensively evaluate different target-reachable paths, enabling DGF to distinguish between optimal paths and more challenging ones.

For **Challenge 2**, we propose using the **Contextual Multi-Armed Bandit (CMAB)** model to optimize path scheduling by comprehensively considering the impact of testing conditions on path scheduling. Unlike other optimization algorithms, the CMAB model extends traditional MAB by incorporating context vectors for context-aware decision-making, enabling efficient processing of complex data with low overhead and rapid convergence to optimal decisions, making it well-suited for directed fuzzing in multi-path scenarios. Firstly, we introduce the concept of the *fuzzing context* to capture key testing conditions, including path characteristics, testing agents, and testing goals. Path characteristics measure the difficulty of exploring paths and the likelihood of covering new target-reachable paths. Testing agents encompass fuzzing for its superior path exploration capability and symbolic execution for its capability of handling more challenging paths through constraint-solving. Testing goals encompass path exploration and exploitation to balance exploration of non-optimal paths and exploitation of optimal paths. With this fuzzing context, CMAB predicts expected rewards for path scheduling where paths with higher expected rewards indicate greater utility for multi-path optimizations. Secondly, CMAB incorporates upper confidence bounds (UCB) to balance path exploitation and exploration for finding better path scheduling schemes and enhancing the diversity of paths toward targets. As optimal paths are sufficiently explored, their UCB scores decrease, prompting the fuzzer to focus on non-optimal paths, thereby improving overall path diversity. Finally, we optimize both the fuzzing and symbolic execution components of MultiGo using the difficulty-guided path scheduling, thus achieving multi-path optimizations. By leveraging CMAB, MultiGo efficiently handles

complex fuzzing context with low overhead, accelerating target reaching and enhancing the diversity of paths toward targets to expose hidden bugs in multiple paths.

In summary, the main contributions of this paper are summarized as follows:

- We introduce a new fitness metric called *path difficulty*, which utilizes the Poisson distribution to estimate the probability of exploring basic blocks based on the observed statistical block frequency. It can comprehensively evaluate different target-reachable paths, allowing MultiGo to distinguish between the optimal paths and the more challenging ones.
- We propose using the *Contextual Multi-Armed Bandit (CMAB)* model to optimize path scheduling by comprehensively considering the impact of testing conditions on path scheduling. The difficulty-guided path scheduling with CMAB enables MultiGo to accelerate target reaching and enhance path diversity toward targets to expose hidden bugs in multiple paths.
- We implemented a tool named MultiGo and evaluated it on three benchmarks consisting of 41 programs with a total of 136 target sites. The experimental results show that MultiGo can reach target sites and expose known vulnerabilities faster than state-of-the-art baseline fuzzers. Moreover, MultiGo discovered 14 undisclosed vulnerabilities from 6 real-world programs. All discovered vulnerabilities have been reported to China National Vulnerability Database (CNNVD).
- The artifact of MultiGo is publicly available via https://anonymous.4open.science/r/multigo-C74C.

2 BACKGROUND AND MOTIVATION

2.1 Background

2.1.1 Reachability evaluation in fuzzing. **Static reachability evaluation.** Directed fuzzing approaches, such as AFLGo, assess the reachability of basic blocks by predefining target sites (e.g., code line numbers) and calculating BB distance based on the control-flow graph (CFG). A shorter distance indicates higher reachability, suggesting the fuzzer is more likely to reach the targets through specific basic blocks. Following AFLGo, the state-of-the-art directed fuzzing approaches have refined this basic block distance metric by incorporating program analysis or data flow information. For instance, Hawkeye introduces trace similarity, CAFL introduces constraint distance, DAFL considers data dependencies, and Beacon assesses path feasibility based on constraint analysis. These approaches combine detailed concepts with BB distance to create more precise fitness metrics during the pre-analysis phase, enhancing fuzzing efficiency.

Dynamic reachability evaluation. Some prior fuzzing works perform dynamic reachability evaluation by incorporating fuzzing runtime information. For example, PDGF [35] and FishFuzz [37] dynamically update the CFG after hitting indirect edges, refining reachability analysis in real time. Probability-based methods like HyperGo [19], DigFuzz [36], Marco [10], K-scheduler [27] and SAVIOR [8], use frequency information to assess the likelihood of covering specific branches and reaching a particular basic block.

2.1.2 Poisson distribution. The Poisson distribution is a widely used discrete probability distribution, which does not require prior information or a specific distribution. It describes the relationship between the frequency of an event within a fixed interval and its average occurrence rate. It assumes that events occur independently, with the probability of occurrence proportional to the length of the time interval. The probability mass function is given by:

$$P(X = k) = (e^{-\lambda} \cdot \lambda^k)/k! \tag{1}$$

Where λ denotes the average number of occurrences within the fixed time interval, while k denotes the specific number of occurrences in a given interval.

2.1.3 Contextual Multi-Armed Bandits model with LinUCB. The Contextual Multi-Armed Bandits (CMAB) model extends the traditional Multi-Armed Bandit (MAB) models by incorporating context-dependent decision-making.

By leveraging context vectors, CMAB captures key contextual information to predict expected rewards, enabling more accurate and adaptive decision-making. Compared to reinforcement learning (RL) algorithms, CMAB is a lightweight model that converges faster and incurs lower computational costs, as it focuses on short-term rewards rather than complex long-term modeling. Additionally, CMAB does not rely on prior distributions but instead learns optimized scheduling based on observed data and context information, making it more flexible and widely applicable than algorithms such as Thompson Sampling and Bayesian sampling, which require prior distributions.

LinUCB is a widely used algorithm within the CMAB framework that models each option's reward as a linear combination of the context vector and a weight vector. It employs Upper Confidence Bounds (UCB) to measure uncertainty and balance exploration and exploitation. Specifically, LinUCB exploits optimal arms with the highest expected rewards to maximize cumulative gains while exploring under-explored arms to discover better solutions. Compared to random sampling algorithms, LinUCB achieves faster convergence and lower computational overhead, making it particularly suitable for linearly separable problems such as personalized recommendations and path optimization.

2.2 Motivation

In this section, we provide a motivating example in Listing 1 to demonstrate the multi-path issue in directed fuzzing. Figure 1 is the CFG of the motivating program.

Challenge 1: Existing static methods rely on a distance-based fitness metric combined with other indicators to evaluate the reachability of basic blocks and paths. For example, AFLGo calculates the distances for $path1$, $path2$, and $path3$ as follows: $\frac{3+3+2}{3}=2.67$, $\frac{3+3+3+2}{4}=2.75$ and $\frac{3+2+1}{3}=2$, respectively. Based on these values, $path3$, having the shortest seed distance, is considered the optimal path and prioritized. However, in this case, $path3$ is not optimal, as exploring basic block 24 for a branch flip to reach the target is highly challenging. Thus, even if $path3$ is prioritized, the fuzzer may struggle to reach the target via this path. Dynamic approaches, such as HyperGo, Marco, K-scheduler [27], and DigFuzz, address this limitation by using Markov-chain-based probabilistic methods to assess path probabilities and estimate the likelihood of reaching basic blocks. However, these methods introduce bias due to three key limitations. First, Markov-chain-based methods inherently favor shorter paths, as path probability is defined as the product of branch probabilities, leading to an exponential decline as path length increases. Consequently, long-path basic blocks (e.g., basic block 16) typically have significantly lower probabilities than those in shorter paths (e.g., basic block 24). Second, these methods assume independent branches, which is often unrealistic since constraints across basic blocks (e.g., basic blocks 14, 15, and 16) are interdependent. Third, accurate branch probability estimation relies on sufficient branch hits, yet fuzzing often encounters rare branches under the same constraint. For instance, in our example, branch <24, 26> has a hit count of 4, while <24, 25> has 0, leading to inaccurate branch probability estimations. As a result,

```
1  void target(int* ptr, int* x) {
2   *x = *x + *ptr;
3  }
4  void foo(int x, int y,int z, int flag){
5   int* ptr = (int*)malloc(sizeof(int));
6   *ptr = 5;
7   if (x < 1000){
8    if (flag > 0){
9     if (flag == FREE_FLAG){
10     free(ptr);
11     target(ptr, &x); //use-after-free
12    }
13   }
```

```
14    else{
15     if (x + y < 1000)
16      if (x + z < 1000)
17       if (y + z < 1000)
18        target(ptr, &x); //integer overflow
19    }
20    return;
21   }
22   else{
23    if (x==0X224 && y==0Xaa5)
24     if (y^2 + z^2 == 0X714ebd)
25      target(ptr, &x);
26    return;
27   }
28  }
```

Listing 1. An example program with paths toward the target.

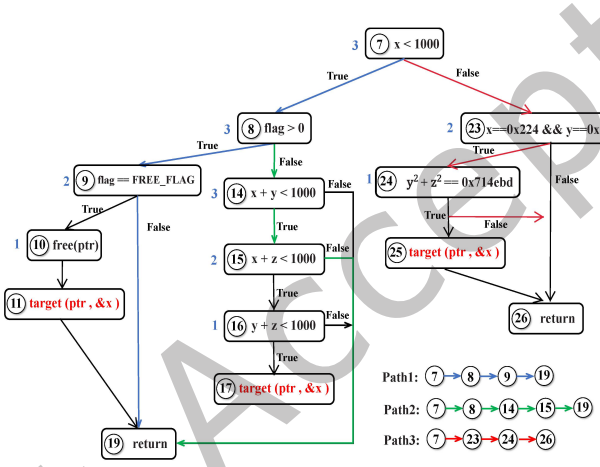

Fig. 1. The control-flow graph of the example program. The blue numbers are the BB distances using AFLGo's distance metric.

dynamic approaches still favor suboptimal paths (*path*1 and *path*3) over the truly optimal path (*path*2). Therefore, existing directed fuzzing methods struggle to distinguish paths in multi-path scenarios and lack an effective scheduling scheme. A novel fitness metric is needed to accurately evaluate and prioritize paths.

Challenge 2: In this case, the optimal path scheduling scheme is to prioritize *path*2 to improve target-reaching efficiency while still exploring *path*1 to potentially trigger a use-after-free vulnerability. However, traditional DGF approaches often employ simulated annealing algorithms, which tend to concentrate resources on exploiting optimal paths. As a result, newly discovered non-optimal paths, such as *path*1, may not receive adequate exploration during the exploitation phase. To address this issue, approaches such as EcoFuzz, FishFuzz, and Marco utilize MAB models and corresponding optimization algorithms. However, these methods rely on simplistic reward definitions, which fail to effectively handle the multi-path issue. In multi-path scenarios, optimizing path scheduling requires more than just observed rewards—it must also consider fuzzing contextual factors such as path characteristics, testing agents, and testing goals. For instance, when the testing goal is path exploitation, *path*2 should be prioritized for fuzzing as *path*2 has the lowest path difficulty and fuzzing excels at covering easy paths. Conversely, when *path*2 has been sufficiently explored, directed fuzzing should shift its focus to *path*1

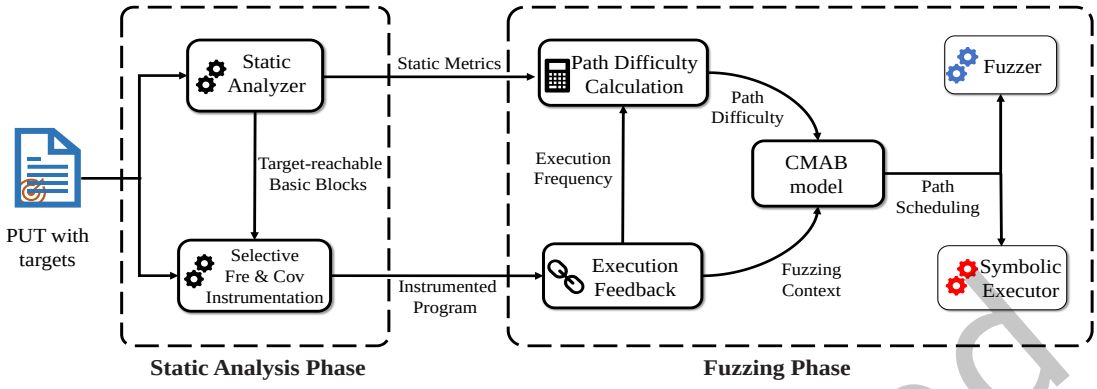

Fig. 2. The overview of MultiGo.

and *path*3 and leverage symbolic execution to navigate complex paths through its constraint-solving capability. However, existing MAB-based optimization and evolutionary algorithms struggle to capture both rewards and fuzzing contextual information simultaneously, making them ineffective for multi-path scheduling. Therefore, a novel path scheduling approach that integrates fuzzing context is essential for handling multiple paths efficiently.

3 MULTIGO

In this section, we present the design details of MultiGo, a directed hybrid fuzzing framework that combines symbolic execution with directed grey-box fuzzing to achieve multi-path optimization and comprehensive target testing, effectively addressing the multi-path issue.

3.1 Overview

This section provides a high-level overview of MultiGo, addressing the two primary challenges. MultiGo employs two key techniques to tackle each challenge: 1) the new fitness metric called path difficulty, and 2) the difficulty-based path scheduling with the Contextual Multi-Armed Bandit (CMAB) model for handling different paths in multi-path situations. Figure 2 outlines the overall workflow of MultiGo, which consists of two main phases:

Static analysis phase. MultiGo takes a program with annotated target sites as input and performs inter-procedural static analysis to identify critical basic blocks. It then selectively instruments critical basic blocks, focusing on collecting the execution frequency of these basic blocks and the coverage of their successor blocks during the next fuzzing phase (**Selective Fre & Cov Instrumentation Module**).

Fuzzing phase. MultiGo iteratively executes the instrumented program as in traditional directed hybrid fuzzing, but it also captures feedback on the execution frequency of critical basic blocks. Based on this feedback, MultiGo's path difficulty calculation module updates the path difficulty for various blocks and re-evaluates their reachability. Additionally, MultiGo uses a novel difficulty-guided path scheduling with CMAB, which combines the fuzzing context captured from testing conditions with the observed runtime information to prioritize paths for the fuzzer and the symbolic executor. This allows MultiGo to dynamically balance the trade-off between exploiting the optimal paths and exploring the non-optimal paths for conducting multi-path optimization.

3.2 Path Difficulty Metric

To overcome the limitations of existing fitness metrics in addressing the multi-path issue, in this section, we first employ the Poisson distribution to estimate the likelihood of exploring basic blocks, referred to as **block difficulty** (§3.2.1). We then integrate block difficulty with the static fitness metric to form the new fitness metric

called **path difficulty** (§3.2.2), and propose the **selective frequency and coverage instrumentation** along with a lightweight approach for path evaluation and updating (§3.2.3).

3.2.1 Block difficulty. **Block difficulty** measures the likelihood of the fuzzer exploring a particular basic block via a specific path. Since Markov-based probabilistic methods are prone to probability-based bias, we use the Poisson distribution to estimate the fuzzer's difficulty in exploring a given basic block based on its execution frequency. The Poisson distribution is particularly suitable for this fuzzing task because it does not rely on prior knowledge (e.g., path complexity derived from static analysis) and effectively models discrete events occurring at a constant average rate within a fixed interval. While fuzzers typically cannot provide path complexity analysis beforehand, they can supply historical runtime information (e.g., previous execution frequencies of basic blocks and paths) for use within the Poisson distribution. Thus, the Poisson distribution can leverage historical fuzzing data to estimate the probability of satisfying path constraints and quantify the difficulty of reaching specific basic blocks.

Given a basic block m on the execution path of seed s, the fuzzer has mutated s during the fuzzing process to generate N paths. If N_m of these N paths execute m along the same path as s, the probability of the fuzzer exploring m exactly K times by mutating s follows the Poisson distribution and is given by:

$$P(X = K) = \frac{e^{-\lambda_m} \cdot {\lambda_m}^K}{K!}, \quad \lambda_m = \frac{100 \cdot N_m}{N} \tag{2}$$

The probability that the fuzzer executes basic block m at least once, denoted as P_m, can be calculated using the cumulative probability of the Poisson distribution:

$$P_m = \sum_{K=1}^{\infty} P(X = K) = 1 - P(X = 0) \tag{3}$$

Based on the probability P_m of the fuzzer executing basic block m, we define the block difficulty, denoted as $D(m)$, as the inverse of P_m:

$$D(m) = \frac{1}{P_m} = \frac{1}{1 - e^{-\lambda_m}} \tag{4}$$

When the execution probability P_m gets higher (i.e., the block is easily explored), the difficulty $D(m)$ is lower, indicating that the block is easier to reach.

3.2.2 Path difficulty. Path difficulty combines dynamic block difficulty with static BB distance, addressing the need for both dynamic and static information. The combination of block difficulty and BB distance is rooted in the program's execution logic. To reach the target basic block, the fuzzer must first explore basic block m and then attempt a branch flip within m's path constraints to navigate toward blocks and branches closer to the target. Based on these principles, the formula for calculating path difficulty is designed as follows:

$$PD(m, T_b) = d_b(m, T_b) \cdot \mathrm{c}^{D(m)} \tag{5}$$

Where $PD(m, T_b)$ denotes the path difficulty of m, which is a combination of block difficulty and BB distance. T_b denotes the target basic block, $d_b(m, T_b)$ denotes the BB distance. The term $c^{D(m)}$ denotes a scaling factor that combines block difficulty with BB distance. Integrating dynamic block difficulty with static BB distance enables a more accurate path evaluation. When directed fuzzing first covers a path, insufficient execution may result in inaccurate branch and path probabilities, causing false positives in probability-based methods like DigFuzz, K-Scheduler, and Marco. To mitigate this, in Equation 5, when path exploration is limited, block difficulty remains close to 1, allowing reachability to be guided by static BB distance, reducing false positives. As exploration progresses, block difficulty estimates the probability of executing a basic block along the path, refining path difficulty for evaluation.

3.2.3 Path evaluation. Based on the path difficulty of basic blocks, we need to evaluate different target-reachable paths. The simplest approach is to take the arithmetic average of the path difficulty for all target-reachable basic blocks in the path, as done in many DGF works like AFLGo and Hawkeye. However, not all target-reachable basic blocks need to be considered. Since fuzzers only need to attempt a branch flip after reaching critical basic blocks to reach targets, we focus exclusively on **critical basic blocks**. If a basic block m is the target-reachable basic block (i.e., $BB\_distance \geq 0$), and more than one of its successor basic blocks (e.g., n) is both uncovered and target-reachable, then both m and n are identified as critical basic blocks in path p. Therefore, we propose a selective frequency and coverage instrumentation method to identify and collect the execution information of critical basic blocks. The collected information, including the hit counts of basic blocks and paths (i.e., execution frequency), will be used to update path evaluation and integrated into the CMAB model to optimize path scheduling and power scheduling (§3.3).

Selective frequency and coverage instrumentation. To identify critical basic blocks and calculate their path difficulty, we perform selective frequency and coverage instrumentation. During the static analysis phase, we first construct the CFG and identify the successor basic blocks of target-reachable blocks. If a basic block m is the target-reachable basic block, and more than one of its successor basic blocks is target-reachable, then m and its target-reachable successor basic blocks will be selectively instrumented to track their execution status and frequencies. During the fuzzing phase, we track the execution path of each seed and check the instrumented successor basic blocks to determine whether the instrumented successors have been executed. If any instrumented successors are uncovered, these uncovered successor basic blocks and their corresponding target-reachable basic block are identified as critical basic blocks. MultiGo then selectively collects execution frequencies for critical basic blocks and calculates the geometric mean of the path difficulty for all critical basic blocks. This geometric mean is then used as the reachability score for the path, ensuring that paths with lower difficulty are given appropriate priority. Since critical basic blocks represent only a small subset of all basic blocks, this selective instrumentation significantly reduces fuzzing overhead compared to instrumenting all basic blocks and collecting frequency data for each one.

Path evaluation and updating. We use the geometric mean of path difficulty to evaluate multiple paths:

$$\widetilde{PD}(p, T_b) = \sqrt[|\xi_b(p)|]{\prod_{m \in \xi_b(p)} PD(m, T_b)} \tag{6}$$

Where p denotes the path, $\widetilde{PD}(p, T_b)$ denotes the geometric mean of path difficulty to evaluate p, $\xi_b(p)$ denotes the set of critical basic blocks in the execution path, and $|\xi_b(p)|$ denotes the number of basic blocks in $\xi_b(p)$.

Since path difficulty is dynamically updated based on the fuzzing runtime information, a lightweight approach is needed to simplify its calculation. We can reduce computational complexity by converting operations such as **exponentiation** and **square roots** into simpler **addition** and **multiplication**, and move any necessary logarithmic operations to the static phase to minimize fuzzing overhead. Therefore, we first combine Equation 5 and Equation 6 to streamline the calculation of the geometric mean.

$$\widetilde{PD}(p, T_b) = \exp\left\{\frac{\sum_{m \in \xi_b(p)} (\log(d_b(m, T_b) \ + \ D(m))}{|\xi_b(p)|}\right\} \tag{7}$$

Next, we simplify Equation 6 primarily based on Equation 5. First, we take the logarithm of both sides of Equation 5 to transform the multiplications and square root operations into summation operations, resulting in the expression $\sum_{m \in \xi b(p)} \log(d_b(m, T_b) \cdot \mathrm{c}^{D(m)})$. To further optimize the computation, we set the constant c in Equation 5 to e, which converts the multiplication into an addition: $\log(d_b(m, T_b) \cdot \mathrm{c}^{D(m)}) = \sum_{m \in \xi_b(p)} (\log(d_b(m, T_b) + D(m))$. Finally, by exponentiating both sides, we obtain the simplified formula. This method allows the fuzzer to perform only addition operations during the forking process, reducing computational overhead.

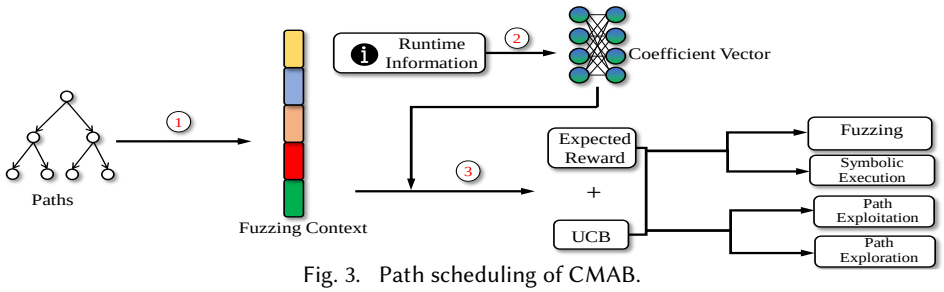

Fig. 3. Path scheduling of CMAB.

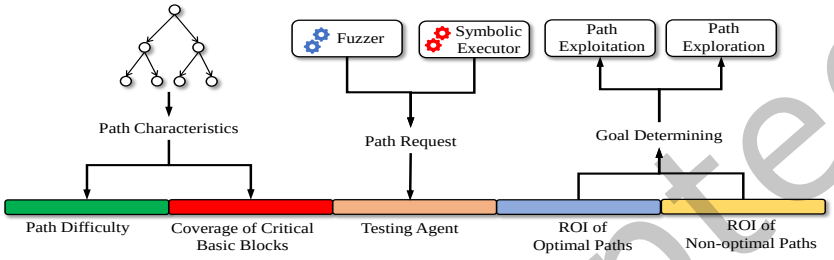

Fig. 4. Composition of fuzzing context.

Recall the motivating example from §2.2, the path difficulty metric used in path evaluation guides the fuzzer to favor *path*2. First, because block 24 is difficult to explore, the path difficulty of block 24 increases as fuzzing progresses, leading to an overall increase in the path difficulty of *path*3. Second, the fuzzer has difficulty satisfying the condition in block 9 to reach the critical basic block 10, leading to increased difficulty for block 10 and the overall path difficulty of *path*1 each time *path*1 is selected but block 10 remains unhit. Consequently, the path difficulties of *path*1 and *path*3 eventually exceed that of *path*2, prompting the fuzzer to prioritize *path*2. Thus, path difficulty, integrating dynamic block difficulty with static BB distance, mitigates false positives and improves path evaluation in multi-path scenarios.

3.3 Difficulty-guided Path Scheduling with CMAB Model

To handle different paths in multi-path scenarios, we propose using the Contextual Multi-Armed Bandit (CMAB) model to design the difficulty-guided path scheduling. Compared to other models, such as deep reinforcement learning methods that face challenges like data scarcity and state-space explosion in fuzzing, the CMAB model more effectively schedules paths and allocates resources between the fuzzer and symbolic executor within only CPU resources and a limited time budget. It effectively utilizes context vectors to capture testing conditions and make context-dependent decisions without high overhead or requiring prior distributions.

We use Figure 3 to illustrate the process of difficulty-guided path scheduling with the CMAB model: ① We introduce the concept of the *fuzzing context*, represented as a context vector that captures key testing conditions (§3.3.1). ② During fuzzing, the runtime information is collected to train optimal coefficient vectors, which are then combined with context vectors to predict the expected rewards of selecting different paths (§3.3.2). Paths with higher expected rewards are more likely to be prioritized. ③ CMAB leverages the Upper Confidence Bound (UCB) to balance path exploration and exploitation, and then combines the UCB score with the expected reward to calculate the CMAB score (§3.3.3). As fuzzing progresses, the UCB scores of well-tested optimal paths gradually decrease, prompting CMAB to explore non-optimal paths, thus accelerating target reaching while enhancing the diversity of paths explored toward targets.

3.3.1 Fuzzing context. We consider three fuzzing contextual factors that influence path scheduling efficiency in multi-path scenarios: **path characteristics**, **testing agents**, and **testing goals**. Moreover, we define *fuzzing context* as a context vector f_p, represented as (PD_p, CB_p, AG, ROI_{opt}, ROI_{non}), to capture the testing conditions for path p. Next, we illustrate the context vector f_p of path p in Figure 4 and explain the rationale behind selecting these testing conditions.

Path characteristics. The fuzzing context incorporates path characteristics, including path difficulty and the coverage of critical basic blocks. These factors significantly impact path scheduling efficiency. For example, given a path p, its path difficulty, represented as PD_p, determines how easily fuzzers can traverse it and ultimately reach target sites. Additionally, its coverage of critical basic blocks, represented as CB_p, indicates whether further exploration of p is necessary. Specifically, high path difficulty hinders fuzzers from reaching critical basic blocks, whereas low coverage of critical basic blocks restricts fuzzers to performing branch flips within a limited subset of basic blocks, thereby limiting fuzzers' ability to cover paths closer to targets. Consequently, the path characteristics of p, defined by PD_p and CB_p, determine its priority for path scheduling. By integrating path characteristics into the fuzzing context, CMAB can effectively predict the expected reward of selecting different paths.

Testing agents. The fuzzing context incorporates the testing agents, which include fuzzing and symbolic execution. Fuzzing excels in path exploration, enabling rapid coverage of easy paths. Symbolic execution provides powerful constraint-solving capabilities to efficiently handle complex paths. For instance, in the motivating example from §2.2, while fuzzing struggles to satisfy the path constraint in basic block 9 of $path1$, symbolic execution efficiently solves the constraint and generates valid inputs within a short time. Thus, assigning paths to the appropriate testing agents plays a crucial role in multi-path scheduling. We use AG as a binary variable representing the testing agent, where 0 corresponds to fuzzing and 1 corresponds to symbolic execution. Based on runtime information about each agent's testing results on different paths, CMAB can predict the expected reward of selecting a specific testing agent for a given path, thus optimizing the overall scheduling strategy.

Testing goals. The fuzzing context incorporates testing goals, which include path exploitation and path exploration. For path exploitation, prioritizing paths with low difficulty allows fuzzers to reach target sites faster. In contrast, for path exploration, paths with higher coverage of critical basic blocks are preferred to enhance the diversity of paths toward targets. Thus, testing goals play a crucial role in testing conditions, as path scheduling under different testing goals directly impacts efficiency. To quantify this, we define the **global return on investment (ROI)** as the ratio of cumulative rewards to cumulative time costs for both path exploitation and exploration:

$$ROI_{opt} = \frac{R_{opt}}{T_{opt}}, \quad ROI_{non} = \frac{R_{non}}{T_{non}} \tag{8}$$

Where ROI_{opt} and ROI_{non} denote the global return on investment of path exploitation and path exploration, respectively. R_{opt} and R_{non} denote their corresponding cumulative rewards, while T_{opt} and T_{non} denote their corresponding cumulative time costs. By incorporating testing goals, the fuzzing context enables CMAB to balance efficiency in reaching targets with covering more paths toward targets. When ROI_{opt} exceeds ROI_{non}, selecting optimal paths yields higher expected rewards, guiding CMAB to accelerate target reaching. As these paths become extensively tested, ROI_{opt} gradually declines below ROI_{non}, indicating that exploring non-optimal paths—those with higher coverage of critical basic blocks—achieves higher expected rewards. Consequently, CMAB shifts its focus to such paths to enhance path diversity.

3.3.2 Expected reward. During fuzzing, CMAB predicts the expected reward for selecting each path, which is then utilized for path scheduling. The expected reward corresponds to the anticipated benefit when MultiGo selects a path p. Paths with higher expected rewards typically indicate either lower path difficulty or higher coverage of critical basic blocks within the specific testing agent and testing goal. To calculate the expected

reward, MultiGo maintains a tuple (θ_p, f_p):

$$E[f_p] = \theta_p^T f_p \tag{9}$$

Where $E[f_p]$ denotes the expected reward of selecting path p, θ_p is the optimal coefficient vector in the LinUCB algorithm and f_p denotes the context vector of p extracted from the fuzzing context. At each trial, the algorithm updates θ_p based on previous observations to better estimate the reward for future decisions. The estimation of θ_p is based on the maintained covariance matrix CV_p and the reward-related vector RV_p:

$$\theta_p = {CV_p}^{-1} RV_p \tag{10}$$

The covariance matrix CV_p is primarily used to represent the uncertainty associated with path p and helps the algorithm make smarter decisions when balancing exploration and exploitation. At the initialization stage, the covariance matrix for each path is set as an identity matrix. As fuzzing progresses, it is updated based on the context vector:

$$CV_p = CV_p + f_p f_p^T \tag{11}$$

The reward-related vector RV_p is crucial for accumulating reward information and is updated as the algorithm runs. At the beginning of the fuzzing process, the reward-related vector for each path is initialized to zero. As rewards are observed from selected arms, the reward vector is updated:

$$RV_p = RV_p + f_p \frac{1}{\overline{DP_{p \longrightarrow q}}} \tag{12}$$

Where $\overline{DP_{p \longrightarrow q}}$ denotes the average path difficulty of new paths q generated when MultiGo selects path p. The lower the average path difficulty of the new paths q generated after selecting path p, the higher the immediate reward MultiGo receives. When computing the expected reward based on the fuzzing context, CMAB only needs to maintain and update the optimal coefficient vectors. The complexity of updating optimal coefficient vectors is only $O(N \times 5)$, where N is the number of paths, and 5 is the dimension of the fuzzing context vector. This allows CMAB to efficiently predict expected rewards with minimal computational overhead.

3.3.3 UCB and CMAB scores. In addition to using the expected reward to predict the benefits of selecting different paths under specific testing conditions, we employ the Upper Confidence Bound (UCB) in LinUCB to balance exploration and exploitation. By computing UCB scores, CMAB favors less-explored paths in high-uncertainty cases and prioritizes paths with high UCB scores. This approach ensures that decision-making accounts for both expected rewards and uncertainty, allowing CMAB to explore promising paths while progressively converging toward better solutions. The UCB score of a path p, denoted as U_p, is calculated using the LinUCB algorithm:

$$U_p = \sqrt{f_p^T CV_p^{-1} f_p} \tag{13}$$

With both the expected reward and UCB score, CMAB assigns selection probabilities based on a combined score, favoring paths with higher CMAB scores:

$$Score(p) = E[f_p] + 0.5U_p \tag{14}$$

In Equation 14, CMAB scores are computed by combining $E[f_p]$ and U_p, allowing MultiGo to balance the trade-off between target-reaching efficiency and path diversity. Initially, when fuzzing launches, CMAB prioritizes paths with the highest expected rewards to accelerate target reaching. As these paths are selected more frequently, their uncertainty, represented by the UCB score, decreases, eventually causing the CMAB scores of well-tested paths to fall below those of under-explored non-optimal paths. Consequently, MultiGo prioritizes under-explored paths with higher uncertainty, thus promoting path exploration to enhance path diversity toward targets.

3.3.4 Path scheduling optimizations. MultiGo is a directed hybrid fuzzing framework that integrates fuzzing and symbolic execution, both serving as testing agents that concurrently initiate path requests. By leveraging difficulty-guided path scheduling, we optimize path scheduling in both the fuzzing and symbolic execution components, effectively addressing the multi-path issue discussed in §2.2.

Path scheduling in the fuzzing component of MultiGo. First, we optimize path prioritization in fuzzing by setting the testing agent indicator $AG = 0$ in the context vector. We then compute the expected rewards, UCB scores, and CMAB scores for all paths. MultiGo subsequently schedules the paths with the highest CMAB scores for fuzzing. Second, we optimize power scheduling by redesigning the power schedule, integrating the CMAB scores with the base energy assigned by AFL:

$$P(s, T_b) = \frac{Score(s)}{\overline{Score}} P_{afl}(s) \tag{15}$$

Where s denotes the seed whose path is path p, $P_{afl}(s)$ denotes the energy assigned by AFL's power schedule, and $P(s, T_b)$ represents the final assigned seed energy after optimization. AFL's power schedule allocates basic energy to the seed based on its characteristics, such as execution speed and bitmap size. $Score(s)$ denotes the CMAB score of path p, which is calculated using Equation 14. $\overline{Score}$ denotes the mean CMAB score across all seeds.

Recall from §2.2 that at the start of testing, $path2$ has the lowest path difficulty and the highest coverage of critical basic blocks, giving it the highest expected rewards. Since all paths have equal UCB scores initially, the fuzzing component of MultiGo prioritizes $path2$, allocating more energy for path exploitation. As $path2$ undergoes extensive fuzzing, its UCB score gradually decreases, eventually falling below those of $path1$ and $path3$, resulting in a lower CMAB score compared to these paths. As a result, fuzzing shifts to explore the non-optimal paths ($path1$ and $path3$). Therefore, through optimized path scheduling based on CMAB scores and path difficulty, the fuzzing component first identifies and exploits $path2$ to accelerate target reaching, while subsequently exploring $path1$ and $path3$ to improve overall path coverage toward targets.

Path scheduling in the symbolic execution component of MultiGo. Firstly, we optimize the path prioritization by setting the testing agent indicator $AG = 1$ in the context vector and computing expected rewards, UCB scores, and CMAB scores for all paths. MultiGo then prioritizes paths with the highest CMAB scores for symbolic execution.

Secondly, to improve the efficiency of constraint-solving in symbolic execution, we dynamically adjust time budgets for different paths based on CMAB scores, gradually abandoning highly complex paths that cannot be resolved by symbolic execution. Constraint-solving time budgets significantly impact symbolic execution efficiency, influencing how long the solver spends on attempting to generate new inputs before terminating. Given a seed s, the symbolic executor tracks its execution path p and identifies critical basic blocks and unexplored branches, represented as $<BB_c, BB_{sc}>$, where BB_c is a critical basic block and BB_{sc} is a target-reachable block not yet covered by fuzzing. The executor attempts to solve the path constraint set from the entry point to BB_{sc}, but its complexity is unknown, often causing excessive solving time (e.g., 30 minutes) without success. Traditionally, a fixed time budget (e.g., 10 seconds in SymCC [25]) is set manually, after which an exception is thrown, and the path is abandoned.

To reduce wasted computation on unsolvable paths while avoiding premature termination of solvable ones, we introduce an adaptive time budget based on CMAB scores. Each path is initially assigned a 10-second budget, following SymCC. The time budget is then dynamically adjusted based on the CMAB score, path constraint complexity, and historical execution time:

$$ST_i(p) = ST_{i-1}(p) + \frac{Score(p) \cdot CB_p}{e^i}, i = 1, 2, \ldots \tag{16}$$

Where $ST_i(p)$ denotes the time budget for p in the i-th selection, CB_p denotes the number of critical basic blocks in path p. As the number of symbolic executions for p increases, $Score(p)$ gradually decreases if symbolic execution repeatedly fails to generate valid inputs to cover path p. Additionally, we introduce the parameter e^i, which grows exponentially as i increases. Thus, the increments in the time budget gradually decrease until they become negligible, preventing excessive allocation to highly complex, unsolvable paths while encouraging symbolic execution to focus on simpler unexplored paths.

Recall from §2.2, fuzzing generates inputs for easy paths much faster than symbolic execution, making it more likely to cover unexplored branches (e.g., <15, 16> and <16, 17>) before symbolic execution. As a result, when symbolic execution selects easy paths, its newly generated inputs often fail to cover new branches since fuzzing has already explored them, leading to lower observed rewards. Based on cumulative rewards and the fuzzing context, CMAB predicts that selecting complex paths yields higher expected rewards for symbolic execution. Consequently, CMAB assigns complex paths (e.g., $path1$ and $path3$ in §2.2) to symbolic execution. Furthermore, with our adaptive solving strategy, if symbolic execution fails to solve highly complex path constraints (e.g., $path3$), CMAB reduces its CMAB score and thus decreasing its selection probability. Simultaneously, the solving strategy lowers its time budget, reallocating resources to more feasible paths. By leveraging difficulty-guided path scheduling with CMAB, fuzzing is more likely to be assigned easy paths, while symbolic execution focuses on more challenging ones, improving path scheduling efficiency in multi-path scenarios.

4 IMPLEMENTATION

The implementation of MultiGo mainly consists of three components: a static analyzer, a fuzzer, and a symbolic executor. For the static analyzer, we leverage the static analysis framework LLVM 11.0 and Clang 11.0 and use the LLVM IR to instrument the program. The fuzzer is built on AFLGo, and the symbolic executor is built on SymCC. The implementation of MultiGo involves about 2000 lines of C/C++ and Rust code. The artifact of MultiGo is publicly available via https://anonymous.4open.science/r/multigo-C74C.

5 EVALUATION

To evaluate the effectiveness of MultiGo, we conducted experiments aiming to answer five research questions:
RQ1: How does MultiGo perform in terms of reaching the target sites?
RQ2: How does MultiGo perform in terms of exposing the vulnerabilities in the target sites?
RQ3: How do the path difficulty fitness metric (i.e., PD) and the difficulty-guided path scheduling affect the performance of MultiGo?
RQ4: How do the multi-path optimizations of MultiGo affect the fuzzing process?
RQ5: How does MultiGo perform in terms of discovering new vulnerabilities?

5.1 Evaluation Setup

Evaluation criteria. We mainly use two types of criteria to evaluate the performance of directed fuzzing techniques.

(1) Time-to-Reach (**TTR**) measures the time taken to generate the first input that can reach a specific target site.

(2) Time-to-Expose (**TTE**) measures the time required to expose known or undisclosed vulnerabilities in the target sites. A crash observed at the target site indicates that the fuzzer has successfully exposed the vulnerability.

Evaluation benchmarks. We selected three benchmarks consisting of 41 real-world programs in total for our evaluation, including the UniBench, the CVE-Benchmark used in AFLGo [5], and the Google-Fuzzer-Test-Suite. These benchmarks are widely adopted by state-of-the-art directed fuzzers, which facilitate the comparison against

Table 1. Comparison of different configurations across different fuzzers

Fuzzers	Fitness Metrics	Fuzzing Strategies	Hybrid Fuzzing Strategies
SymCC	N/A	Coverage-guided Gray-box Fuzzing	Coverage-guided Strategy used in SymCC
SymGo	Distance	Directed Gray-box Fuzzing	Coverage-guided Strategy used in SymCC
SymGo-PD	Path Difficulty	Directed Gray-box Fuzzing	Coverage-guided Strategy used in SymCC
MultiGo-PB	Probabilistic Metric used in DigFuzz	Directed Gray-box Fuzzing	CMAB model
MultiGo	Path Difficulty	Directed Gray-box Fuzzing	CMAB model

the baseline fuzzers. We also used 9 extra real-world programs to test MultiGo's performance in discovering new vulnerabilities. The extra programs are detailed in Table 6.

(1) UniBench [16] offers a collection of 20 diverse real-world programs that are widely used by state-of-the-art fuzzers, such as WindRanger and PDGF. We use AFL++ to conduct a preliminary 48-hour pre-testing on the UniBench programs. During this process, we identified code locations that took over 1 hour to reach. To ensure fairness and objectivity, we randomly selected **5** code locations from those identified locations as target sites in our evaluation.

(2) The CVE-Benchmark was used by AFLGo [5] and widely adopted by recent works [7, 11, 21]. It is used to reproduce known vulnerabilities in order to compare the efficiencies of MultiGo with state-of-the-art fuzzers.

(3) To better understand MultiGo's performance in finding different types of vulnerabilities, we also evaluated it on the standard fuzzing benchmark Google-Fuzzer-Test-Suite (GFTS) [1]. This evaluation on a standardized test suite provides a fair evaluation of MultiGo's capabilities.

Baselines. In our evaluation, we compared MultiGo with the state-of-the-art directed fuzzers and hybrid fuzzers. The baseline fuzzers had to meet three specific criteria: 1) The baseline fuzzers had been publicly available at the time of writing this paper; 2) The baseline directed fuzzers should support target setting. Most directed fuzzers rely on predefined targets to guide their testing process. Thus, we excluded directed fuzzers such as ParmeSan and SAVIOR, which do not allow target setting; 3) The symbolic execution engine of baseline hybrid fuzzers should have the same constraint-solving capability as MultiGo's symbolic execution engine. Based on these criteria, we selected state-of-the-art directed fuzzers, including AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL, as well as two state-of-the-art hybrid fuzzers, SymCC and SymGo. SymGo is a directed hybrid fuzzer formed by combining AFLGo with the symbolic execution engine of SymCC, enabling it to possess directed testing capabilities. The comparison of different configurations (e.g., fitness metrics, fuzzing strategies, and hybrid fuzzing strategies) across different fuzzers is outlined in Table 1.

Experiment settings. We conducted the experiments on the server equipped with Intel(R) Xeon(R) Gold 6133 CPU @ 2.50GHz with 80 cores and 512G RAM, and used Ubuntu 20.04 LTS as the operating system. All the experiments were repeated **5** times within a time budget of **24 hours**. When testing the programs, we used the benchmarks' recommended seed corpus as initial seeds. Given that MultiGo requires two CPU cores to simultaneously launch both the fuzzing and symbolic execution instances, the compared fuzzers also employed parallel fuzzing by launching two fuzzing instances.

Experimental results analysis. We utilize the Mann-Whitney U test (p-value) to measure the statistical significance and the Vargha-Delaney statistic ($\hat{A}_{12}$) [3] to measure the probability of one technique performing better than the other. In the presentation of experimental results, we use the entry "N/A" to indicate that the fuzzer failed to compile the program due to code issues, and "T.O." to indicate that the fuzzer couldn't reach the target site within the allocated 24-hour time budget. For WindRanger, some entries are marked as "N/A" due to segmentation fault errors or being unable to obtain distance information during program testing. As for Beacon, most entries showing "N/A" might be because Beacon is incompatible with UniBench. For "N/A" entries, we did not use them to calculate the speedups and p-values. For the "T.O." entries, we believe that these fuzzers

might still reach the targets if given more testing time than 24 hours. Thus, we use a slightly larger value of 1500 minutes to calculate speedups and p-values as an upper-bound estimate.

5.2 Reaching Target Sites

5.2.1 Analysis of TTR results. To answer **RQ1**, we tested programs from UniBench, with a total of 100 target sites, and evaluated the TTR of different fuzzers. Detailed results of TTR and the target code locations are listed in Table 2. According to the results of TTR, MultiGo can reach the most (92/100) target sites compared to AFLGo (44/100), SelectFuzz (64/100), Beacon (26/100), WindRanger (36/100), DAFL (60/100), SymCC (49/100), and SymGo (51/100) within the time budget. Moreover, on most target sites (89/100), MultiGo outperforms all other fuzzers and achieves the shortest TTRs. In terms of the mean TTR of reaching the target sites, MultiGo demonstrates 36.38×, 11.90×, 37.49×, 11.09×, 26.68×, and 27.96× speedup compared to AFLGo, SelectFuzz, WindRanger, DAFL, SymCC, and SymGo, respectively. Since Beacon is incompatible with UniBench and most of its entries show "N/A", we do not compare the mean TTR results with Beacon. We conducted both the Mann-Whitney U test (p-value) and the Vargha-Delaney test ($\hat{A}_{12}$), and found that all the p-values were less than 0.05, and the mean $\hat{A}_{12}$ against AFLGo, SelectFuzz, WindRanger, DAFL, Symcc, and SymGo is 0.92, 0.78, 0.89, 0.76, 0.83, and 0.89, respectively. Based on the above analysis, we can conclude that **MultiGo can reach the target sites faster than baseline fuzzers on the UniBench**.

5.2.2 Analysis of TTR performance of different fuzzers. We also conducted a performance analysis to explore why MultiGo outperforms other fuzzers in terms of mean TTR. SelectFuzz, Beacon, and DAFL employ selective instrumentation or path-pruning methods to exclude code locations that are irrelevant to the targets. Additionally, WindRanger designs precise fitness metrics based on data flow information. However, during the testing on UniBench targets, we discovered two limitations. Firstly, the baseline fuzzers still struggle to cover difficult (complex) paths. For instance, in the case of `jhead` with 5 targets (#11-#15) and `tcpdump` with 5 targets (target #41-#45), although SelectFuzz, Beacon, and DAFL exclude irrelevant code locations to enhance directedness, they still find it hard to cover the difficult paths and are hindered from reaching targets in a short time. Although SymCC utilizes a symbolic execution engine to help navigate complex paths, its solving strategy fails to adapt to the fuzzing context. As a result, it is unable to successfully cover complex paths leading to targets. Secondly, the baseline directed fuzzers encounter difficulties in accurately identifying target-reachable basic blocks. AFLGo outperformed other baseline fuzzers when testing targets #44 and #53. This is because, during static identification, some reachable basic blocks were mistakenly labeled as unreachable. As a result, these blocks and branches were pruned or selectively not instrumented, slowing down the fuzzer's progress in reaching targets. In fact, all baseline fuzzers experience similar challenges in the static analysis, such as the inaccuracy of identifying target locations, leading to compilation failures (SelectFuzz, Beacon, and DAFL) or missing critical distance information (AFLGo and WindRanger). Among them, Beacon exhibited the most significant issues.

Both MultiGo and baseline fuzzers experience timeouts when testing programs from the UniBench benchmark due to the challenges posed by complex paths in large-scale programs. Firstly, current state-of-the-art directed gray-box fuzzers, such as SelectFuzz, DAFL, and Beacon, often encounter timeouts with these complex paths since their random mutations struggle to satisfy complex path constraints within a limited time budget. Secondly, symbolic execution techniques, like SymCC and SymGo, remain unable to solve overly complex constraint paths. As a result, both DGF and symbolic execution methods, including MultiGo, may fail to handle complex paths within a 24-hour time budget, leading to timeouts. However, MultiGo mitigates this challenge more effectively, as it not only experiences significantly fewer timeouts but also achieves much shorter TTRs compared to the baselines.

Table 2. The TTR results on programs from UniBench

No	Program	Version	Target sites	AFLGo	SelectFuzz	Beacon	WindRanger	DAFL	SymCC	SymGo	SymGo-PD	MultiGo-PB	MultiGo
1			parser.c:281	T.O.	53.2m	99.4m	61.1m	**34.6m**	283.4m	T.O.	117.4m	53.9m	47.3m
2			c.c:1783	12.8m	8.1m	22.1m	**6.4m**	10.1m	10.5m	9.4m	8.6m	7.1m	7.4m
3	cflow	1.6	parser.c:105	0.8m	0.5m	13.5m	0.9m	**0.4m**	0.7m	1.2m	1.3m	2.8m	1.9m
4			parser.c:1223	1.2m	**0.7m**	0.8m	2.4m	1.1m	2.4m	1.6m	8.1m	13.1m	9.1m
5			parser.c:108	12.8m	5.5m	68.1m	8.3m	7.2m	7.9m	8.6m	5.1m	4.8m	**4.8m**
6			Ap4SampleEntry.cpp:780	1421.1m	1011.4m	N/A	1365.2m	1231.6m	1351m	1210.2m	1053.7m	984.3m	**949.3m**
7			Ap4TrunAtom.cpp:138	1042.8m	452.1m	N/A	T.O.	312.4m	782.4m	892.6m	437.8m	399.2m	**298.4m**
8	mp42aac	Bento4 1.5.1-628	Ap4AtomFactory.cpp:300	912.5m	822.6m	N/A	1042.6m	842.1m	951.3m	886.3m	776.4m	693.4m	**725.2m**
9			Ap4DvccAtom.cpp:66	842.1m	777.4m	N/A	1042.7m	982.4m	757.3m	523.1m	392.1m	429.1m	**325.9m**
10			Ap4AtomFactory.cpp:490	T.O.	529.4m	N/A	T.O.	**312.6m**	T.O.	1324.2m	715.6m	593.1m	339.1m
11			exif.c:1339	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	301.4m	143.1m	**39.4m**
12			exif.c:1327	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	84.6m	5.2m	**4.8m**
13	jhead	3.00	iptc.c:143	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	556.3m	209.6m	**197.4m**
14			iptc.c:91	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	711.3m	**15.7m**	18.4m
15			makernote.c:174	T.O.	T.O.	N/A	T.O.	T.O.	932.5m	1102.4m	218.3m	59.8m	**26.3m**
16			layer3.c:1116	1142.6m	N/A	N/A	984m	N/A	744.3m	841.2m	95.2m	37.5m	**10.8m**
17			interface.c:690	1098.0m	N/A	N/A	324.1m	N/A	152.4m	81.9m	31.5m	24.9m	**17.1m**
18	mp3gain	1.5.2	mp3gain.c:602	T.O.	352.1m	N/A	T.O.	489.6m	T.O.	T.O.	482.5m	316.4m	**98.4m**
19			layer3.c:1133	672.4m	**32.4m**	N/A	529.6m	56.4m	231.5m	279.1m	649.2m	204.7m	42.7m
20			apetag.c:341	290m	132m	N/A	91.2m	104.8m	95.6m	132.6m	62.9m	20.7m	**12.3m**
21			bitstream.c:452	148.2m	89.1m	N/A	256.4m	67.4m	102.5m	97.2m	76.3m	**32.6m**	62.1m
22			console.c:61	674.2m	98.4m	N/A	482.4m	234.6m	172.3m	281.6m	84.3m	66.9m	**18.3m**
23	lame	3.99.5	quantize_pvt.c:441	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
24			VbrTag.c:778	26.5 m	**1.6m**	N/A	39.1m	2.6m	2.5m	1.4m	6.2m	9.3m	7.9m
25			get_audio.c:1605	T.O.	189.1m	N/A	T.O.	221.4m	T.O.	T.O.	298.2m	208.4m	**43.5m**
26			jp2_cod.c:841	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**451m**
27			jpc_dec.c:1393	T.O.	846.2m	N/A	653m	462.3m	94.3m	89.1m	31.5m	12.5m	**6.2m**
28	imginfo	jasper 2.0.12	jas_image.c:378	472.4m	253.7m	N/A	349.2m	**204.8m**	409.2m	453.6m	439.4m	84.6m	342.4m
29			jas_stream.c:823	T.O.	N/A	N/A	T.O.	N/A	894.4m	1123m	206.7m	4.9m	**1.2m**
30			jpc_dec.c:591	521.4m	321.7m	N/A	452.4m	246.4m	552.9m	504.3m	284.3m	209.1m	**121.2m**
31			io-pcx.c:495	857.4m	423.5m	**246.1m**	T.O.	623.2m	811.7m	763.3m	498.2m	410.6m	329.6m
32			io-tif.c:511	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	998.4m	T.O.	**841m**
33	gdk-pixbuf-pixdata	gdk-pixbuf 2.31.1	pixops.c:989	159.2m	82.4m	254.3m	124.5m	**46.3m**	148.2m	154.6m	89.8m	58.9m	55.2m
34			io-jpeg.c:691	T.O.	236.4m	T.O.	T.O.	332.2m	T.O.	T.O.	481.7m	291.6m	**54.3m**
35			io-tga.c:360	126m	**46.2m**	89.4m	101.0m	66.4m	94.6m	111.7m	98.4m	86.3m	58.1m
36			jv_dtoa.c:3122	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	**1223m**
37			jv_dtoa.c:2004	T.O.	1249.1m	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	**1026.2m**
38	jq	1.5	jv_dtoa.c:2518	T.O.	1431.5m	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	1325.2m	**851.2m**
39			jv_unicode.c:42	T.O.	**682.1m**	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	1198.0m	723.4m
40			jv_dtoa.c:3044	T.O.	225.2m	T.O.	N/A	652.4m	T.O.	T.O.	736.4m	104.7m	**42.1m**
41			print-radius.c:992	283.4m	102.4m	N/A	**62.4m**	78.2m	395.7m	291.3m	168.4m	89.4m	68.4m
42			print-ntp.c:412	1436.7m	721.4m	N/A	974.3m	1123.5m	T.O.	1311.2m	329.1m	94.6m	**36.9m**
43	tcpdump	4.8.1	print-rsvp.c:1252	T.O.	**98.4m**	N/A	T.O.	128.6m	T.O.	T.O.	532.4m	238.4m	182.3m
44			print-snmp.c:607	359.3m	562.3m	N/A	192.1m	465.8m	583.2m	412.6m	159.7m	42.3m	**14.3m**
45			print-l2tp.c:606	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
46			captoinfo.c:189	T.O.	N/A	N/A	N/A	N/A	T.O.	T.O.	68.5m	20.6m	**14.6m**
47			parse_entry.c:317	982.4m	623.4m	N/A	N/A	781.5m	1032.6m	966.8m	644.7m	**410.7m**	455.9m
48	tic	ncurses 6.1	name_match.c:111	1186.2m	93.6m	N/A	N/A	43.1m	582.3m	866.4m	108.2m	33.7m	**24.3m**
49			comp_scan.c:860	264.1m	N/A	N/A	N/A	42.3m	18.4m	49.3m	6.7m	4.9m	**3.2m**
50			visbuf.c:74	T.O.	T.O.	N/A	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
51			util.c:43	826.9m	123.3m	N/A	622.4m	T.O.	822.5m	786.4m	110.7m	46.2m	**45.6m**
52			avc.c:1023	T.O.	78.4m	N/A	T.O.	T.O.	T.O.	T.O.	832.8m	139.5m	**16.9m**
53	flvmeta	1.2.1	info.c:543	246.9m	423.1m	N/A	185.9m	325.4m	74.2m	167.2m	52.9m	17.7m	**15.2m**
54			flvmeta.c:1023	T.O.	211.4m	N/A	T.O.	69.3m	T.O.	T.O.	236.7m	66.3m	**53.2m**
55			check.c:769	T.O.	N/A	N/A	T.O.	N/A	T.O.	T.O.	526.7m	97.4m	**85.2m**
56			tif_dirinfo.c:1595	1423.5m	238.4m	N/A	T.O.	341.2m	582.3m	681.3m	319.2m	138.2m	**122.4m**
57			tif_read.c:335	T.O.	**106.7m**	N/A	T.O.	214.9m	T.O.	T.O.	T.O.	T.O.	1233.7m
58	tiffsplit	libtiff 3.9.7	tif_jbig.c:277	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**912.9m**
59			tif_dirread.c:1977	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	467.1m	**421.6m**
60			tif_strip.c:154	1828m	48.3m	N/A	T.O.	32.4m	1049.1m	1139m	91.5m	29.4m	**7.2m**
61			elf.c:8796	249.4m	30.2m	**18.5m**	281.5m	23.4m	181.7m	238.3m	203.4m	374.8m	261.4m
62			elf.c:8793	T.O.	T.O.	986m	T.O.	T.O.	T.O.	T.O.	349.8m	198.1m	102m
63	nm	binutils-5279478	dwarf2.c:2378	1313.4m	692.3 m	831m	1065m	923.1m	789.4m	868.5m	429.8m	**210.4m**	245.4m
64			dwarf1.c:281	T.O.	164.3	**98.2m**	T.O.	321.5m	241.5m	268.3m	206.7m	199.3m	187m
65			elf-properties.c:51	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**853m**
66			XRef.cc:645	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**193.4m**
67			Stream.cc:2658	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**204.5m**
68	pdftotext	4.00	GfxFont.cc:1337	1345.2m	T.O.	N/A	T.O.	T.O.	1392.4m	1223.4m	533.7m	176.4m	**129.1m**
69			Stream.cc:1004	725.0m	T.O.	N/A	T.O.	T.O.	672.9m	824.5m	577.8m	429.1m	**269.2m**
70			GfxFont.cc:1643	**637.4m**	T.O.	N/A	T.O.	T.O.	782.1m	514.6m	1042.1m	T.O.	924.2m
71			jpger.c:5017	617.6m	126.7m	N/A	N/A	89.4m	673.1m	436.8m	191.7m	143.8m	**67.1m**
72			select.c:4301	T.O.	N/A	T.O.	N/A	N/A	T.O.	T.O.	T.O.	9.6m	**8.2m**
73	sqlite3	SQLite 3.8.9	func.c:1029	T.O.	1203.4m	T.O.	N/A	1136.1m	T.O.	T.O.	T.O.	T.O.	**896.4m**
74			insert.c:1498	T.O.	1366.4m	T.O.	N/A	985.4m	T.O.	T.O.	T.O.	T.O.	**857.8m**
75			vdbe.c:1984	T.O.	98.4m	89.6m	N/A	67.3m	T.O.	T.O.	56.8m	49.2m	**14.2m**
76			tiffcomposite.cpp:82	73.1m	6.4m	N/A	68.1m	15.2m	32.6m	59.3m	11.2m	5.7m	**2.5m**
77			XMPMeta-Parse.cpp:1037	126.6m	**11.3m**	N/A	168m	45.3m	83.7m	78.1m	45.7m	33.8m	16.3m
78	exiv2	0.26	XMPMeta-Parse.cpp:847	37.5m	26.4m	N/A	**21.4m**	28.3m	8.6m	13.4m	36.7m	22.6m	39.8m
79			tiffvisitor.cpp:1044	102.4m	135.2m	N/A	T.O.	**36.2m**	143.7m	111.4m	56.9m	88.3m	41.2m
80			XMPMeta-Parse.cpp:896	86.7m	6.1m	N/A	421m	67.2m	32.9m	53.8m	46.3m	2.5m	**1.7m**
81			elf.c:9509	T.O.	188.2m	782m	T.O.	623.4	T.O.	T.O.	461.7m	173.8m	**81.3m**
82			section.c:936	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	1241.9m	**823.1m**
83	objdump	binutils-2.28	bfd.c:1108	T.O.	622.4	361.4m	1288.1m	592.1m	1381.6m	983.5m	663.8m	**344.9m**	512.4m
84			bfdio.c:262	T.O.	T.O.	1123.6m	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	**792.1m**
85			stabs.c:372	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
86			rawdec.c:268	T.O.	421.4m	N/A	N/A	512m	T.O.	T.O.	T.O.	984.1m	**292.1m**
87			decode.c:557	T.O.	123.9m	N/A	N/A	135.9m	T.O.	T.O.	T.O.	397.6m	**297.4m**
88	ffmpeg	4.0.1	dump.c:632	T.O.	322.8m	N/A	N/A	N/A	T.O.	T.O.	T.O.	583.5m	**101.2m**
89			utils.c	T.O.	T.O.	N/A	N/A	T.O.	T.O.	T.O.	T.O.	1258.6m	**738.1m**
90			eatgv.c:274	T.O.	T.O.	N/A	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	**1231.3m**
91			jsrun.c:572	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
92			jsgc.c:47	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
93	mujs	1.0.2	jsdump.c:292	T.O.	782.4m	N/A	T.O.	812.3m	T.O.	T.O.	T.O.	T.O.	**582.3m**
94			jsvalue.c:362	T.O.	992.3m	N/A	T.O.	746.3m	T.O.	T.O.	T.O.	T.O.	**679.1m**
95			jsvalue.c:396	T.O.	752.9m	N/A	968.8m	611.2m	T.O.	1165.8m	951.7m	885.3m	**586.3m**
96			initcode.c:242	324.7m	**56.3m**	N/A	223.2m	128.3m	394.7m	301.4m	112.5m	64.7m	89.3m
97			pngc:410	871.9m	531.3m	N/A	681.3m	**98.4m**	692.4m	769.3m	408.1m	193.6m	364m
98	swftools	0.9.2	poly.c:137	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
99			jpeg2swf.c:257	677.8m	**257.3m**	N/A	541.1m	289.1m	677.3m	632.6m	398.3m	116.3m	301.4m
100			swfc.c:1794	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
	speedup			**36.38×**	**11.90×**	**N/A**	**37.49×**	**11.09×**	**26.68×**	**27.96×**	**8.04×**	**1.99×**	-
	mean $\hat{A}_{12}$			**0.92**	**0.78**	**N/A**	**0.89**	**0.76**	**0.83**	**0.89**	**0.71**	**0.63**	-
	mean p-values			0.001	0.021	**N/A**	0.003	0.024	0.0014	0.004	**0.031**	**0.044**	-

5.3 Exposing Vulnerabilities

To answer **RQ2**, we compared MultiGo with the baseline fuzzers in terms of their ability to expose known vulnerabilities. Building on the methodology of SelectFuzz, we utilized both the CVE-Benchmark and the Google-Fuzzer-Test-Suite to conduct our evaluation.

5.3.1 TTE results on CVE-Benchmark. We selected known vulnerabilities with CVE-IDs as targets for our testing, which is similar to SelectFuzz. The target site information and the TTE results are presented in Table 3. In summary, for most of the target sites (12/20), MultiGo outperformed all the baseline fuzzers and achieved the shortest TTE. With respect to the mean TTE of exposing vulnerabilities, MultiGo demonstrated 4.27×, 1.86×, 3.61×, 3.30×, and 3.11× speedup compared to AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL, respectively. All p-values were less than 0.05, and the mean $\hat{A}_{12}$ against AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL was 0.88, 0.67, 0.78, 0.80, and 0.75, respectively. Based on the above analysis, we can conclude that **MultiGo can expose known vulnerabilities faster than the baseline fuzzers on the CVE-Benchmark.**

TTE performance analysis of different fuzzers. We conducted experiments on the CVE-Benchmark using the Docker images provided by Beacon (commit `a09c8cb`) and the artifacts provided by DAFL. However, the experimental results we obtained differed from the experimental results presented by Beacon and DAFL. For example, we observed differences in targets #2 and #16-#18. For Beacon, since the author did not provide explicit initial seed selection when testing each program, we assume that the differences resulted from the initial seeds. As for DAFL, since we used the same initial seeds as SelectFuzz instead of the seeds provided by DAFL, there were some discrepancies between our experimental results and the results presented by DAFL. For instance, in target #1, #2 and #4, DAFL exposed vulnerabilities within 300 minutes, far surpassing the *Time-Out* results mentioned in DAFL's paper. For other targets, since we used simple files as initial seeds similar to the seeds used in DAFL, our experimental results were similar to those of DAFL.

5.3.2 TTE results on the GFTS Benchmark. We also compared MultiGo with the baseline fuzzers on the GFTS Benchmark. We used the initial seeds provided in GFTS if they were available; otherwise, we used a file containing the string "hello" as the initial seed. We set the final crashing point as the target for each vulnerability and utilized the recommended configuration options in each baseline fuzzer. When calculating the mean TTE of MultiGo and baseline fuzzers, we excluded targets that all fuzzers can reach in less than 0.01 minutes (e.g., target #10 and #16).

The detailed information on target sites and the TTE results are presented in Table 4. MultiGo has the most target sites (6) that achieve the shortest TTE compared to AFLGo (0), SelectFuzz (2), Beacon (1), WindRanger (2), and DAFL (0). In terms of mean TTE of exposing vulnerabilities, MultiGo demonstrated 2.10×, 1.68×, 78.81×, 2.05×, and 2.55× speedup compared to AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL, respectively. All p-values were less than 0.05, and the mean $\hat{A}_{12}$ against AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL was 0.79, 0.66, 0.81, 0.64, and 0.71, respectively. Based on the above analysis, we can conclude that **MultiGo can expose known vulnerabilities faster than the baseline fuzzers on the GFTS Benchmark**.

TTE performance analysis of different fuzzers. As Beacon, WindRanger, and DAFL did not provide the program compilation methods for the GFTS benchmark, we compiled them based on the compilation methods they provided in the CVE-Benchmark. However, we encountered some compilation issues. For certain programs like `openssl`, `libssh`, and `woff2`, we encountered compilation errors due to `environmental conflicts` or `target location failures`. Other runtime errors like `segmentation faults` also occurred when we attempted to launch the fuzzing campaign. Consequently, Beacon and DAFL lacked experimental results for some targets and their mean TTE was inferior to that of AFLGo. Moreover, when we used WindRanger to compile `json`, `libssh`, `pcre`, and `re2`, it failed to generate the distance information. Without the distance information, WindRanger could only test programs in an undirected manner. Additionally, the overhead introduced by strategies such

Table 3. The results of TTE on the CVE-Benchmark

No.	Program	CVE-ID	AFLGo	SelectFuzz	Beacon	WindRaner	DAFL	MultiGo
1	objdump-2.28	2017-8392	97.2m	21.1m	19.4m	78.8m	26.1m	**17.2m**
2	objdump-2.28	2017-8396	132.6m	**33.7m**	39.1m	102.5m	37.2m	42.8m
3	objdump-2.28	2017-8397	T.O.	1053.1m	T.O.	T.O.	T.O.	**561.6m**
4	objdump-2.28	2017-14940	168.8m	**85.2m**	T.O.	121.6m	282.1m	102.4m
5	cxxfilt-2.26	2016-4491	448.2m	241.3m	258.2m	298.7m	274.3m	**57.6m**
6	cxxfilt-2.26	2016-4492	10.8m	8.2m	43.6m	7.47m	6.46m	**2.67m**
7	cxxfilt-2.26	2016-6131	348.5m	211.4m	292m	318m	232.9m	**97.1m**
8	libming-4.48	2018-8807	331.6m	67.8m	267.8m	171.2m	**58.5m**	68.3m
9	libming-4.48	2018-8962	234.4m	102.6m	163.5m	121.8m	198.2	**41.2m**
10	libming-4.48	2018-11095	T.O.	523.6m	252.1m	1311m	186.4m	**109.8m**
11	libming-4.48	2018-11225	T.O.	389.4m	438m	996m	476.8m	**176.3m**
12	libpoppler.so.87	2019-10872	T.O.	**371.9m**	T.O.	T.O.	T.O.	T.O.
13	libpoppler.so.87	2019-10873	T.O.	1156.3m	T.O.	T.O.	**924.2m**	T.O.
14	libpoppler.so.87	2019-14494	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
15	xmllint-20904	2017-9047	T.O.	858.4m	T.O.	T.O.	1022.7m	**723.5m**
16	xmllint-20904	2017-9048	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
17	xmllint-20904	2017-9049	T.O.	T.O.	T.O.	T.O.	T.O.	**669.3m**
18	xmllint-20904	2017-9050	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
19	lrzip-0.631	2017-8846	348.4m	126.1m	156.2m	223.7m	T.O.	**61.3m**
20	lrzip-0.631	2018-11496	201.2m	67.1m	98.1m	169.6m	42.8m	**21.4m**
	speedup		**4.27×**	**1.86×**	**3.61×**	**3.30×**	**3.11×**	-
	mean $\hat{A}_{12}$		**0.88**	**0.67**	**0.78**	**0.80**	**0.75**	-
	mean p-values		0.007	0.043	0.015	0.011	0.023	-

Table 4. The results of TTE on the Google-Fuzzer-Test-Suite

No	Program	Vuln. Code	AFLGo	SelectFuzz	Beacon	WindRanger	DAFL	MultiGo
1	boringssl	asn1_lib.c:459	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
2	guetzli	output_image.cc:398	36.7m	12.6m	31.8m	34.2m	22.1m	**10.2m**
3	harfbuzz	hb-buffer.cc:419	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
4	json	fuzzer-parse_json.cpp:50	5.4m	5.6m	**3.2m**	7.2m	7.2m	3.5m
5	lcms	cmsintrp.c:642	552.6m	456.2m	T.O.	412.6m	T.O.	**113.8m**
6	libarchive	archive_read_support_format_warc.c:537	T.O.	**628.7m**	T.O.	T.O.	826.1m	T.O.
7	libssh	messages.c:1001	19.2m	7.2m	23.6m	65.1m	N/A	**16.6m**
8	libxml2	parser.c:10666	6.6m	8.9m	T.O.	4.1m	1.4m	**2.1m**
9	libxml2	dict.c:489	26.9m	23.4m	T.O.	**11.3m**	16.9m	14.2m
10	openssl-1.0.1f	t1_lib.c:2586	<0.01m	<0.01m	N/A	<0.01m	N/A	<0.01m
11	openssl-1.0.2d	target.cc:145	7.2m	11.3m	N/A	6.7m	N/A	**6.4m**
12	pcre	pcre2_match.c:1426	42.6m	**33.7m**	48.1m	39.2m	N/A	38.2m
13	re2	nfa.cc:532	T.O.	858.4m	T.O.	T.O.	682.7m	**249.0m**
14	vorbis	codebook.c:407	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
15	woff2	woff2_dec.cc:1274	439.2m	424.6	N/A	**421.8m**	N/A	442.5m
16	c-ares	ares_create_query.c:196	<0.01m	<0.01m	<0.01m	<0.01m	<0.01m	<0.01m
	speedup		**2.10×**	**1.68×**	**78.81×**	**2.05×**	**2.55×**	-
	mean $\hat{A}_{12}$		**0.79**	**0.66**	**0.81**	**0.64**	**0.71**	-
	mean p-values		0.016	0.044	0.027	0.018	0.032	-

as extra basic block recognition, distance calculation, and power schedule optimization led to longer TTEs for WindRanger compared to the TTEs of AFLGo.

We further studied why MultiGo cannot outperform other baseline fuzzers in some cases. Taking target #6 as an example, as Figure 5 shows, there is only one path leading to the target function `xstrpisotime()`, and there are no path constraints along this execution path. As a result, MultiGo's multi-path optimization was ineffective in this particular scenario. In contrast, SelectFuzz and DAFL enhance target-reaching efficiency by selectively instrumenting and excluding irrelevant code executions. Therefore, SelectFuzz and DAFL outperformed WindRanger and MultiGo on target #6.

5.4 Impact of Optimizations on the Overall Performance

To answer **RQ3**, we conducted incremental experiments to evaluate the effects of the two optimizations on MultiGo's overall performance. Since MultiGo's fuzzing component is based on AFLGo and its symbolic execution component is based on SymCC, we also used SymGo as a baseline tool.

The fuzzing component of SymGo follows AFLGo's strategies, while the combination of the fuzzing and symbolic execution components adopts SymCC's strategy (where the fuzzer only provides "interesting seeds" to the symbolic executor, and SymCC sorts these seeds based on factors like improving code coverage). Besides, **we replaced the distance fitness metric in SymGo with the path difficulty metric, while retaining SymCC's strategy, forming *SymGo-PD***. The calculation and update of the path difficulty metric in SymGo-PD are consistent with those in MultiGo, ensuring a fair comparison. Finally, we integrated the difficulty-guided path scheduling into SymGo-PD and removed SymCC's strategy to form the full version of MultiGo. The comparison of different configurations (e.g., fitness metrics, fuzzing strategies, and hybrid fuzzing strategies) across different fuzzers is outlined in Table 1.

In addition to the incremental experiments, to provide statistical evidence supporting the novelty of the path difficulty metric in MultiGo over probability-based fitness metrics used in previous works (e.g., DigFuzz, K-scheduler, and Marco), we conducted an ablation experiment. Since DigFuzz is also a hybrid fuzzer that employs a Markov-chain-based probabilistic model to evaluate the difficulty of covering different paths, we selected it as a representative for similar works, including K-scheduler and Marco, for comparison. Specifically, **we replaced the path difficulty metric in MultiGo with DigFuzz's Markov-chain-based probabilistic method, forming a new directed hybrid fuzzer, *MultiGo-PB***. We compared the TTR results of MultiGo-PB and MultiGo on UniBench and analyzed intermediate data to evaluate their effectiveness in reaching targets. For the incremental and ablation experiments, the configurations and target sites remained the same as in §5.2.

Detailed results are listed in Table 2. Firstly, in the incremental experiments, SymGo-PD (69) and MultiGo (92) can reach more target sites than SymGo (51). Moreover, MultiGo outperforms SymGo and SymGo-PD by 27.96× and 8.04×, respectively, in terms of the mean TTR of reaching the target sites. These results demonstrate that **each**

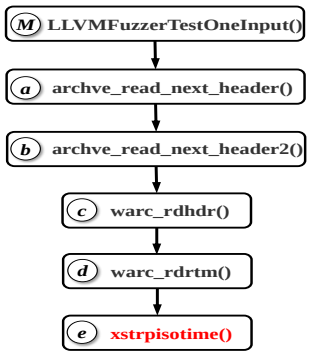

Fig. 5. The control-flow graph of the program with target #6 in the GFTS Benchmark.

Table 5. Intermediate data analysis results

Fuzzers	S_r	S_e	P_r	Ft
SymGo	2834	19	1.06	1708
SymGo-PD	8691	39	1.98	1593
MultiGo-PB	10389	209	2.37	1189
MultiGo	13739	254	2.83	1421

optimization has a significant impact on reducing TTR. Secondly, in the ablation experiments, MultiGo (92) can reach more target sites than MultiGo-PB (78). Moreover, MultiGo outperforms MultiGo-PB by 1.99× in the mean TTR of reaching the target sites and the mean p-value is less than 0.05. Furthermore, as shown in Table 5, MultiGo covered an average of 2.83 paths that reach target sites while MultiGo-PB covered 2.37 on average. Based on the TTR results and the diversity of paths that reach target sites, the ablation experiments provide statistical evidence that the path difficulty metric is more effective than the probability-based method used in DigFuzz in both accelerating target reaching and enhancing path diversity toward targets. This demonstrates the novelty of the path difficulty metric in MultiGo.

5.5 Intermediate Data Analysis

To validate the effectiveness of MultiGo's strategies in multi-path situations and to provide a more intuitive illustration of the effects of different optimizations, we propose four metrics to analyze the intermediate experimental data when testing programs on the UniBench benchmark. In Table 5, **the value of each metric represents the average result obtained from testing all programs in UniBench**.

(1) **Number of target-reachable paths** (i.e., S_r). We use the number of seeds that cover different target-reachable paths to represent the number of target-reachable paths. According to Table 5, we observe that the value of S_r increases significantly from 2834 in SymGo to 13739 in MultiGo. This indicates that the implementation of each optimization strategy in MultiGo, including the path difficulty fitness metric and the difficulty-guided path scheduling, effectively enhances the fuzzer's ability to cover more target-reachable paths. In conclusion, these strategies significantly improve MultiGo's performance in multi-path situations.

(2) **Number of seeds generated by the symbolic executor** (i.e., S_e). The symbolic executor produces seeds that enable the fuzzer to navigate complex constraints, enhancing the efficiency of directed hybrid fuzzing. We assess the impact of difficulty-guided path scheduling by measuring the number of seeds the symbolic executor generates. As shown in Table 5, MultiGo achieves a significantly higher S_e value (254) compared to SymGo-PD (39), indicating that the symbolic executor in MultiGo operates more efficiently. These results validate that difficulty-guided path scheduling enhances symbolic execution's effectiveness, supporting fuzzing in navigating complex paths and overcoming some of its inherent limitations. Due to limited symbolic execution optimization and constraint-solving capabilities, symbolic execution faces scalability challenges in large programs. Inefficient path scheduling can allocate low-reward paths, limiting symbolic execution's ability to aid fuzzing in deeper path exploration. For instance, symbolic execution generated only 19 in SymGo and 39 in SymGo-PD, respectively. By employing difficulty-based path scheduling with CMAB, MultiGo significantly improves symbolic execution efficiency, increasing the number of generated seeds. Furthermore, MultiGo's fuzzing component further mutated S_e, discovering 3,192 new target-reachable paths on average, greatly enhancing comprehensive target testing. Thus, the difficulty-guided path scheduling with CMAB improves symbolic execution's scalability, enabling it to serve as an effective component for more efficient target testing in MultiGo.

(3) **Number of paths that reached target sites** (i.e., P_r). In multi-path situations, our goal is not only to reach targets more quickly through optimal paths but also to reach target sites through a greater variety of paths for thorough testing. Thus, P_r is a crucial metric for evaluating the fuzzer's performance in multi-path scenarios. By observing the values of P_r among SymGo, SymGo-PD, and MultiGo, we find that the value of P_r increases

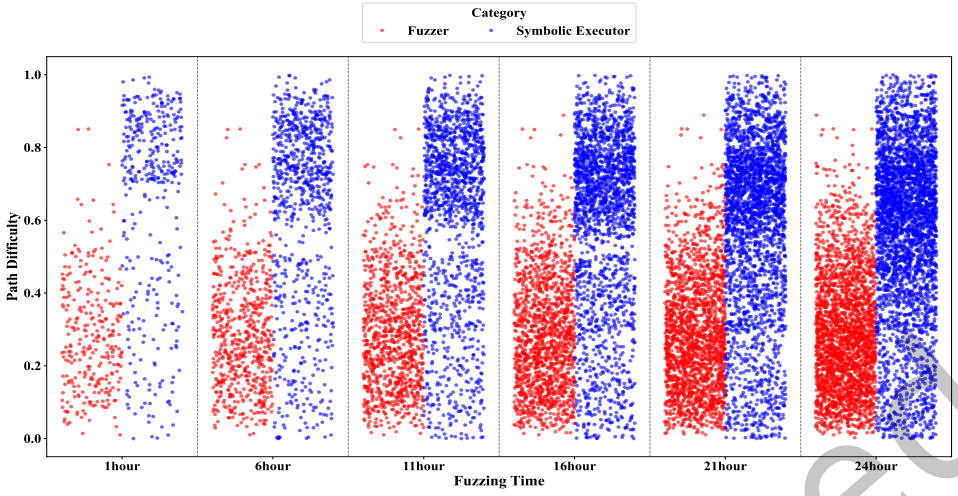

Fig. 6. Path difficulty distribution over time

from 1.06 to 2.83 per target site, nearly tripling. This indicates that the optimizations in MultiGo significantly enhance its ability to cover more paths that reach target sites in multi-path situations. Consequently, MultiGo can perform more thorough testing on target sites.

(4) **The fuzzing throughput of fuzzers** (i.e., Ft). To evaluate the impact of path difficulty and CMAB model on fuzzing performance and scalability, we measured the fuzzing throughput of the fuzzers. For a single program, the fuzzing throughput is calculated as the total number of seed executions divided by 24 hours. We then computed the average fuzzing throughput across all programs and used this as the overall fuzzing throughput, denoting it as Ft. As shown in Table 5, the Ft of SymGo-PD and MultiGo decreased by only 6.73% and 16.80%, respectively, compared to SymGo. This indicates that the calculation and updating of the path difficulty metric resulted in only a 6.73% decrease in Ft, while the introduction of difficulty-guided path scheduling led to a 10.07% decrease. Therefore, we can conclude that the incurred overhead is acceptable and does not affect scalability, even for large-scale programs.

5.6 Visible Reflection of MultiGo's Effectiveness

To visually reflect the effectiveness of MultiGo in multi-path scenarios, we utilize Figure 6 to illustrate distinct paths processed by the fuzzer and symbolic executor during the fuzzing process. The X-axis represents the time points (1, 6, 11, 16, 21, and 24 hours) at which paths were recorded. The Y-axis represents the normalized value

Table 6. The programs tested for detecting new bugs

No	Program	Version
1	fig2dev	3.2.7a
2	gifsicle	1.90
3	jbig2dec	0.16
4	freetype	2.10.0
5	ImageMagick	7.0.8.6
6	libjpeg-turbo	1.5.1
7	libpng	1.6.37
8	pdftops	Libpoppler 0.74
9	GraphicsM	1.3.28

Table 7. New vulnerabilities detected by MultiGo

No	Program	Bug Location	Bug Type	CNNVD-ID
1	gdk-pixbuf-2.31	io-pcx.c:450	heap-buffer-overflow	2023-15365826
2	gdk-pixbuf-2.31	gdk-pixdata.c:142	heap-buffer-overflow	2023-77272850
3	gdk-pixbuf-2.31	io-ico.c:710	heap-buffer-overflow	2023-18312486
4	gdk-pixbuf-2.31	io-ico.c:681	heap-buffer-overflow	2023-94830436
5	gdk-pixbuf-2.31	io-tga.c:401	out-of-bounds write	APPLYING
6	jhead-3.00	exif.c:669	out-of-bounds read	2023-08835347
7	mp3gain-1.5.2	common.c:328	memcpy-param-overlap	APPLYING
8	xpdf-4.00	Stream.cc:2919	heap-buffer-overflow	APPLYING
9	xpdf-4.00	Lexer.cc:489	global-buffer-overflow	APPLYING
10	xpdf-4.00	TextOutputDev.cc:3065	stack-overflow	APPLYING
11	xpdf-4.00	Dict.cc:98	heap-buffer-overflow	APPLYING
12	flvmeta-1.2.1	dump_xml.c:151	out-of-bound read	2023-32614978
13	fig2dev	free.c:152	Null pointer dereference	2023-52914469
14	fig2dev	free.c:173	Null pointer dereference	2023-65505499

of path difficulty for each path. The red and blue dots represent the paths selected for fuzzing and symbolic execution, respectively.

It can be observed that number of dots increases as new paths are discovered. Besides, the blue dots are concentrated in the upper part of the figure while the red dots are concentrated in the lower part. This indicates that the fuzzer tends to select easier paths while the symbolic executor handles more difficult paths. This demonstrates that MultiGo reasonably allocates resources between the fuzzer and symbolic executor based on path difficulty in the multi-path situation. Furthermore, as fuzzing proceeds, the path difficulty of the paths handled by the symbolic executor decreases gradually. This is because the paths with high difficulty can still produce new paths, which are also likely to be difficult. This makes it hard for MultiGo to discover new branches leading to targets along these paths. Consequently, the priority of these difficult paths selected by the symbolic executor gradually decreases.

5.7 Discovering New Vulnerabilities

To answer **RQ5**, we used 9 real-world programs in addition to UniBench to discover new vulnerabilities. We used a static analyzer (e.g., IKOS [2]) to label potential vulnerabilities as the targets and run MultiGo for 24 hours to detect new vulnerabilities. Finally, MultiGo discovered 14 undisclosed vulnerabilities (listed in Table 7) from 6 real-world programs. The new vulnerabilities involve heap-buffer-overflow, out-of-bounds read/write, stack-overflow, Null pointer dereference, and memcpy-param-overlap. Among the 14 new vulnerabilities discovered, 8 have already been assigned CNNVD-IDs, while the remaining 6 are currently in the process of being assigned CNNVD-IDs. CNNVD extensively collects vulnerability information similar to CVE, and it is widely accepted by researchers.

6 DISCUSSION

Multi-path optimization in MultiGo. The multi-path scenarios are prevalent in directed fuzzing, with our observations indicating that 94% (128 out of 136) of the evaluated target sites (§5.5) have multiple paths leading to them. For example, target #81 in UniBench has the highest number of paths (16) toward its target. Unlike existing DGF techniques, MultiGo utilizes path difficulty to differentiate between optimal and challenging paths. Based on the difficulty-based fitness metrics, we introduce the CMAB model and the concept of fuzzing context to create a customized path scheduling scheme that better integrates DGF and symbolic execution, accelerating target

reaching while enhancing coverage of potential vulnerability paths. This approach enables MultiGo to cover more paths for comprehensive target testing (shown in Figure 6). Compared to other directed fuzzing techniques, MultiGo's strategy achieves higher accuracy and efficiency in multi-path directed fuzzing.

Accuracy of path-difficulty-based fitness metric. In this paper, we use path difficulty to evaluate the complexity of path constraints by focusing on the frequency of path executions. We excluded factors such as the number of variables and the complexity of operators, positing that the frequency of path executions objectively measures the difficulty a fuzzer encounters in covering various paths. Our approach addresses limitations in traditional probability-based fitness metrics, such as those used by DigFuzz [30], AFLFast [6], and HyperGo [19], which are often limited by issues like variable independence and the bias towards shorter paths. In contrast, the path-difficulty-based fitness metric leverages the Poisson distribution based on statistical block frequency, thereby avoiding the use of branch and path probabilities and mitigating the biases present in traditional methods. Using the Poisson distribution helps resolve issues of insufficient hits on certain paths, providing a more accurate and objective assessment of path difficulty. Additionally, we utilize the geometric mean, which is more sensitive to smaller values, instead of the arithmetic average to calculate path difficulty. This sensitivity ensures a more nuanced evaluation of path difficulty, forming a more precise and reliable fitness metric.

7 THREATS TO VALIDITY

False positives and false negatives of MultiGo. Although MultiGo designs path difficulty based on basic block execution frequency and static fitness metrics, which improves path evaluation accuracy, it still exhibits false positives and false negatives. False positives arise when certain seeds are extensively fuzzed due to a limited initial seed corpus, inflating execution counts for their associated basic blocks. A false positive occurs when a path is initially evaluated as simple ($normalized\ path\ difficulty >= 0.5$) but is later reclassified as complex, or vice versa. We observed that in the first hour during fuzzing, 21.38% of paths experienced false positives due to changes in their path difficulty evaluation. However, as fuzzing progressed, this rate declined to 9.2% by the 23rd hour. Conversely, false negatives occur when basic blocks distant from the initial seed corpus remain unexplored, preventing their path difficulty from being evaluated. This is reflected in the false negative rate, which captures the proportion of unexplored target-reachable basic blocks. Our data shows a significant reduction in this rate, from 81.6% in the first hour to 36.2% by the 23rd hour, highlighting MultiGo's improved exploration efficiency. Therefore, as fuzzing progresses and more basic blocks and paths are explored, MultiGo gradually reduces the false positives and false negatives, minimizing their impact on reachability guidance.

Limitations of symbolic execution in MultiGo. Although the optimized path scheduling scheme with CMAB significantly improves symbolic execution's efficiency and scalability, MultiGo's symbolic execution component remains limited when handling complex path constraints. For instance, in UniBench target #45 in tcpdump, symbolic execution failed to symbolize and solve two key path constraints (*if (GET_BE_U_2(ptr) & L2TP_FLAG_TYPE)* and *if (attr_type) == 32*), preventing valid input generation and reaching targets. Similar challenges may arise in other larger-scale programs with complex execution paths. However, it is worth noting that in most target evaluations, MultiGo's symbolic execution successfully generates an average of 254 seeds, outperforming baseline fuzzers. This demonstrates its effectiveness in addressing the multi-path issue.

Impact of targets' characteristics on MultiGo's performance. MultiGo's performance in reaching targets and exposing vulnerabilities is also affected by target characteristics. For accelerating target reaching, MultiGo's advantage depends on whether its identified optimal paths are much easier to traverse than those identified by other fuzzers. For instance, in target #27 on UniBench, when MultiGo's optimal paths are significantly easier, it achieves substantial speedups. Conversely, in targets #90 and #100 on UniBench, if the optimal paths remain challenging, the TTR improvement may be limited, or MultiGo may fail to reach the targets within 24 hours. For accelerating vulnerability exposure, MultiGo, like other directed fuzzers (e.g., AFLGo, DAFL, Beacon, SelectFuzz),

does not pre-analyze paths or prioritize them based on vulnerability likelihood. Instead, it rapidly generates target-reaching inputs while exploring more paths toward targets to expose bugs. If vulnerabilities are located on optimal paths, MultiGo can expose them quickly. However, if they exist on non-optimal paths, MultiGo may require more time to cover the non-optimal paths.

8 RELATED WORK

Directed Grey-box Fuzzing. AFLGo is the first directed grey-box fuzzer. It calculates the distances between the seeds and pre-defined targets to prioritize the seeds closer to the targets, which casts reachability as an optimization problem to minimize the distance between the seeds and their targets. Based on AFLGo's idea, Hawkeye [7] proposes the concept of trace similarity and adjusts its seed prioritization, power scheduling, and mutation strategies to enhance directedness. However, Hawkeye suffers the same issues as those of AFLGo when encountering complex path constraints. Even when assigning more energy to closer seeds, it is difficult for these fuzzers to satisfy complex path constraints and cover the paths toward target sites. Some directed grey-box fuzzers, such as LOLLY [18], Berry [17], UAFL [29], and CAFL [14], propose new fitness metrics, such as a new fitness metric based on sequence similarity, to enhance directedness and detect hard-to-manifest vulnerabilities. These methods are derived from the analysis of program characteristics or the root causes of different vulnerabilities. Thus, for some specific programs or fuzzing processes, these new fitness metrics may be inaccurate and have a negative effect, which has been discussed in Section 2.2. Other directed grey-box fuzzers use data flow information and data condition information to enhance directedness. WindRanger [9] uses the deviation basic blocks (DBBs) and the data flow information for seed distance calculation, seed mutation, seed prioritization, and power schedule. Beacon [11] leverages a provable path-pruning method to reduce the exploration of infeasible paths. However, due to the limitations of static analysis, Beacon's analysis of infeasible paths (e.g., Beacon cannot recognize indirect calls) may be inaccurate. This can result in the incorrect pruning of some feasible paths, and consequently slowing down the process of reaching target sites. Besides, FuzzGuard [38] uses the deep neural network to extract the features of reachable seeds and filter out the unreachable seeds to improve efficiency. To search for inputs that can reach the target sites, MC^2 [26] designs an asymptotically optimal randomized directed grey-box fuzzer that has logarithmic expected execution complexity in the number of possible inputs. However, DGF still struggles to penetrate the hard-to-satisfy path constraints. MultiGo selects the better paths that have fewer hard-to-satisfy path constraints and utilizes symbolic execution to assist DGF in passing through such path constraints.

Directed Hybrid Fuzzing. Directed hybrid fuzzing uses the heuristic strategies in hybrid fuzzing to gain directedness. Directed hybrid fuzzers achieve directedness by prioritizing the symbolic execution of reachable seeds or closer seeds. Hydiff [13], SAVIOR [8] and Badger [23] prioritize the seeds that may cause the specific program bug locations as the target sites, and then prioritize symbolic execution of the seeds reachable from more target sites. 1dVul[24] and Berry [17] combine the precision of DSE and the scalability of DGF to mitigate their individual weaknesses. However, modern directed hybrid fuzzers suffer from the limitation of symbolic execution. Since the symbolic executor may fail to solve many unexplored branches or attempt to solve unreachable branches, such ineffective constraint solving will have a negative impact on the directedness of directed hybrid fuzzing. Thus, MultiGo introduces difficulty-guided path scheduling with the CMAB model to optimize path scheduling in both fuzzing and symbolic execution while also improving the constraint-solving strategy of symbolic execution. This enhancement increases the efficiency of both fuzzing and symbolic execution in reaching targets and exploring more paths.

9 CONCLUSION

In this paper, we propose MultiGo, a directed hybrid fuzzer specifically designed for multi-path scenarios in directed fuzzing. MultiGo utilizes the path difficulty as a fitness metric to distinguish between optimal and challenging paths. By leveraging CMAB, MultiGo efficiently handles complex fuzzing context with low overhead and optimizes path scheduling in both fuzzing and symbolic execution components, enhancing the efficiency of both DGF and symbolic execution in directed hybrid fuzzing. This approach enables MultiGo to accelerate target reaching, while enhancing path diversity toward targets to conduct thorough target testing. MultiGo is evaluated on 136 target sites of 41 real-world programs across 3 benchmarks. The experimental results show that MultiGo outperforms the state-of-the-art directed fuzzers (AFLGo, SelectFuzz, Beacon, WindRanger, and DAFL) and hybrid fuzzers (SymCC and SymGo) in reaching target sites and exposing known vulnerabilities. MultiGo also discovered 14 undisclosed 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] 2023. *Google's fuzzer-test-suite.* https://github.com/google/fuzzer-test-suite
[2] 2023. *IKOS.* https://github.com/NASA-SW-VnV/ikos
[3] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire. 2002. The nonstochastic multiarmed bandit problem. *SIAM J. Comput.* 32, 1 (2002), 48–77.
[4] Marcel BoHme, Van Thuan Pham, Manh Dung Nguyen, and Abhik Roychoudhury. 2017. Directed Greybox Fuzzing. In *Acm Sigsac Conference on Computer & Communications Security*. 2329–2344.
[5] Marcel Böhme. 2023. *Directed Greybox Fuzzing with AFL.* https://github.com/aflgo/aflgo
[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2019. Coverage-Based Greybox Fuzzing as Markov Chain. *IEEE Transactions on Software Engineering* 45, 5 (2019), 489–506. https://doi.org/10.1109/TSE.2017.2785841
[7] 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, CCS 2018, Toronto, ON, Canada, October 15-19, 2018.* ACM, 2095–2108. https://doi.org/10.1145/3243734.3243849
[8] Yaohui Chen, Peng Li, Jun Xu, Shengjian Guo, Rundong Zhou, Yulong Zhang, Tao Wei, and Long Lu. 2020. SAVIOR: Towards Bug-Driven Hybrid Testing. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 1580–1596. https://doi.org/10.1109/SP40000.2020.00002
[9] Zhengjie Du, Yuekang Li, Yang Liu, and Bing Mao. 2022. WindRanger: A Directed Greybox Fuzzer driven by DeviationBasic Blocks. In *ICSE '22: 44st International Conference on Software Engineering*. ACM.
[10] Jie Hu, Yue Duan, and Heng Yin. 2024. Marco: A Stochastic Asynchronous Concolic Explorer. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 59, 12 pages. https://doi.org/10.1145/3597503.3623301
[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 *The 43rd IEEE Symposium on Security and Privacy(S&P'22)*.
[12] Tae Eun Kim, Jaeseung Choi, Kihong Heo, and Sang Kil Cha. 2023. DAFL: Directed Grey-Box Fuzzing Guided by Data Dependency. In *Proceedings of the 32nd USENIX Conference on Security Symposium* (Anaheim, CA, USA) *(SEC '23)*. USENIX Association, USA, Article 276, 18 pages.
[13] Tor Lattimore. 2016. Regret Analysis of the Finite-Horizon Gittins Index Strategy for Multi-Armed Bandits. In *Proceedings of the 29th Conference on Learning Theory, COLT 2016, New York, USA, June 23-26, 2016 (JMLR Workshop and Conference Proceedings, Vol. 49)*, Vitaly Feldman, Alexander Rakhlin, and Ohad Shamir (Eds.). JMLR.org, 1214–1245. http://proceedings.mlr.press/v49/lattimore16.html

[14] Gwangmu Lee, Woochul Shim, and Byoungyoung Lee. 2021. Constraint-guided Directed Greybox Fuzzing. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu

[15] Penghui Li, Wei Meng, and Chao Zhang. 2024. SDFuzz: Target States Driven Directed Fuzzing. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, Davide Balzarotti and Wenyuan Xu (Eds.). USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/li-penghui

[16] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. 2021. 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, 2777–2794. https://www.usenix.org/conference/usenixsecurity21/presentation/li-yuwei

[17] Hongliang Liang, Lin Jiang, Lu Ai, and Jinyi Wei. 2020. Sequence Directed Hybrid Fuzzing. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*. IEEE, 127–137. https://doi.org/10.1109/SANER48275.2020.9054807

[18] Hongliang Liang, Yini Zhang, Yue Yu, Zhuosi Xie, and Lin Jiang. 2019. Sequence Coverage Directed Greybox Fuzzing. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 249–259. https://doi.org/10.1109/ICPC.2019.00044

[19] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Kai Lu, and Gen Zhang. 2024. HyperGo: Probability-based directed hybrid fuzzing. *Computers & Security* 142 (2024), 103851. https://doi.org/10.1016/j.cose.2024.103851

[20] Peihong Lin, Pengfei Wang, Xu Zhou, Wei Xie, Gen Zhang, and Kai Lu. [n. d.]. 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. https://www.ndss-symposium.org/wp-content/uploads/2024-514-paper.pdf

[21] 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)*.

[22] 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, San Sebastian, Spain, October 14-15, 2020*. USENIX Association, 47–62. https://www.usenix.org/conference/raid2020/presentation/nguyen

[23] Yannic Noller, Rody Kersten, and Corina S. Pasareanu. 2019. Badger: Complexity Analysis with Fuzzing and Symbolic Execution. In *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019 (LNI, Vol. P-292)*, Steffen Becker, Ivan Bogicevic, Georg Herzwurm, and Stefan Wagner (Eds.). GI, 65–66. https://doi.org/10.18420/se2019-16

[24] Jiaqi Peng, Feng Li, Bingchang Liu, Lili Xu, Binghong Liu, Kai Chen, and Wei Huo. 2019. 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, 605–616. https://doi.org/10.1109/DSN.2019.00066

[25] Sebastian Poeplau and Aurélien Francillon. 2020. Symbolic execution with SymCC: Don't interpret, compile!. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 181–198. https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[26] Abhishek Shah, Dongdong She, Samanway Sadhu, Krish Singal, Peter Coffman, and Suman Jana. 2022. MC2: Rigorous and Efficient Directed Greybox Fuzzing *(CCS '22)*. Los Angeles, CA, USA. https://doi.org/10.1145/3548606.3560648

[27] Dongdong She, Abhishek Shah, and Suman Jana. 2022. Effective Seed Scheduling for Fuzzing with Graph Centrality Analysis. In *2022 IEEE Symposium on Security and Privacy (SP)*. 2194–2211. https://doi.org/10.1109/SP46214.2022.9833761

[28] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. 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, 999–1010. https://doi.org/10.1145/3377811.3380386

[29] Haijun Wang, Xiaofei Xie, Yi Li, Cheng Wen, Yuekang Li, Yang Liu, Shengchao Qin, Hongxu Chen, and Yulei Sui. 2020. 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, 999–1010. https://doi.org/10.1145/3377811.3380386

[30] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards optimal concolic testing. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. ACM, 291–302. https://doi.org/10.1145/3180155.3180177

[31] Y. Wang, X. Jia, Y. Liu, K. Zeng, and P. Su. 2020. Not All Coverage Measurements Are Equal: Fuzzing by Coverage Accounting for Input Prioritization. In *Network and Distributed System Security Symposium*.

[32] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. 2020. MemLock: memory usage guided fuzzing. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*. ACM, 765–777. https://doi.org/10.1145/3377811.3380396

[33] Songtao Yang, Yubo He, Kaixiang Chen, Zheyu Ma, Xiapu Luo, Yong Xie, Jianjun Chen, and Chao Zhang. 2023. 1dFuzz: Reproduce 1-Day Vulnerabilities with Directed Differential Fuzzing *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 867–879. https://doi.org/10.1145/3597926.3598102

[34] Tai Yue, Pengfei Wang, Yong Tang, Enze Wang, Bo Yu, Kai Lu, and Xu Zhou. 2020. EcoFuzz: Adaptive Energy-Saving Greybox Fuzzing as a Variant of the Adversarial Multi-Armed Bandit. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue

[35] Y. Zhang, Y. Liu, J. Xu, and Y. Wang. 2024. Predecessor-aware Directed Greybox Fuzzing. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, Los Alamitos, CA, USA, 40–40. https://doi.org/10.1109/SP54263.2024.00040

[36] Lei Zhao, Yue Duan, Heng Yin, and Jifeng Xuan. 2019. Send Hardest Problems My Way: Probabilistic Path Prioritization for Hybrid Fuzzing. In *26th Annual Network and Distributed System Security Symposium, NDSS 2019, San Diego, California, USA, February 24-27, 2019*. The Internet Society. https://www.ndss-symposium.org/ndss-paper/send-hardest-problems-my-way-probabilistic-path-prioritization-for-hybrid-fuzzing/

[37] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. 2023. FISHFUZZ: Catch Deeper Bugs by Throwing Larger Nets. In *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Anaheim, CA, 1343–1360. https://www.usenix.org/conference/usenixsecurity23/presentation/zheng

[38] 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)*. USENIX Association, 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong