Computers & Security 142 (2024) 103851

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

HyperGo: Probability-based directed hybrid fuzzing

Peihong Lin, Pengfei Wang*, Xu Zhou, Wei Xie, Kai Lu, Gen Zhang

National University of Defense Technology, Kaifu Qu, Changsha, China

ARTICLE INFO

Keywords:
Directed greybox fuzzing
Symbolic execution
Hybrid fuzzing
Software security

ABSTRACT

Directed grey-box fuzzing (DGF) is a target-guided fuzzing intended for testing specific targets (e.g., the potential buggy code). Despite numerous techniques proposed to enhance directedness, the existing DGF techniques still face challenges, such as taking into account the difficulty of reaching different basic blocks when designing the fitness metric, and promoting the effectiveness of symbolic execution (SE) when solving the complex constraints in the path to the target. In this paper, we propose a directed hybrid fuzzer called HyperGo. To address the challenges, we introduce the concept of path probability and combine the probability with distance to form an adaptive fitness metric called *probability-based distance*. By combining the two factors, probability-based distance can adaptively guide DGF toward paths that are closer to the target and have more easy-to-satisfy path constraints. Then, we put forward an Optimized Symbolic Execution Complementary (OSEC) scheme to combine DGF and SE in a complementary manner. The OSEC would prune the unreachable branches and unsolvable branches, and prioritize symbolic execution of the seeds whose paths are closer to the target and have more branches that are difficult to be covered by DGF. We evaluated HyperGo on 2 benchmarks consisting of 25 programs with a total of 120 target sites. The experimental results show that HyperGo achieves 37.75×, 29.11×, 23.34×, 95.61× and 143.22× speedup compared to AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan, respectively in reaching target sites, and 3.44×, 3.63×, 4.10×, 3.26×, and 3.00× speedup in exposing known vulnerabilities. Moreover, HyperGo discovered 10 undisclosed vulnerabilities from 5 real-world programs.

1. Introduction

Grey-box fuzzing has been a scalable and effective approach to discovering vulnerabilities in software in recent years (Böhme et al., 2016; Chen et al., 2019, 2020a; Arshad et al., 2020). Based on the feedback information from the execution of the program under test (PUT), grey-box fuzzers utilize an evolutionary algorithm to generate specific inputs that can cause erroneous runtime behavior (e.g., memory corruptions or data abort) of PUT. Most existing fuzzers are coverage-guided (CGF) (lcamtuf, 2023; Chen and Chen, 2018; Gan et al., 2020; Lemieux and Sen, 2018) as they focus on improving the code coverage to test the deeper level of code. However, not all parts of the code in PUT are equally important because the majority of the code are safe and only a small portion has vulnerabilities. For example, according to Shin and Williams (2013), only 3% of the source code files in Mozilla Firefox have vulnerabilities. Thus, researchers aim to focus on strengthening the tests for the vulnerable parts of the code. To achieve directedness, the originally directed fuzzers were based on symbolic execution (SE) (Ganesh et al., 2009; Ma et al., 2011; Yang et al., 2011; Marinescu and Cadar, 2013), which uses program analysis and constraint solving to generate inputs that exercise different program paths. Such directed fuzzers cast the reachability problem as an iterative constraint satisfaction problem. However, since directed symbolic execution relies on heavyweight program analysis and constraint solving, it suffers from scalability and compatibility limitations (Yun et al., 2018).

In 2017, a directed grey-box fuzzer AFLGo (BoHme et al., 2017) was proposed. It leverages lightweight compile-time instrumentation to drive the fuzzing toward a set of pre-defined target locations. Different from CGF which strives to increase the code coverage, DGF intends to reach and test the target sites (e.g., the potential buggy code). A fitness metric is a criteria used to evaluate the performance of a solution or an entity within a specific context. In the context of optimization algorithms, such as genetic algorithms or evolutionary algorithms, a fitness metric is often used to quantify how well a potential solution solves a given problem or meets certain objectives. Based on the call graph and control-flow graph information of the PUT, DGF uses the distance between inputs and target sites as the fitness metric to assist seed selection and seed energy allocation. Thus, DGF can prioritize the seeds that

* Corresponding author.
E-mail addresses: phlin22@nudt.edu.cn (P. Lin), pfwang@nudt.edu.cn (P. Wang), zhouxu@nudt.edu.cn (X. Zhou), xiewei@nudt.edu.cn (W. Xie), kailu@nudt.edu.cn (K. Lu), zhanggen@nudt.edu.cn (G. Zhang).

https://doi.org/10.1016/j.cose.2024.103851
Received 1 January 2024; Received in revised form 16 March 2024; Accepted 12 April 2024
Available online 18 April 2024
0167-4048/© 2024 Elsevier Ltd. All rights reserved.

are more likely to reach the targets (i.e., optimal seeds), which makes DGF outperforms CGF in specific scenarios, such as patch testing (Peng et al., 2019), bug reproduction (Kim and Yun, 2019; Wang et al., 2020b; Nguyen et al., 2020), and potential buggy code verification (Wen et al., 2020; Wang et al., 2020a).

To accelerate reaching targets and exposing vulnerabilities, the state-of-the-art DGF techniques proposed in recent years have employed various methods to enhance directedness. For instance, some DGF techniques redefine the fitness metric based on trace similarity (Hawkeye (Chen et al., 2018)), data-flow graph (CAFL (Lee et al., 2021), WindRanger (Du et al., 2022)), and sequence coverage (LOLLY (Liang et al., 2019)) while some other DGF techniques prune infeasible paths (BEACON (Huang et al., 2022)), use the number of oracle queries required by a fuzzing algorithm to find a target-reaching input (MC^2 (Shah et al., 2022)) and utilize symbolic execution to penetrate the complex path constraints toward targets, namely directed hybrid fuzzing (Chen et al., 2020b; Noller et al., 2020, 2019). However, despite these achievements, the existing DGF techniques still face two challenges.

Challenge 1: Taking into account the difficulty of reaching different basic blocks to design a more effective fitness metric. Following AFLGo, most of the state-of-the-art DGF techniques proposed new fitness metrics and methods based on the knowledge of program analysis (e.g., DBB-distance in WindRanger and constraint-distance in CAFL). Although the methods can be beneficial in testing certain programs, they may be inaccurate while some specific programs do not meet their assumptions (e.g., WindRanger assumes that the complexity of path constraints is related to the number of corresponding seed bytes). During the fuzzing process, reaching different basic blocks has different probabilities since the path constraints are not all equally satisfied. It is challenging for the fuzzer to reach target sites through those basic blocks that are difficult to cover. Thus, it is a challenge to adaptively analyze the probability of reaching different basic blocks without prior knowledge of program analysis and combine the probability with the basic-block-level distance (i.e., BB distance) to form a more effective fitness metric.

Challenge 2: Promoting the effectiveness of symbolic execution when solving the complex constraints in the path to the targets. As DGF takes the random mutation, it may not be able to satisfy the complex constraints within the allotted time budget. To address this issue, a complementary symbolic execution technique can be introduced to assist DGF, which should meet three requirements: (1) preferentially solving the path constraints of the branches closer to target sites, (2) preferentially solving the path constraints of the branches that are difficult to be covered by DGF, and (3) pruning the branches that are unsolvable by symbolic execution or do not contribute to reaching the target. However, the symbolic execution techniques used in state-of-the-art hybrid fuzzers (such as SAVIOR (Chen et al., 2020b), Symcc (Poeplau and Francillon, 2020), DigFuzz (Zhao et al., 2019), and Hydiff (Noller et al., 2020)) cannot meet all three requirements simultaneously. Thus, it is a challenge to design a complementary symbolic execution technique to effectively assist DGF.

In this paper, we propose HyperGo, the probability-based directed hybrid fuzzing. For challenge 1, we introduce the concept of path probability which is dynamically calculated based on branch hits, and then combine the path probability with BB distance to form an adaptive fitness metric called ***probability-based distance***. The path probability reflects the difficulty of DGF reaching one basic block while the BB distance reflects the likelihood of DGF reaching the target sites through the basic block. By combining the two factors, probability-based distance can effectively guide DGF toward paths that are closer to the target and have easier-to-satisfy path constraints (Section 3.1). Upon the introduction of a new fitness metric, it is imperative to optimize the power schedule to adaptively balance the exploitation of seeds with short distances and the exploration of more seeds that are reachable to the target sites (i.e., reachable seeds). To achieve this, we develop an optimization strategy for the power schedule called the Directed Multi-Armed Bandit (DMAB) model. Based on the continuously changing probability-based distance and path probability, the DMAB model adaptively assigns more energy to seeds that have shorter seed distances and higher probabilities of covering new branches.

For challenge 2, we propose an Optimal Symbolic Execution Complementary (i.e., OSEC) scheme that combines DGF and SE in a complementary way. In OSEC, we implement three strategies to improve the effectiveness of the combination between DGF and SE: (1) pruning branches that do not contribute to reaching target sites (i.e., unreachable branches), (2) pruning branches whose path constraints cannot be solved by SE (i.e., unsolvable branches), and (3) prioritizing the symbolic execution of the seeds whose paths are closer to the target and have more branches that are difficult to be covered by DGF. The first and second strategies aim at improving the efficiency of SE, while the third strategy aims at creating complementarity between SE and DGF. Specifically, we prompt DGF to explore branches with simpler path constraints, while SE is geared towards solving the path constraints of more complex branches. Based on this method, DGF and SE work in a complementary way and reach target sites more efficiently.

The main contributions of this paper are summarized as follows:

- We propose an adaptive fitness metric called probability-based distance, which combines basic-block-level distance with path probability to achieve higher accuracy and efficiency. It can steer DGF to reach the target sites faster through the closer paths which are easier to re-exercise.
- We propose a power scheduling optimization strategy called the DMAB model to implicitly balance the exploitation of seeds with short distances and the exploration of more reachable seeds. The seeds that have shorter seed distances and higher probabilities of covering new branches will be assigned more energy.
- We propose an OSEC scheme to combine DGF and SE in a complementary manner. The OSEC prunes the unreachable and unsolvable branches and prioritizes the symbolic execution of the seeds whose paths are closer to the target and have more branches that are difficult to be covered by DGF.
- We implemented a tool named HyperGo and evaluate it on 2 datasets consisting of 25 programs with a total of 120 target sites. The experimental results show that HyperGo achieves 37.75×, 29.11×, 23.34×, 95.61× and 143.22× speedup compared to AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan, respectively in reaching target sites, and 3.44×, 3.63×, 4.10×, 3.26×, and 3.00× speedup in exposing known vulnerabilities. Moreover, HyperGo discovered 10 undisclosed vulnerabilities from 5 real-world programs.
- HyperGo is publicly available on our website. https://gitee.com/paynelin/hypergo

2. Background and motivation

2.1. Background

We first introduce the background knowledge of the DGF techniques and hybrid fuzzing techniques.

Distance calculation and power schedule. AFLGo calculates the distances between the inputs and predefined targets. The seed distance is calculated as the arithmetic mean of BB distances of the basic blocks in the seed's trace. The BB distance is determined by the number of edges in the call graph and control-flow graphs to the target basic blocks while each edge has the same weight. Then, at run-time, AFLGo views the fuzzing process as a Markov chain and leverages a simulated annealing strategy to gradually assign more energy to the seeds that are closer to targets. It casts reachability as an optimization problem to minimize the distance between the generated seeds and their targets.

Hybrid fuzzing. Hybrid fuzzing involves a combination of fuzzing and symbolic execution. Fuzzing is excellent at exploring common code

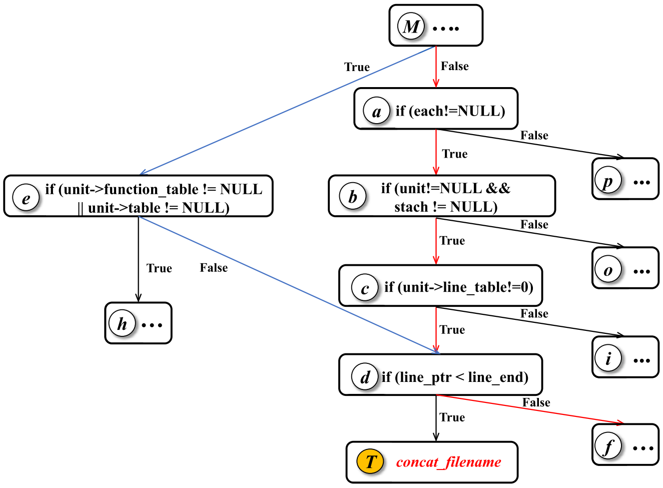


Fig. 1. Two execution traces toward target function `concat_filename()`. The nodes denote the basic blocks, and the branch conditions are represented nearby.

regions and discovering more paths, while symbolic execution can track seed execution paths and reverse branch conditions to identify the branches that are not covered by fuzzing, namely *unexplored branches*. The constraint-solver is then invoked to solve the path constraints of the unexplored branches in the abstracted syntax and generate new seeds to assist fuzzing in satisfying the path constraints. However, existing hybrid fuzzing techniques are not well-suited to meet the needs of DGF, and they face four problems: (1) solving unreachable branches, (2) solving unsolvable branches, (3) solving the branches that had been covered by DGF, and (4) performing not well in adaptively adjusting the solving priority of seeds and the time budget of solving branches. These problems are more pronounced in DGF compared to CGF.

2.2. Motivation

Example of challenge 1. Fig. 1 shows a real-world example (CVE-2017-15023) in GNU Binutils 2.29 (GNU, 2023). Two execution traces (Trace 1 <*M*, *a*, *b*, *c*, *d*, *f*> is marked as red lines, and Trace 2 <*M*, *e*, *d*, *f*> is marked as blue lines) are toward the bug function `concat_filename()`. The call site of `concat_filename()` is denoted as *T*. Following AFLGo, the seed distance of the seed covering Trace 1 (i.e., Seed 1) should be (1+2+3+4+3)/5 = 2.3, and the seed distance of the seed covering Trace 2 (i.e., Seed 2) should be (1+2+3)/3 = 2. Thus, Seed 2 has a shorter seed distance and will be assigned more energy. WindRanger introduces the concept of *NumOfEffectiveBytes* and combines it with the seed distance to form a new fitness metric called *DBB-distance*. Based on the new fitness metric, Seed 1 still has a greater DBB-distance (2.18) than Seed 2's DBB-distance (2.06). Thus, in both works, Seed 2 is given priority.

However, due to the fuzzer's random mutation strategy, it is difficult for the fuzzer to satisfy this branch condition and simultaneously cover the branches <*e*, *d*> and <*d*, *T*>. Therefore, even though Trace 2 is closer to the target site based on static analysis, it is infeasible for the fuzzer to reach the target sites. Even if Seed 2 is given more energy, the fuzzer still struggles to mutate Seed 2 and reach the target sites. Notably, both BEACON (Huang et al., 2022) and SelectFuzz (Changhua Luo and Li, 2023) would perform poorly in inferring path feasibility in this case. The symbolic execution of BEACON would fail to recognize the path infeasibility of Trace 2 due to the complex path constraints, and SelectFuzz (Changhua Luo and Li, 2023) would fail to recognize that Trace 2 is more complex than Trace 1 based on the number of successor basic blocks. Based on this real-program-based example, **we can conclude that the static fitness metric based on program analysis may be inaccurate. We need an adaptive fitness metric that combines the probability to more accurately guide DGF in different real programs and different fuzzing stages.**

Example of challenge 2. During our research, we investigated the state-of-the-art directed hybrid fuzzers and evaluated their performance in directed testing. For instance, we used SAVIOR (Chen et al., 2020b) to test tcpdump five times, and each test lasted for 24 hours. According to the test results, we found that only **37%** of the new inputs generated by the symbolic executor are reachable, only **28%** of the attempts to generate new seeds are successful, and only **41%** of the newly generated seeds are regarded as interesting. Furthermore, SAVIOR is incapable of dynamically adjusting the time budget for solving different branches, resulting in the skipping of some important branches within a very limited time budget and the inability to perform symbolic execution on all seeds within 24 hours. The investigation of SAVIOR demonstrates the issues of the state-of-the-art directed hybrid fuzzing techniques, such as solving unreachable or unsolvable branches, and generating a low proportion of interesting seeds. **Thus, we need to redesign the working scheme of symbolic execution to alleviate these issues and combine DGF and SE in a complementary manner.**

3. Probability-based directed hybrid fuzzing

In this paper, we propose a probability-based directed hybrid fuzzer named HyperGo. As Fig. 2 shows, HyperGo consists of the following three major components.

Static analyzer. The static analyzer is designed to provide precise information to both the directed greybox fuzzer and the symbolic executor, including unique basic block addresses, BB distances, and sibling branches of each branch. To calculate the address of all basic blocks, the static analyzer utilizes a hash algorithm based on the last statements of each basic block (for example, *exam.cpp:24* indicates line 24 of the file exam.cpp). Then, the static analyzer identifies sibling branches based on the successive basic blocks of each basic block. For example, if a basic block B_1 has two successors, B_2 and B_3, the branch $<B_1, B_2>$ and branch $<B_1, B_3>$ are sibling branches. Additionally, we adopt the same method as AFLGo to calculate the BB distances.

Directed greybox fuzzer. The directed greybox fuzzer continuously mutates seeds in an attempt to generate inputs that can cover target sites. We introduce the probability-based distance calculation module and the DMAB model to the fuzzer. The calculation module calculates the probability-based distance by analyzing the statistical path probability and BB distance, and the DMAB model optimizes the power schedule based on this new fitness metric.

Symbolic executor. The symbolic executor tracks the path of the seeds provided by the directed greybox fuzzer to identify unexplored branches. Then, the symbolic executor invokes the constraint-solver to solve the path constraints of the unexplored branches. We introduce the OSEC scheme to alleviate the limitations of hybrid fuzzing and complement the combination of DGF and SE.

At compile time, the static analyzer analyzes the program and stores the analysis information, such as BB distance, locally. This information is loaded by the directed greybox fuzzer and the symbolic executor as the fuzzing campaign is launched. During the fuzzing process, the directed greybox fuzzer continuously generates seeds and provides them to the symbolic executor. The symbolic executor tracks the paths of these seeds to identify unexplored branches and generates new inputs by solving path constraints of the unexplored branches. The new inputs are then fed back to the directed greybox fuzzer, enabling it to quickly reach target sites.

3.1. Probability-based distance

We introduce the concept of path probability and combine it with BB distance to form an adaptive fitness metric called ***probability-based distance***. In this section, we provide a detailed introduction to the concept and calculation method of probability-based distance.

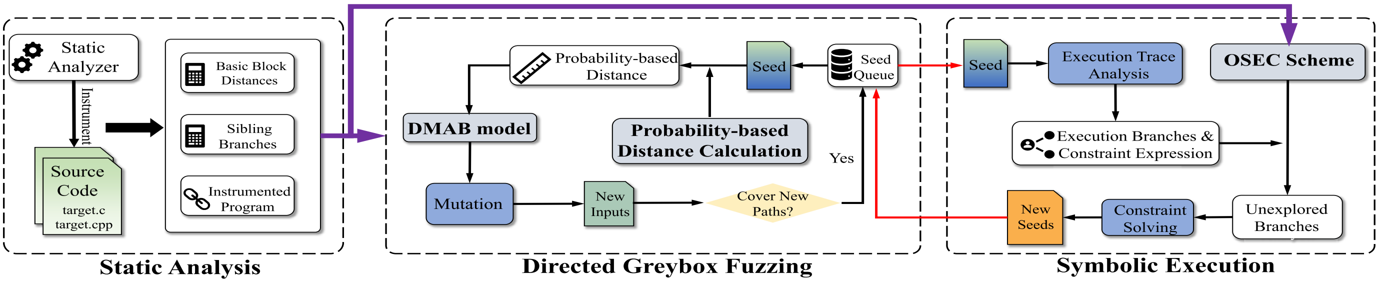


Fig. 2. The overview of HyperGo.

3.1.1. Definition of probability-based distance

In recent works, such as MDPC (Wang et al., 2018), a seed's execution trace in fuzzing is treated as a Markov Chain, and the path probability is calculated based on branch probabilities in the execution path. Based on this assumption, we propose a definition for path and path probability:

Definition 1 *(The path of a basic block).* Given the execution trace $Trace$ for a seed and a basic block m in $Trace$, the path of m, denoted as $path_m$, is a sequence of basic blocks in $Trace$ from the entry basic block to m.

Definition 2 *(Path probability).* The path probability of a basic block m, given its path $path_m$, is defined as the product of the probabilities of the branches that are covered by $path_m$:

$$P(path_m) = \prod \{P(br_i) | br_i \in path_m\} \tag{1}$$

Where br_i denotes a branch in $path_m$, $P(path_m)$ and $P(br_i)$ denote the path probability and branch probability, respectively. The branch probability is calculated based on the branch hits:

$$P(br_j) = \frac{hit_j}{\sum_{k=1}^{total} (hit_k)}, \tag{2}$$

where $P(br_j)$ denotes the branch probability of the j^{th} branch of the branch condition (j can be 1 or 2 for a binary branch), $total$ denotes the total number of branches, and hit_j denotes the number of branch hits of the j^{th} branch. Notably, the branch hits of the branches that are not covered by fuzzing will be regarded as 1 since we believe that any branch has the probability of being discovered.

Since the fuzzer utilizes a random mutation strategy, the process of fuzzing can be viewed as a form of random sampling. As the number of samples increases in random sampling, the statistical probability will gradually approach the theoretical probability. Therefore, it is reasonable to expect that the statistical branch probabilities and path probabilities will converge toward their theoretical values with the increasing number of mutations in fuzzing. The probability $P(br_j)$ for all branches will be updated every minute. We have carefully selected this update interval to strike a balance between computational overhead and accuracy. A very short interval would lead to a high overhead in updating $P(br_j)$ for all branches, whereas an overly long interval would result in inaccuracies when calculating $P(path_m)$ and the probability-based distance of each seed. Following thorough evaluations, we have concluded that updating $P(br_j)$ for all branches every minute is the optimal choice to maintain this balance.

As shown in Fig. 3, we use a simple artifactual C program as an example to illustrate how we calculate branch probability and path probability. In Fig. 3, each node (e.g., b_1) represents a basic block, and each edge (e.g., $<b_1, b_2>$) represents a basic block transition (i.e., branch). The path towards b_5 is a sequence of basic blocks starting from the entry basic block b_1, represented as $<b_1, b_2, b_3, b_5>$. The digitals above the line (e.g., 0.5 above branch $<b_1, b_2>$) represent the branch probabilities evaluated based on statistical probabilities. For example, if the branch hits of $<b_1, b_2>$ and $<b_1, b_{11}>$ are both 50,000, the branch probabilities of the two branches are both 0.5. After obtaining the branch probabilities, we can calculate the path probabilities of different basic blocks. For instance, the statistical path probability of b_5 is calculated as 0.5×0.3×0.3 = 0.045. Given the high throughput of fuzzing and the large number of random samples generated over time, it is reasonable and accurate to calculate branch and path probabilities based on statistical branch hits.

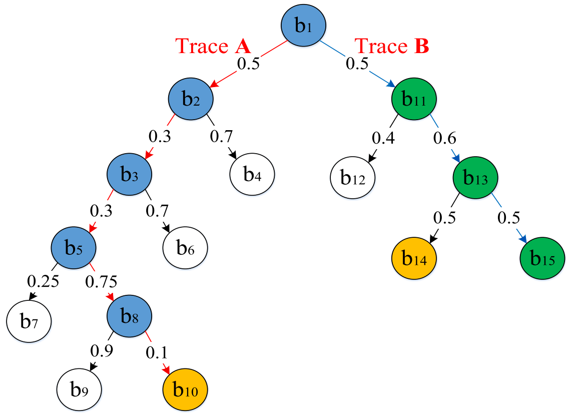


Fig. 3. The calculating method of path probability.

3.1.2. Calculation of probability-based distance

After obtaining the path probabilities of basic blocks, we combine them with BB distances to calculate the probability-based distances. The design is based on two considerations: (1) the probability-based distance is positively correlated with distance and negatively correlated with path probability, (2) introducing a factor of path probability to establish both an upper bound and a lower bound for the distance. This allows us to control the impact of path probability and amplify the effect of changes in path probability, as the probability-based distance will exponentially change with path probability.

$$d_p(m, T_b) = d_b(m, T_b) \cdot \mathrm{c}^{-P(path_m)} \tag{3}$$

Where $d_p(m, T_b)$ denotes the probability-based distance, T_b denotes the target basic block, $d_b(m, T_b)$ denotes the BB distance. $c^{(-P(path_m))}$ denotes the factor of path probability to establish both an upper bound ($d_b(m, T_b)$) and a lower bound ($\frac{c}{d_b(m,T_b)}$) for the distance. $P(path_m)$ denotes the path probability of m, and **c** is a constant which is greater than 1.

3.1.3. Calculation of seed distance

Up to now, almost all existing DGF techniques have used the arithmetic mean of all or part of the BB distances to calculate the seed distance. However, we have observed that in the seed's path, there are some basic blocks that are very close to or have already reached the target sites. These basic blocks contribute more to reaching and testing the target sites and we name them as ***critical basic blocks***. Since the arithmetic mean mainly reflects the general level of the population or the central tendency of the distribution (Plackett, 1958), it cannot reflect the existence of critical basic blocks in the path. **Therefore, we use the geometric mean, which is more sensitive to the minimum value, to calculate the seed distance.** For example, in Fig. 3, Trace A (<b_1, b_2, b_3, b_5, b_8, b_{10}>) is covered by Seed A, Trace B (<b_1, b_{11}, b_{13}, b_{15}>) is covered by Seed B, and b_{10} and b_{14} are the target basic blocks. Based on the arithmetic mean, the seed distance of Seed A $((0+1+2+3+4+3)/6 = 2.16)$ is greater than the seed distance of Seed B $((1+2+3)/3 = 2)$, resulting in Seed B being preferred. However, since Seed A can reach the target basic block, prioritizing Seed A is more reasonable. Given using the geometric mean, the seed distance of Seed A ($\sqrt[6]{0\times1\times2\times3\times4\times3} = 0$) will be smaller than the seed distance of Seed B ($\sqrt[3]{1\times2\times3} = 1.81$). We can see that, in this example, the geometric mean more accurately reflects the presence of critical basic blocks than the arithmetic mean. For this reason, we use the geometric mean to calculate the seed distance.

$$d_s(s,T_b) = \begin{cases} 0, \ if\ d_p(m_i,T_b) == 0 \wedge m_i \in \xi_b(s) \\ \sqrt[|\xi_b(s)|]{\prod_{m\in\xi_b(s)} d_p(m,T_b)}, \ else \end{cases} \tag{4}$$

Where s denotes the seed, $\xi_b(s)$ denotes the set of basic blocks in the execution path of the seed, and $|\xi_b(s)|$ denotes the number of basic blocks in $\xi_b(s)$. If there is a basic block m_i whose BB distance is 0, it means that the current seed has hit a target basic block. In this case, we consider its seed distance as 0.

Calculation Simplification. To avoid the high overhead caused by the Product and Sqrt operations, we combine Equation (3) and Equation (4) to simplify the calculation of the geometric mean.

$$d_s = \begin{cases} 0, \ if\ d_p(m_i,T_b) == 0 \wedge m_i \in \xi_b(s) \\ \exp\left\{ \frac{\sum_{m\in\xi_b(s)} (\log(d_b(m,T_b) - P(path_m))}{|\xi_b(s)|} \right\}, \ else \end{cases} \tag{5}$$

We mainly simplify the second term in Equation (4) based on Equation (3). First, we take the logarithm of both sides of the equation in Equation (3) to convert the Product and Sqrt operations into Summation operations, which yields $\sum_{m\in\xi b(s)} \log(d_b(m,T_b)\cdot \mathrm{c}^{-P(path_m)})$. Then, to optimize the computation process, we set the constant c in Equation (2) to **e**, so that we can convert the multiplication operation in $\log(d_b(m,T_b)\cdot \mathrm{c}^{-P(path_m)})$ into an addition operation, which yields $\sum m \in \xi_b(s)(\log(d_b(m,T_b) - P(path_m))$. Finally, we exponentiate the equation variables and obtain the optimized formula shown in Equation (5). Through this simplification method, the forking process only needs to perform addition operations, greatly reducing the computation overhead.

3.2. Power schedule optimization

Most of the existing DGF techniques use seed distance as the fitness metric, such as AFLGo and WindRanger, which explicitly divides the fuzzing process into the *exploration* phase and *exploitation* phase. The exploration phase is designed to uncover as many paths as possible (like many coverage-guided fuzzers), and DGF in this phase favors seeds that expose new paths and prioritizes them. Then, based on the known paths, the exploitation phase is invoked to drive the engine toward the target code areas. In this phase, DGF prioritizes seeds that are closer to the targets and assigns more energy to them. However, the constraints on the closer seeds' paths may be difficult for DGF to satisfy, leading to a failure in generating new seeds within the limited time budget. Therefore, it is challenging to assign reasonable energy for both phases. With the probability-based distance, we design a Directed Multi-Armed Bandit (i.e., DMAB) model to optimize the power schedule, which can implicitly coordinate the exploration and exploitation in DGF.

The Multi-Armed Bandit (i.e., MAB) problem results from the slot machine with multiple arms. The player plays one of the arms and obtains a reward. The player's main goal is maximizing the rewards in finite trials. Recent works, such as EcoFuzz (Yue et al., 2020) and MobFuzz (Zhang et al., 2022), have applied the Adversarial MAB model to improve CGF, where the arms represent the seeds, and the reward represents the probability of uncovering new paths by mutating a seed. Based on the Adversarial MAB, EcoFuzz assumes that the probability of uncovering new paths decreases as the coverage of CGF increases. However, different from CGF, HyperGo measures the difficulty of covering new branches based on the branch probability of unexplored branches. Since the branch probabilities are only related to the complexity of condition constraints, which is a fixed value, we model the process of HyperGo covering new branches as a *Stochastic Multi-Armed Bandit* problem.

In the stochastic MAB model, there are N fixed parallel arms, and at each time step *t*, one arm indexed as *i*, $(i \in K = 1,2,....,N)$ is selected to play. After playing arm *i*, the player receives a reward, and the rewards of each slot machine may follow a fixed probability distribution. Using the greedy algorithm, we tend to select the arm with the highest reward. However, to obtain the global optimum, we need to explore different arms to evaluate their reward probability and select the arm with the highest reward expectation. In the DMAB model, we map the elements in DGF to rewards and reward probabilities and combine them into reward expectations. Compared to traditional DGF methods, DMAB eliminates the need to explicitly differentiate exploration and exploitation. The assignment of seed energy is determined based on the difficulty of reaching the target sites for each seed, which implicitly incorporates the exploration and exploitation requirement.

3.2.1. Elements in DMAB model

We map the elements in DGF to the DMAB model as follows.

Reward. We consider mutating seed s as playing the slot machine. After mutating the seed s, a new input is generated and a reward is obtained. The value of the reward depends on whether the new input covers a new branch. There are two possible values for the reward: 0 or a value $r(d_s)$ which is negatively related to the seed distance, represented as d_s. The reward of 0 indicates that the new input cannot cover a new branch while the reward of $r(d_s)$ indicates that the new input covers a new branch. Based on a large number of tests, we find that new seeds generated from mutating the seed with shorter distance are more likely to have shorter distances. Thus, we believe that the reward is negatively correlated with the seed distance of seed s. **To exploit the seeds with shorter distances, we prefer the seeds with higher rewards and assign the seeds more energy**.

Reward probability. To estimate the reward expectation, we need to evaluate the probability distribution of the reward. Whether DGF can cover new branches is related to whether the input can satisfy the constraint conditions of unexplored branches in the path. Therefore, we use the average branch probability of all unexplored branches in the path of seed s to evaluate the probability of the fuzzer covering an unexplored branch through mutating seed s, represented as reward probability. **To explore more paths, we prefer the seeds that have higher probabilities of covering new branches and assign the seeds more energy.**

Reward expectation. We combine the reward and reward probability to evaluate the reward expectation. The design of the reward expectation is:

$$E(r) = \frac{1}{d_s(s,T_b)} \cdot \frac{\sum_{br\in\Theta(s)} P(br)}{|\Theta(s)|} \tag{6}$$

Where $d_s(s,T_b)$ denotes the seed distance of the seed s, $\Theta(s)$ denotes the set of unexplored branches in the path of s, $P(br)$ denotes the branch probability. The first term in Equation (6) represents the reward, which is negatively correlated with $d_s(s,T_b)$. The second term represents the reward probability, which is equal to the average branch probability of all unexplored branches.

Based on Equation (6), the DMAB model can dynamically coordinate the exploitation and exploration. As the fuzzing campaign launches, all seeds have not been sufficiently mutated and they have similar reward probabilities. Therefore, the DMAB model will exploit closer seeds with higher reward expectations. As fuzzing progresses, the closer seeds are sufficiently fuzzed and their reward probabilities will decrease as the number of mutations increases. According to Equation (6), the reward expectation of these closer seeds will gradually become lower than that of the seeds with higher reward probability. As a result, the DMAB model will assign more energy to the seeds with higher reward probabilities to explore more new paths, implicitly switching to exploration.

3.2.2. Design of power schedule

After obtaining the reward expectations of all seeds, we optimize the power schedule based on the reward expectations to coordinate exploration and exploitation in DGF. We design the power schedule for two objectives. Firstly, the fuzzer should assign more energy to seeds with higher reward expectations, and less energy to seeds with lower reward expectations. Secondly, the energy of seeds can be adaptively adjusted with the progress of fuzzing. Based on these two objectives, we redesign the power schedule in HyperGo.

Firstly, we normalize the reward expectations of all seeds.

$$\tilde{E}(r) = \frac{E(r) - \min E}{\max E - \min E} \tag{7}$$

Where $\min E$ denotes the minimum reward expectation among all seeds, and $\max E$ denotes the maximum reward expectation among all seeds. Then, we integrate the normalized reward expectations with AFL's power schedule to form an optimized power schedule.

$$P(s, T_b) = \begin{cases} P_{afl}(s) \cdot 2^{10 \cdot \tilde{E}(r) - 5}, & if \;\; \Theta(s) \neq \emptyset \\ \frac{P_{afl}(s)}{32}, & if \;\; \Theta(s) == \emptyset \end{cases} \tag{8}$$

Where $P_{afl}(s)$ denotes seed energy assigned by AFL's power schedule, $P(s, T_b)$ denotes the finally assigned seed energy after optimization. AFL's power schedule assigns basic energy to the seed based on the seed's characteristics, such as the seed's execution speed and the size of its bitmap. By taking into account both the seed's characteristics and the reward expectation, we integrate AFL's power schedule and reward expectations to design the optimized power schedule. Moreover, to prevent the overuse of seeds and neglect of seeds that may contribute more to reaching target sites, we design the term $2^{10 \cdot \tilde{E}(r) - 5}$ to control the adjustment of AFL's assigned energy within the range of [1/32, 32].

As fuzzing progresses, the number of unexplored branches and branch probabilities will change. This means that the reward expectations of all seeds will also change constantly. This allows HyperGo to dynamically assign seed energy and balance the trade-off between exploration and exploitation in a more accurate way. In Section 5.4, we demonstrate the effectiveness of the DMAB model in steering DGF to reach the target sites.

3.3. Optimized symbolic execution complementary scheme

To address the issues of directed hybrid fuzzing mentioned in Section 2.1, we design an Optimized Symbolic Execution Complementary (i.e., OSEC) scheme. In the OSEC scheme, we take three measures: pruning the unreachable branches, pruning the unsolvable branches, and dynamically prioritizing the symbolic execution of seeds. Algorithm 1 represents the workflow of the OSEC scheme.

In Algorithm 1, Ω_s represents the set of all seeds provided by DGF. D represents a dictionary containing (*seed index*, *seed distance*, *average branch probability*) triplets. Based on the seed's index, we can obtain the seed distance and the average branch probability of all unexplored branches in the seed's path (represented as $\bar{P}(\Theta(s))$). ψ_s represents the set of all new seeds generated by the symbolic executor, $\overline{SA}(\Theta(s))$ represents the average number of solving attempts for unexplored branches, and τ_{br} represents the set of the unexplored branch's path constraints. Before the symbolic execution of seeds, the seed's $\bar{P}(\Theta(s))$ is calculated by DGF based on Equation (6) and the $\overline{SA}(\Theta(s))$ is initialized to 1.

Algorithm 1 The workflow of OSEC scheme.

```
Input: $\Omega_s$, $D$
Output: $\psi_s$
 1: while $s \in \Omega_s$ do
 2:     $d_s, \bar{P}(\Theta(s)) \leftarrow D(s)$
 3:     Score(s) = Cal_Sco($d_s$, $\bar{P}(\Theta(s))$, $\overline{SA}(\Theta(s))$)
 4:     Sort($\Omega_s$)
 5: end while
 6: while true do
 7:     s ← Top_Rank($\Omega_s$)
 8:     $\Theta(s) \leftarrow$ Sym_Exe($s$)
 9:     while $br \in \Theta(s)$ do
10:         if $br$ is unreachable or $br$ is unsolvable then
11:             delete $br$ from $\Theta(s)$
12:         else
13:             $new\_seed \leftarrow$ Constraint_Solve($\tau_{br}$)
14:             $\psi_s = \psi_s \cup \{new\_seed\}$
15:         end if
16:     end while
17:     Score(s) = Cal_Sco($d_s$, $\bar{P}(\Theta(s))$, $\overline{SA}(\Theta(s))$)
18:     Sort($\Omega_s$)
19: end while
```

Firstly, the OSEC calculates the priority scores based on the seed distance, the average branch probability, and the average solving attempts. The priority scores are used to sort the seeds in descending order (Lines 2–5). Then, the OSEC continuously selects the seed with the highest priority score from Ω_s, tracks the execution path of the seed, and identifies all unexplored branches to form the set $\Theta(s)$ (Lines 7–8). Next, the OSEC determines whether the unexplored branches are unsolvable or unreachable. If so, the OSEC prunes these branches. If not, the OSEC invokes the constraint-solver to solve the path constraints of these branches to generate new seeds. The new seeds are added to the set of new seeds ψ_s (Lines 9–16). After the symbolic execution of the seed, the value of d_s, $\bar{P}_{br}(\Theta(s))$, and $\overline{SA}(\Theta(s))$ will all change. Therefore, we recalculate the priority score of the seed and re-sort all seeds (Lines 17–18). In the following sections, we will introduce in detail how to prune branches and calculate seed priority scores.

3.3.1. Pruning the unreachable branches

To avoid solving path constraints of the branches that do not contribute to reaching the target sites (i.e., unreachable branches), we need to prune the unreachable branches. We determine whether an unexplored branch is an unreachable branch based on the reachability of its destination basic block. We assume that if the destination basic block of a branch is unreachable, all successor basic blocks of that destination basic block are also unreachable. Thus, we consider this branch as an unreachable branch and give up solving its path constraints.

To determine whether an unexplored branch, represented as $<m_s, m_d>$, is unreachable, the OSEC loads the mappings of basic block addresses and BB distances ($<BB\_add, BB\_dis>$) provided by the static analyzer to obtain the BB distances of all basic blocks. Then, the OSEC checks the BB distance of m_d (whether $d_b(m_d, T_b) \geq 0$) to determine the reachability of branch $<m_s, m_d>$. If branch $<m_s, m_d>$ is unreachable, the symbolic executor will abandon solving this branch and prune it from the set of unexplored branches.

3.3.2. Pruning the unsolvable branches

To alleviate the issue of solving path constraints for the branches that cannot be solved (i.e., unsolvable branches) within a limited time budget, we need to prune the unsolvable branches. When solving the path constraints of a branch, we make two basic assumptions. (1) The time budget for solving the path constraints of each branch has a lower bound (e.g., 5 s) and an upper bound (e.g., 15 min). The time budget increases with the number of solving attempts, up to the upper limit. (2) If the symbolic executor fails to solve a branch due to the complexity of the branch's path constraints, all subsequent branches of that unsolvable branch are also unsolvable since they have more path constraints than the unsolvable branch.

Based on these two assumptions, we can prune the unsolvable branches. Firstly, we dynamically adjust the time budget within the ranger of $[lower\_bound, upper\_bound]$ according to the solving attempts. Based on empirical experience, we increase the time budget by 1 minute after each attempt. If the branch cannot be solved within the upper limit time budget, we consider it as an unsolvable branch and give up solving it. Secondly, during the symbolic execution of a seed, we would record the solving results (e.g., success or failure) of the unexplored branches that have been identified. Then, before solving each branch, we check whether its predecessor branch is unsolvable. If unsolvable, based on the second assumption, the path constraints of this branch are too complicated for the symbolic executor to solve within the limited time budget, and thus the branch is regarded as an unsolvable branch. Through these two steps, we prune the unsolvable branches to alleviate the pressure of constraint solving.

3.3.3. Prioritizing the symbolic execution of seeds

To achieve a complementary integration of DGF and SE, we need to dynamically adjust the order of seeds for symbolic execution. Due to its high fuzzing throughput, DGF can quickly cover paths with easy-to-satisfy path constraints. In contrast, SE has the excellent constraint-solving ability to generate inputs that satisfy complex path constraints. To fully leverage the respective strengths of DGF and SE, we want (1) both SE and DGF to prioritize seeds with shorter seed distances, (2) SE to prioritize solving unexplored branches that are difficult for DGF to cover, and (3) SE to prioritize seeds with fewer solving attempts and lower time budgets. We calculate the priority scores of different seeds based on these three considerations and adjust the order of seeds for symbolic execution accordingly.

Firstly, similar to the method of evaluating reward probability in Section 3.2, we use the branch probability of unexplored branches to evaluate the difficulty of the fuzzer covering the unexplored branches through mutating seed s.

$$EDF(s) = \frac{\sum_{br\in\Theta(s)} P(br)}{|\Theta(s)|} \tag{9}$$

Where $EDF(s)$ denotes the estimated difficulty. Similarly, we use the average branch probability of all unexplored branches in the path of seed s as the estimated difficulty. Moreover, based on the branch probability calculation method in Section 3.1, the branch probabilities of unexplored branches are all greater than 0.

Then, we evaluate the solving difficulty of branches based on the number of solving attempts. That is, more solving attempts indicate that the symbolic executor has more difficulty in solving the path constraints of the branch. We use the average solving attempts of all unexplored branches in the path of seed s to evaluate the solving difficulty of the seed's branches.

$$EDS(s) = \frac{\sum_{br\in\Theta(s)} SA(br)}{|\Theta(s)|} \tag{10}$$

Where EDS(s) denotes the estimated difficulty, $SA(br)$ denotes the number of solving attempts of the unexplored branch.

Based on $EDF(s)$, $EDS(s)$, and $d_s(s, T_b)$, we score the seeds to determine their order for symbolic execution. To ensure that the three indicators have the same weight in affecting the priority score of seeds, we use a normalization method as shown in Equation (7) to calculate their normalized value. Then, we can calculate the priority score of different seeds.

$$Score(s) = \frac{\widetilde{EDF}(s)}{\widetilde{EDS}(s) \cdot \widetilde{d}_s(s, T_b)} \tag{11}$$

Where $\widetilde{EDF}(s)$, $\widetilde{EDS}(s)$ and $\widetilde{d}(s, T_b)$ are the normalized value. After obtaining the priority scores of all seeds, the OSEC scheme will prioritize the seeds with higher scores for symbolic execution. Moreover, since the three factors used to calculate the priority scores, $EDF(s)$, $EDS(s)$, and $d(s, T_b)$, are changing dynamically, the OSEC scheme will adaptively adjust the order of symbolic execution of the seeds. By this method, the OSEC scheme dynamically prioritizes the optimal seeds whose unexplored branches are hard for the fuzzer to penetrate through, are more likely to be solved by the symbolic executor, and are closer to targets.

During the symbolic execution, the symbolic executor will constantly attempt to solve the unexplored branches in the seeds' paths and generate new interesting seeds for DGF.

4. Implementation

The implementation of HyperGo 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 AFL 2.52b, and the symbolic executor is built on Symcc. The implementation part of HyperGo is implemented with about 2000 lines of C/C++ and RUST code. HyperGo is publicly available on our website (https://gitee.com/paynelin/hypergo).

5. Evaluation

To evaluate the effectiveness of HyperGo, we conducted experiments aiming to answer five research questions:

RQ1: What about the performance of HyperGo in terms of reaching the target sites?
RQ2: What about the performance of HyperGo in terms of exposing the vulnerabilities in the target sites?
RQ3: How probability-based distance, the DMAB model, and the OSEC scheme take effect in the overall performance of HyperGo?
RQ4: Is the probability-based fitness metric effective in finding better seeds for directed fuzzing?
RQ5: What about the performance of HyperGo in terms of discovering new vulnerabilities?

5.1. Evaluation setup

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

(1) Time-to-Reach (TTR) is used to evaluate the time spent on generating the first input which can reach the specific target site.

(2) Time-to-Expose (TTE) is used to evaluate the time spent on exposing the (known or undisclosed) vulnerabilities in the target sites. When a crash is observed at the target site, it indicates that the fuzzer has successfully exposed the vulnerability.

Evaluation Benchmarks. We selected two datasets and 7 real-world programs with potential vulnerabilities.

(1) UniBench (Li et al., 2021) provides real-world programs of different types and the corresponding seed corpus. The state-of-the-art fuzzing techniques, such as WindRanger, have used the UniBench as the benchmark for testing. To answer RQ1, RQ3, RQ4, and RQ5, we tested the 20 programs from UniBench and used AFL++ (Fioraldi et al., 2020) to select target sites from each program by conducting preliminary experiments. We first ran AFL++ for 48 hours and collected all the seeds generated by AFL++. Then, we use afl-cov to re-run these seeds, so that we can obtain the code locations covered and the time when they are covered, represented as pairs like (line, time). Finally, among the locations that are reached using from 1 hour to 48 hours (i.e., more than 1 hour), we randomly selected 4 code locations as the targets.

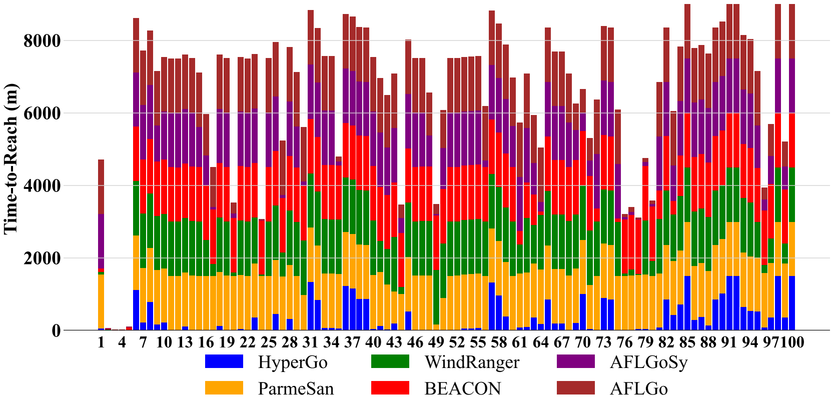


Fig. 4. TTR of AFLGo, AFLGoSy, BEACON, WindRanger, ParmeSan, and HyperGo on the UniBench.

(2) AFLGo testsuite (Böhme, 2023) was proposed in AFLGo's paper and website to evaluate the directness of DGF, and it had been used as a benchmark by many state-of-the-art directed fuzzers (e.g., Hawkeye and WindRanger). To answer RQ2, we selected it as the benchmark.

(3) Additional real-world programs. In addition to UniBench, we add another 9 real-world programs to construct a new testbench (all the programs are listed in Table 4). To answer RQ4, we used sanitizers (e.g., UBSAN (Undefined behavior sanitizer, 2023), ASAN (Serebryany et al., 2012)) to label the target sites for the testbench and tested them with HyperGo.

Baselines. In our evaluation, we compared HyperGo with the state-of-the-art directed greybox fuzzers that are publicly available by the time of writing this paper, including WindRanger, BEACON, ParmeSan, and AFLGo. To conduct the incremental experiments, we combined AFLGo with SymCC to form a new directed hybrid fuzzer, which is called AFLGoSy, as the baseline.

Experiment Settings. We conducted the experiments on the machine equipped with Intel(R) Xeon(R) Gold 6133 CPU @ 2.50 GHz with 80 cores 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 seeds in the BenchMarks' recommended seed corpus as initial seeds. Given that HyperGo requires two CPU cores to simultaneously launch both fuzzing and symbolic execution instances, the compared fuzzers also employed parallel fuzzing by launching two fuzzing instances (one acting as the master instance and the other as the slave instance). For experimental results analysis, we utilize the Mann-Whitney U test (p-value) to measure the statistical significance and the Vargha-Delaney statistic ($\hat{A}_{12}$) (Auer et al., 2002) to measure the probability of one technique performing better than another. For experimental results analysis, we utilize the Mann-Whitney U test (p-value) to measure the statistical significance and the Vargha-Delaney statistic ($\hat{A}_{12}$) (Auer et al., 2002) to measure the probability of one technique performing better than another.

5.2. Reaching target sites

To answer RQ1, we tested programs from UniBench, with a total of 100 target sites, and evaluated the TTR of different fuzzers. We set the timeout threshold as 24 hours. The detailed results of TTR are listed in Table 2. In Table 2, the entry "N/A" indicates that the fuzzer failed to compile the program due to code issues, while "T.O." indicates 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 encountering segmentation fault errors or being unable to obtain distance information during program testing. As for BEACON and ParmeSan, most entries showing "N/A" might be because it is incompatible with UniBench. For "N/A" entries, we did not use them to calculate the speedups and p-values. As for the "T.O." entries, we believe that these fuzzers might still reach the targets in subsequent fuzzing processes. Therefore, we opted for a slightly larger value of 1500 minutes to calculate speedups and p-values.

According to the results of TTR, HyperGo can reach the most (95/100) target sites compared to AFLGo (28/100), AFLGoSy (38/100), BEACON (14/100), WindRanger (25/100), and ParmeSan (11/100) within the time budget. Moreover, on most of the target sites (89/100), HyperGo outperforms all other fuzzers and achieves the shortest TTRs. In terms of mean TTR of reaching the target sites, HyperGo demonstrates 37.75×, 29.11×, 23.34×, 95.61× and 143.22× speedup compared to AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan, respectively. We conducted both the Mann-Whitney U test (p-value) and the Vargha-Delaney test ($\hat{A}_{12}$), all the p-values are less than 0.01, and the mean $\hat{A}_{12}$ against AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan are 0.88, 0.85, 0.92, 0.86, and 0.91, respectively. Based on the above analysis, we can conclude that **HyperGo can reach the target sites faster than baseline fuzzers**.

To reflect the results in a straight way, we use bar charts to visualize the results. In Fig. 4, the x-axis represents the target site ID (1-100), the y-axis represents the total TTR of all fuzzers in minutes, and a shorter bar indicates a shorter TTR. Since some fuzzers cannot compile some programs or reach the target sites within the 24-hour time budget, resulting no TTR. To distinguish these cases, the TTR of such a case is represented as 1500 minutes in Fig. 4. From the figure, we can clearly see that the blue bars are much shorter than the other bars, which means that HyperGo can reach most of the target sites faster than the baseline fuzzers.

5.3. Exposing vulnerabilities

To answer RQ2, following BEACON and WindRanger, we used the AFLGo testsuite and set the known vulnerabilities with CVE IDs in the programs as the target sites. The information on target sites and the TTE results are presented in Table 1. As Table 1 shows, among the 20 vulnerabilities, HyperGo exposed the most (18) compared to AFLGo (14), AFLGoSy (15), BEACON (13), WindRanger (16), and ParmeSan (14). Besides, on most of the target sites (15/20), HyperGo outperformed all the baseline fuzzers and achieved the shortest TTE. Among the 20 vulnerabilities, HyperGo costs longer time than baselines for three CVEs (2016-4487, 2016-4490, and 2015-8540). The three CVEs are swiftly discovered by all fuzzers within a few minutes of launching the fuzzing campaign. During this initial period, the branch hits for all branches are insufficient to accurately assess branch probabilities and calculate path probabilities. Consequently, in some instances during these first few minutes, there is a possibility of the fuzzer and symbolic executor incorrectly prioritizing branches. However, as the fuzzing process continues and the branch hits increase, HyperGo would address this issue and perform better than the baseline fuzzers in exposing the deeper bugs. With respect to the mean TTE of exposing vulnerabilities, HyperGo demonstrated 3.44×, 3.63×, 4.10×, 3.26× and 3.00× speedup compared to

Table 1
The results of TTE on AFLGo testsuite.

Prog.	CVE-ID	AFLG	AFLS	BEAC	Wind	Parm	HyGo
binutils$_{2.26}$	2016-4487	2.33m	2.42m	0.63m	1.21m	0.95m	1.42m
	2016-4488	4.23m	3.60m	32.1m	3.32m	2.62m	2.12m
	2016-4489	3.36m	4.11m	2.98m	5.88m	2.31m	1.89m
	2016-4490	1.15m	1.81m	2.35m	2.63m	0.82m	1.68m
	2016-4491	448m	389m	258m	298m	212m	69.3m
	2016-4492	10.8m	13.2m	43.6m	7.47m	4.33m	3.94m
	2016-6131	348m	236m	292m	318m	244m	101m
libming$_{4.48}$	2018-8807	331m	218m	267m	171m	301m	68.3m
	2018-8962	234m	271m	163m	121m	198m	43.7m
	2018-11095	T.O.	914m	252m	1311m	T.O.	118m
	2018-11225	T.O.	T.O.	438m	996m	T.O.	202m
LibPNG$_{1.5.1}$	2011-2501	10.2m	12.3m	N/A	7.81m	4.53m	2.16m
	2011-3328	69.1m	54.3m	N/A	49.3m	193m	21.1m
	2015-8540	0.88m	1.19m	N/A	0.96m	3.41m	2.65m
xmllint$_{2.9.4}$	2017-9047	T.O.	T.O.	T.O.	T.O.	T.O.	983m
	2017-9048	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
	2017-9049	T.O.	T.O.	T.O.	T.O.	T.O.	635m
	2017-9050	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
Lrzip$_{0.631}$	2017-8846	348m	284m	156m	223m	466m	69.4m
	2018-11496	201m	226m	98.1m	169m	126m	33.9m
speedup		**3.44×**	**3.63×**	**4.10×**	**3.26×**	**3.00×**	-
mean $\hat{A}_{12}$		**0.84**	**0.82**	**0.79**	**0.76**	**0.80**	-
mean p-values		0.009	0.013	0.006	0.026	0.008	-

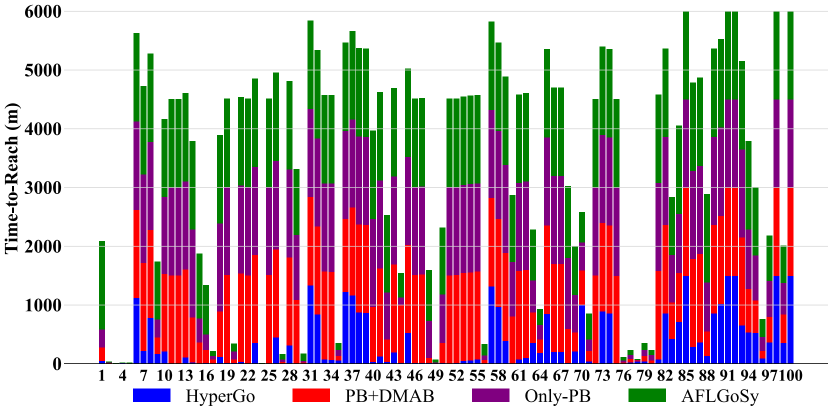


Fig. 5. Incremental experiment results of AFLGoSy, Only-PB, PB+DMAB, and HyperGo using TTR.

AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan, respectively. All p-values were less than 0.05, and the mean $\hat{A}_{12}$ against AFLGo, AFLGoSy, BEACON, WindRanger, and ParmeSan were 0.84, 0.82, 0.79, 0.76, and 0.80, respectively. Based on the above analysis, we can conclude that **HyperGo can expose known vulnerabilities faster than the baseline fuzzers**.

5.4. The impact of the optimizations on the overall performance

To answer RQ3, we conducted incremental experiments to evaluate the effects of the three optimizations on HyperGo's overall performance. We use AFLGoSy as the base tool. Since the DMAB model and OSEC scheme are based on the probability-based distance, disabling the probability-based distance calculation module will disable the other modules. Thus, we first add the probability-based distance module to AFLGoSy to implement a new tool (i.e., Only-PB). Then, we add the DMAB model to Only-PB, forming a new tool (i.e., PB+DMAB). Finally, we add the OSEC scheme to PB+DMAB to form HyperGo. In the incremental experiment, the configurations and the target sites remain unchanged as Section 5.2.

According to the TTR results, Only-PB (41), PB+DMAB (46), and HyperGo (95) can all reach more target sites than AFLGoSy (38). Moreover, HyperGo outperforms AFLGoSy, Only-PB, and PB+DMAB by 33.79×, 23.01×, and 14.78× respectively in the average TTR of reaching the target sites. Detailed results are listed in Table 2. These results demonstrate that **each optimization has a significant impact on reducing TTR, and using one or two optimization strategies (Only-PB and PB+DMAB) are far less effective than using all three optimization strategies simultaneously (HyperGo).** To visualize the experimental results, the results of TTR are shown in Fig. 5, in which the x-axis represents the target site ID (1-100), and the y-axis represents the total TTR of all fuzzers in minutes.

5.5. Intermediate data analysis

To demonstrate that HyperGo is more accurate than static-based DGF techniques and to more intuitively illustrate the effects of different optimizations, we analyzed the intermediate experimental data and used three metrics for analysis:

(1) The number of reachable seeds generated by the fuzzers, i.e., **Rseeds**. Through Rseeds, we can observe whether a fuzzer can cover more paths leading to target sites, thereby reflecting a

Table 2
The TTR results on programs from UniBench.

No	Prog	Version	Target sites	AFLGo	AFLGoSy	BEACON	WindRanger	ParmeSan	Only-PB	PB+DMAB	HyperGo
1	cflow	1.6	parser.c:281	T.O.	T.O.	99.4m	61.1m	T.O.	311m	224m	51.8m
2			c.c:1783	12.8m	9.43m	22.1m	6.45m	10.1m	8.41m	7.09m	8.82m
3			parser.c:105	0.82m	1.21m	13.5m	0.93m	0.44m	2.37m	2.88m	1.68m
4			parser.c:1223	1.22m	1.66m	0.83m	2.44m	6.23m	2.44m	7.21m	10.8m
5			parser.c:108	12.8m	8.63m	68.1m	8.32m	6.76m	4.33m	2.69m	1.65m
6	mp42aac	Bento4 1.5.1-628	Ap4AvccAtom.cpp:82	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	1124m
7			Ap4TrunAtom.cpp:139	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	223m
8			Ap4SbgpAtom.cpp:81	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	781m
9			Ap4TfdtAtom.cpp:71	T.O.	985m	N/A	T.O.	N/A	304m	287m	166m
10			Ap4AtomFactory.cpp:490	T.O.	1324m	N/A	T.O.	N/A	1318m	1311m	215m
11	jhead	3.00	exif.c:1339	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	5.52m
12			exif.c:1327	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	2.74m
13			iptc.c:143	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	107m
14			iptc.c:91	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	771m	20.8m
15			makernote.c:174	T.O.	1102m	N/A	T.O.	T.O.	417m	349m	11.3m
16	mp3gain	1.5.2	layer3.c:1116	1142m	841m	N/A	984m	N/A	262m	229m	10.84m
17			interface.c:690	1098m	81.9m	N/A	324m	N/A	67.6m	61.1	9.02m
18			mp3gain.c:602	T.O.	T.O.	N/A	T.O.	N/A	T.O.	771m	119m
19			interface.c:663	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	12.8m
20			apetag.c:341	290m	132m	N/A	91.2m	N/A	132m	67.4m	11.8m
21	lame	3.99.5	bitstream.c:823	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	36.8m
22			lame.c:2148	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	8.73m
23			uantize_pvt.c:441	T.O.	T.O.	N/A	1269m	N/A	T.O.	T.O.	354m
24			VbrTag.c:778	26.5 m	1.42m	N/A	39.1m	N/A	1.40m	1.41m	2.96m
25			get_audio.c:1605	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	11.5m
26	imginfo	jasper 2.0.12	jp2_cod.c:841	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	451m
27			jpc_dec.c:1393	T.O.	89.1m	N/A	653m	T.O.	39.0m	33.1m	0.35m
28			jp2_cod.c:636	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	314m
29			jas_stream.c:823	T.O.	1123m	N/A	T.O.	T.O.	1101m	1088m	0.71m
30			jpc_dec.c:1393	T.O.	121m	N/A	T.O.	984m	26.8m	23.0m	0.81m
31	gdk-pixbuf-pixdata	gdk-pixbuf 2.31.1	gdk-pixbuf-loader.c:387	T.O.	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	1339m
32			io-qtif.c:511	T.O.	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	841m
33			io-ani.c:403	T.O.	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	72.3m
34			io-jpeg.c:691	T.O.	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	68.6m
35			io-tga.c:360	126m	111m	T.O.	T.O.	N/A	106m	74.1m	60.7m
36	jq	1.5	jv_dtoa.c:3122	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	1241m	1223m
37			jv_dtoa.c:2004	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	1163m
38			jv_dtoa.c:2518	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	875m
39			jv_unicode.c:42	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	864m
40			jv_dtoa.c:3044	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	929m	37.1m
41	tcpdump	4.8.1	print-aodv.c:259	T.O.	T.O.	N/A	843m	T.O.	T.O.	T.O.	124m
42			print-ntp.c:412	1436m	1311m	N/A	974m	1239m	801m	383m	33.7m
43			print-rsvp.c:1252	T.O.	T.O.	N/A	T.O.	889m	T.O.	T.O.	194m
44			print-snmp.c:607	359m	412m	N/A	192m	992m	124m	992m	11.2m
45			print-l2tp.c:606	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	526m
46	tic	ncurses 6.1	captoinfo.c:189	T.O.	T.O.	N/A	N/A	T.O.	T.O.	T.O.	15.1m
47			alloc_entry.c:141	T.O.	T.O.	N/A	N/A	T.O.	T.O.	T.O.	19.1m
48			name_match.c:111	1186m	866m	N/A	N/A	T.O.	623m	86.6m	19.5m
49			comp_scan.c:860	264m	49.3m	N/A	N/A	168m	14.3m	7.24m	1.10m
50			entries.c:78	1038m	1134m	N/A	N/A	883m	824m	336m	19.9m
51	flvmeta	1.2.1	json.c:1036	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	10.1m
52			avc.c:1023	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	12.9m
53			api.c:718	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	50.1m
54			flvmeta.c:1023	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	60.6m
55			check.c:769	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	74.7m
56	tiffsplit	libtiff 3.9.7	tif_ojpeg.c:1277	T.O.	188m	N/A	T.O.	N/A	79.7m	60.4m	7.41m
57			tif_read.c:335	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	1321m
58			tif_jbig.c:277	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	967m
59			tif_dirread.c:1977	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	388m
60			tif_strip.c:154	1328m	1139m	N/A	T.O.	N/A	926m	796m	8.73m
61	nm	binutils-5279478	tekhex.c:325	T.O.	T.O.	364m	798m	N/A	T.O.	T.O.	78.4m
62			elf.c:8793	T.O.	T.O.	986m	T.O.	N/A	T.O.	T.O.	102m
63			dwarf2.c:2378	1313m	868m	831m	1065m	N/A	553m	512m	357m
64			dwarf1.c:281	T.O.	268m	98m	T.O.	N/A	241m	233m	187m
65			elf-properties.c:51	T.O.	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	853m

Table 2 (*continued*)

No	Prog	Version	Target sites	AFLGo	AFLGoSy	BEACON	WindRanger	ParmeSan	Only-PB	PB+DMAB	HyperGo
66	pdftotext	4.00	XRef.cc:645	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	201m
67			Stream.cc:2658	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	201m
68			GfxFont.cc:1337	1345m	1223m	N/A	T.O.	N/A	1208m	579m	17.5m
69			Stream.cc:1004	725m	824m	N/A	T.O.	N/A	631m	330m	207m
70			GfxFont.cc:1643	637m	514m	N/A	T.O.	N/A	477m	586m	1004m
71	sqlite3	SQLite 3.8.9	pager.c:5017	617m	436m	N/A	N/A	1214m	196m	174m	44.1m
72			select.c:4301	T.O.	T.O.	367m	N/A	T.O.	T.O.	T.O.	2.55m
73			func.c:1029	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	896m
74			insert.c:1498	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	857m
75			vdbe.c:1984	T.O.	T.O.	89.6m	N/A	T.O.	T.O.	T.O.	0.90m
76	exiv2	0.26	tiffcomposite.cpp:82	73.1m	59.3m	N/A	68.1m	N/A	26.0m	27.6m	1.90m
77			XMPMeta-Parse.cpp:1037	126m	78.1m	N/A	168m	N/A	72.9m	60.5m	21.8m
78			XMPMeta-Parse.cpp:847	37.5m	13.4m	N/A	21.4m	N/A	16.2m	14.3m	39.8m
79			tiffvisitor.cpp:1044	102m	111m	N/A	T.O.	N/A	109m	93.8m	39.5m
80			XMPMeta-Parse.cpp:896	86.7m	78.4m	N/A	421m	N/A	95.3m	57.2m	1.72m
81	objdump	binutils-2.28	elf.c:9509	T.O.	T.O.	782m	T.O.	T.O.	T.O.	T.O.	78.4m
82			section.c:936	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	862m
83			bfd.c:1108	T.O.	983m	361m	1288m	T.O.	812m	621m	423m
84			bfdio.c:262	T.O.	T.O.	1123m	T.O.	T.O.	1011m	833m	712m
85			stabs.c:372	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.
86	ffmpeg	4.0.1	rawdec.c:268	T.O.	T.O.	N/A	N/A	N/A	T.O.	T.O.	286m
87			decode.c:557	T.O.	T.O.	N/A	N/A	N/A	T.O.	T.O.	369m
88			dump.c:632	T.O.	T.O.	N/A	N/A	N/A	836m	411m	139m
89			utils.c	T.O.	T.O.	N/A	N/A	N/A	T.O.	T.O.	863m
90			eatgv.c:274	T.O.	T.O.	N/A	N/A	N/A	T.O.	T.O.	1021m
91	mujs	1.0.2	jsrun.c:572	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	T.O.
92			jsgc.c:47	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	T.O.
93			jsdump.c:292	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	652m
94			jsvalue.c:362	T.O.	T.O.	N/A	T.O.	N/A	1013m	736m	539m
95			jsvalue.c:396	T.O.	1165m	N/A	968m	N/A	761m	561m	523m
96	swftools	0.9.2	initcode.c:242	324m	301m	N/A	223m	N/A	241m	131m	89.3m
97			png.c:410	871m	769m	N/A	681m	N/A	617m	433m	364m
98			poly.c:137	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	T.O.
99			jpeg2swf.c:257	677m	632m	N/A	541m	N/A	541m	484m	355m
100			swfc.c:1794	T.O.	T.O.	N/A	T.O.	N/A	T.O.	T.O.	T.O.
			speedup	**37.75×**	**29.11×**	**23.34×**	**95.61×**	**143.22×**	**23.01×**	**14.80×**	-
			mean $\hat{A}_{12}$	**0.88**	**0.85**	**0.92**	**0.86**	**0.91**	**0.82**	**0.79**	-
			mean p-values	0.002	0.008	0.008	0.003	0.001	0.009	0.012	-

* T.O. means that the fuzzers cannot reach target sites within 24 hours and N/A means that the fuzzer cannot successfully test the programs.

Table 3
Intermediate data analysis using different seeds.

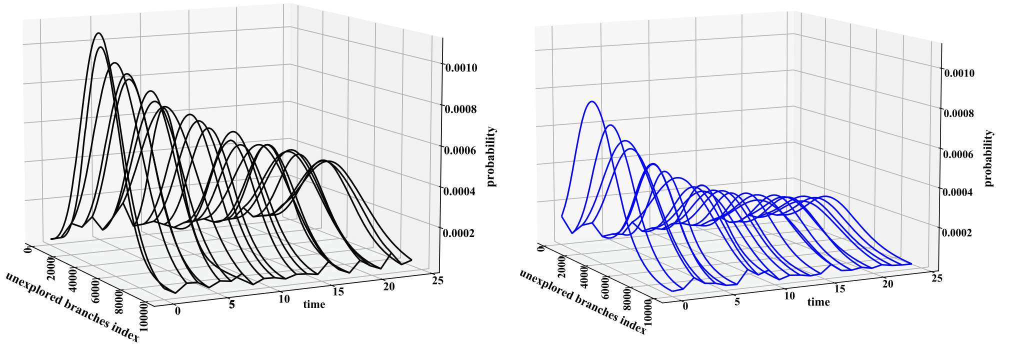
	AFLGo	BEAC	Wind	Parm	AFSy	On-PB	PB+DM	HyperGo
RSeeds	2311	312	2532	1463	2479	5112	7313	12432
PRseeds	52.4%	79.8%	64%	43.8%	46.3%	63.1%	69.3%	78.2%
SRseeds	-	-	-	-	18	22	29	267

fuzzer's accuracy in analyzing path reachability and the capability of satisfying path constraints.

(2) The proportion of reachable seeds (i.e., **PRseed**) among all seeds. If the number and proportion of reachable seeds are higher, it indicates that the fuzzer can avoid spending time on infeasible and unreachable paths.

(3) The number of reachable seeds generated by the symbolic executor, i.e., **SRseeds**. The more SRseeds indicate that the symbolic executor can provide more assistance to the fuzzer to cover new paths.

We evaluated the fuzzers on the programs from UniBench and counted the number of these three seed types, which are presented in Table 3. From Table 3, we can draw two conclusions. **Firstly, HyperGo can more accurately and efficiently explore more reachable and feasible paths to the target sites compared with other directed greybox fuzzers.** By comparing the Rseeds of all fuzzers, we can see that HyperGo can generate more reachable seeds within the same time budget. Although BEACON has higher PRseeds, it has the lowest Rseeds among all fuzzers due to wrongly pruning some reachable paths. The inaccuracy of static analysis prevents BEACON from exploring more reachable paths to the target sites. Apart from BEACON, HyperGo has the highest PRseeds among all fuzzers. **Secondly, the performance of HyperGo, which uses all three optimizations, is significantly better than that of the fuzzers using only one (Only-PB) or two strategies (PB+DMAB).** For the average number of RSeeds, both Only-PB (5112) and PB+DMAB (7313) are much greater than that of AFLGoSy (2479). This indicates that the probability-based distance and DMAB model can effectively explore more reachable paths to the target sites. As for the SRseeds, those of HyperGo are much greater than those of other fuzzers. This indicates that the OSEC scheme significantly improved the efficiency of symbolic execution, which helped and tested code areas that are difficult for the fuzzer to reach. Furthermore, according to the number of all three seed types, we can see that the performance of HyperGo, which uses all three optimizations, is significantly better than that of the



(a) HyperGo

(b) AFLGosy

Fig. 6. Branch probability distribution of HyperGo and AFLGoSy.

fuzzers using only one or two strategies. This implies that the overall design of HyperGo, including the new fitness metric, optimized power schedule, and the OSEC scheme, works in a complementary way and achieves significant improvement.

5.6. Branch probability distribution analysis of the unexplored branches

In the incremental experiments and intermediate data analysis, we observe that HyperGo can reach target sites faster than AFLGoSy and explore more paths leading to target sites. To verify whether the seeds preferentially selected by HyperGo using probability-based optimizations are better than those non-probability tools (RQ4), we analyze the high-priority seeds generated by HyperGo and AFLGoSy when testing UniBench. First, we divide the fuzzing process into 24 intervals, and each lasts for one hour. At each time point, such as 1-hour, we collect the top 100 seeds with the highest priority from the programs under test. Hence, at each time point, we collect 1600 seeds with the highest priority from the programs of UniBench. Then, we analyze the probability distribution of the seeds' unexplored branches.

The results are shown in Fig. 6, which includes two subfigures, depicting the unexplored branch probability distribution for HyperGo and AFLGoSy, respectively. For each subfigure, the x-axis represents the index of unexplored branches, the y-axis represents different time points, and the z-axis represents the branch probability. Each point (x, y, z) on the coordinate axis represents the branch probability of the x^{th} unexplored branch at y^{th} time point. Both Fig. 6(a) and Fig. 6(b) contain 24 branch probability distribution curves, corresponding to the 24 time points. To better illustrate the results, we sorted all branch probabilities and placed unexplored branches with higher probabilities closer to the middle. Then, to better visualize the overall branch probability, we performed curve fitting on all branch probabilities, resulting in a curve resembling a Gaussian distribution. We can obtain the maximum branch probability of different time points by identifying the highest point in the middle of the curve, and we can evaluate the overall branches' probabilities according to the overall height of the curve.

From Fig. 6, we can observe that the height of the AFLGoSy and HyperGo curves decreases gradually with the increasing of time, indicating the branch probability of all unexplored branches decreases during the fuzzing process. This is because the fuzzer is getting difficult to satisfy the branch conditions of unexplored branches as the fuzzing iterations increase. Furthermore, it is noteworthy that HyperGo's unexplored branches generally exhibit higher branch probabilities than AFLGoSy's. This indicates that the overall seed quality of HyperGo is better than that of AFLGoSy, making HyperGo can explore more paths toward target sites. This can be attributed to HyperGo's probability-based optimizations, which prioritize seeds that are more likely to cover unexplored branches as compared to AFLGoSy. Hence, we can conclude that **the probability-based fitness metric employed by HyperGo plays a crucial role in discovering better seeds, which are more likely to cover unexplored branches so as to explore more paths toward target sites.** This is essential for achieving faster attainment of the target sites in directed fuzzing.

```
1  static gboolean gdk_pixbuf__pcx_load_increment{
2   if(context->current_task==PCX_TASK_LOAD_DATA) {
3    switch(context->bpp) {
4     ...
5     case 4:
6      retval=pcx_increment_load_data_4(context);
7      static gboolean pcx_increment_load_data_4(){
8       ...
9       p=read_pixel_4(planes[0], i)&0xf;
10     }
11    }
12  }
13 }
14 static guchar read_pixel_4(){
15  if(!(offset % 2))
16   etval = data[offset] >> 4;
17 }
```

Listing 1: Example of a heap-overflow in gdk-pixbuf 2.31.1.

5.7. Discovering new vulnerabilities

To answer RQ4, we used HyperGo to test real-world programs. We first used sanitizers (i.e., UBSAN (Undefined behavior sanitizer, 2023) and ASAN (Serebryany et al., 2012)) to locate and label potential vulnerabilities as the target sites. Then, we run HyperGo for 24 hours to detect new vulnerabilities. Finally, HyperGo discovered 10 undisclosed vulnerabilities from 5 real-world programs. The information about these vulnerabilities is presented in Table 4. From the table, we can see that the new vulnerabilities involve heap-buffer-overflow, out-of-bounds read/write, and Null pointer deference.

We also used the baseline fuzzers to detect them. As a result, among the 10 discovered vulnerabilities, 5 could also be detected by AFLGo, 5 by AFLGoSy, 5 by BEACON, 4 by WindRanger, and 2 by ParmeSan. As for the reason that the vulnerabilities could not be detected, one is that the fuzzer could not run the program or obtain the distance information for analysis, while the other reason is the vulnerability could not be triggered within the time budget. From the above result, we can conclude that **HyperGo can detect new vulnerabilities from real-world programs, and it outperforms the baseline fuzzers**. We use the example in Listing 1 as a case study to explain why HyperGo could discover more vulnerabilities than the baseline fuzzers. Listing 1 shows a heap-overflow vulnerability in function `read_pixel_4()` of gdk-pixbuf 2.31.1. At Line 16, due to the lack of range restriction on variable `offset`, if the value of `offset` exceeds the memory allocated for the array `data`, a heap overflow would occur. To trigger this

Table 4
New vulnerabilities detected by HyperGo.

No	Prog	Bug location	Bug Type	CNNVD-ID	GSBWPH
1	cflow1.6	symbol.c:302	heap-buffer-overflow	2023-88222684	× ×✓✓×✓
2	gdk-pixbuf-2.31	gdk-pixdata.c:439	heap-buffer-overflow	2023-38595027	✓✓✓✓✓✓
3	gdk-pixbuf-2.31	io-qtif.c:437	out-of-bound read	2023-61429059	✓✓✓✓×✓
4	gdk-pixbuf-2.31	io-pcx.c:271	heap-buffer-overflow	2023-36676426	×××××✓
5	gdk-pixbuf-2.31	io-pcx.c:528	heap-buffer-overflow	2023-93057825	✓✓✓✓×✓
6	gdk-pixbuf-2.31	gdk-pixdata.c:142	heap-buffer-overflow	2023-18623971	××✓××✓
7	jhead-3.00	jpgqguess.c:195	heap-buffer-overflow	2023-28389092	××××✓✓
8	flvmeta-1.2.1	dump_xml.c:271	out-of-bound read	2023-88566232	×××××✓
9	fig2dev	bound.c:525	Null pointer dereference	2023-43290258	✓✓×××✓
10	fig2dev	arrow.c:89	out-of-bound read	2023-87146636	✓✓×××✓

1* In the last column, letters G, S, B, W, P, and H represent AFLGo, AFLGoSy, BEACON, WindRanger, ParmeSan, and HyperGo, respectively.
2* '×' denotes that the fuzzer was unable to discover the vulnerability, while '✓' signifies that the fuzzer was able to discover the vulnerability.

vulnerability, the fuzzer needs to generate inputs that satisfy both the path constraints at Line 1 and Line 3 and satisfy the root cause of the vulnerability. Within the time budget of 24 hours, AFLGo, AFLGoSy, WindRanger, and BEACON failed to generate specific inputs that satisfy all three conditions simultaneously to trigger this vulnerability. Based on three optimization strategies, HyperGo was able to reach Line 10 more efficiently and generated an input that triggers the vulnerability within 160 minutes.

6. Discussion

HyperGo adopts three optimizations to enhance the directedness, including the probability-based distance, the DMAB model, and the OSEC scheme. Specifically, the probability-based distance prioritizes the optimal seeds which have shorter seed distances and higher path probabilities. The DMAB model optimizes the power schedule, which implicitly balances the exploitation of seeds with short distances and the exploration of more reachable seeds. The OSEC scheme combines DGF and SE in a complementary manner. After pruning the unreachable and unsolvable branches, HyperGo prioritizes the symbolic execution of the seeds with higher scores to accelerate the speed of reaching targets. Experiments have proved the effectiveness of HyperGo in reaching the target sites (Section 5.2), exposing the known vulnerabilities (Section 5.3), and discovering new vulnerabilities (Section 5.7). Moreover, we also proved the effectiveness of the three optimizations (Section 5.4), and we can visually see their effectiveness via branch probability distribution of unexplored branches (Section 5.6).

Different from the SOTA experience-based and intuition-based DGF techniques, HyperGo adopts the probability-based fitness metrics and improvement methods that allowed it to maintain high accuracy across testing different programs in different fuzzing phases. The probability is calculated according to the simple branch hits rather than relying on program analysis or expert knowledge. Therefore, the probability-based fitness metric, the OSEC scheme, and the DMAB model can adaptively select the optimal seeds or optimal paths in current fuzzing phases according to the testing information. Compared to other DGF techniques, HyperGo's adaptability allows it to have higher accuracy when testing most programs.

Threat to validation. Aiming to design an adaptive approach, we adopt several heuristic parameters in HyperGo, which are set empirically (e.g., the setting of the adjustment factor). The values of these parameters might to some extent affect the performance of HyperGo. However, after extensive experiments, we believe the setting of these parameters is stable and suitable for most of the testing scenarios.

7. Related work

In this section, we focus on discussing the most related works: directed greybox fuzzing and directed hybrid fuzzing.

Directed Grey-box Fuzzing. AFLGo is the first directed greybox 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 (Chen et al., 2018) 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 if they assign more energy to the closer seeds, it is difficult for them to satisfy the complex path constraints to cover the path toward target sites. Some directed greybox fuzzers, such as LOLLY (Liang et al., 2019), Berry (Liang et al., 2020), UAFL (Wang et al., 2020a), and CAFL (Lee et al., 2021), propose new fitness metrics, such as 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 take a negative effect, which has been discussed in Section 2.2. Other directed greybox fuzzers use data flow information and data conditions information to enhance directedness. WindRanger (Du et al., 2022) uses the deviation basic blocks (DBBs) and the data flow information for seed distance calculation, seed mutation, seed prioritization, and power schedule. BEACON (Huang et al., 2022) 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 (Zong et al., 2020) uses the deep neural network to extract the features of reachable seeds and filter out the unreachable seeds to improve efficiency. To search the inputs that can reach the target sites, MC^2 designs an asymptotically optimal randomized directed greybox fuzzer that has logarithmic expected execution complexity in the number of possible inputs. However, DGF still suffers from being difficult to penetrate through the hard-to-satisfy path constraints. HyperGo selects the better paths that have fewer hard-to-satisfy path constraints and utilizes symbolic execution to assist DGF to pass 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 (Lattimore, 2016), SAVIOR (Chen et al., 2020b) and Badger (Noller et al., 2019) prioritize the seeds that may cause the specific program bug locations as the target sites, and then prioritizes symbolic execution of the seeds which are reachable from more target sites. DrillerGO (Kim and Yun, 2019), 1dvul (Peng et al., 2019), and Berry (Liang et al., 2020) 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 succeed in solving the unreachable branches, such useless constraint solving will have a negative impact on the directedness of directed hybrid fuzzing. Thus, HyperGo uses the OSEC scheme to prune the unreachable and unsolvable branches and prioritize the symbolic execution of the optimal seeds to better combine DGF and SE.

8. Conclusion

In this paper, we propose HyperGo, a probability-based directed hybrid fuzzer. HyperGo adopts the probability-based distance as the fitness metric and an optimized power schedule (namely DMAB model), which can steer DGF to faster reach the target sites through the paths that are easier to re-exercise and closer to the target sites. Using the OSEC scheme, HyperGo combines DGF and SE in a complementary manner to focus on solving constraints toward reachable targets. HyperGo is evaluated on 100 target vulnerabilities of 21 real-world programs from 2 datasets, the experiment results show that HyperGo outperforms the state-of-the-art directed fuzzers (AFLGo, BEACON, WindRanger, and ParmeSan) in reaching target sites and exposing known vulnerabilities. Moreover, HyperGo also discovered 10 undisclosed vulnerabilities and demonstrated its effectiveness in vulnerability discovery.

CRediT authorship contribution statement

Peihong Lin: Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Funding acquisition, Formal analysis, Data curation, Conceptualization. **Pengfei Wang:** Writing – review & editing, Writing – original draft, Visualization, Validation, Supervision, Software, Resources, Project administration, Methodology, Investigation, Data curation, Conceptualization. **Xu Zhou:** Visualization, Validation, Supervision. **Wei Xie:** Resources, Project administration, Methodology. **Kai Lu:** Validation, Supervision, Resources, Project administration, Methodology, Investigation. **Gen Zhang:** Writing – review & editing, Visualization, Validation, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

I have shared the artifact on-line.

Acknowledgements

This work is partially supported by the National Key Research and Development Program of China under Grant No. 2021YFB0300101, the National Natural Science Foundation of China (62272472, 61902405, U22B2005, 61972412, 62306328), the Hunan Provincial Natural Science Foundation (2021JJ40692), and the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013).

References

GNU Binutils. https://www.gnu.org/software/binutils/.

Undefined behavior sanitizer – clang 9 documentation. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.

Arshad, A., Weissbacher Blair, S., Mambretti, W., HotFuzz, M. Egele, 2020. Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. In: Network and Distributed System Security Symposium.

Auer, P., Cesa-Bianchi, N., Freund, Y., Schapire, R.E., 2002. The nonstochastic multiarmed bandit problem. SIAM J. Comput. 32 (1), 48–77.

Böhme, Marcel, 2023. Directed greybox fuzzing with AFL. https://github.com/aflgo/aflgo.

Böhme, Marcel, Pham, Van-Thuan, Roychoudhury, Abhik, 2016. Coverage-based greybox fuzzing as Markov chain. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security. Vienna, Austria, October 24–28, 2016. ACM, pp. 1032–1043.

BoHme, Marcel, Pham, Van Thuan, Nguyen, Manh Dung, Roychoudhury, Abhik, 2017. Directed greybox fuzzing. In: Acm Sigsac Conference on Computer & Communications Security, pp. 2329–2344.

Changhua Luo, Wei Meng, Li, Penghui, 2023. SelectFuzz: efficient directed fuzzing with selective path exploration. In: 2023 IEEE Symposium on Security and Privacy. SP.

Chen, Hongxu, Xue, Yinxing, Li, Yuekang, Chen, Bihuan, Xie, Xiaofei, Wu, Xiuheng, Liu, Yang, 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, pp. 2095–2108.

Chen, Hongxu, Guo, Shengjian, Xue, Yinxing, Sui, Yulei, Zhang, Cen, Li, Yuekang, Wang, Haijun, Liu, Yang, 2020a. MUZZ: thread-aware grey-box fuzzing for effective bug hunting in multithreaded programs. In: 29th USENIX Security Symposium. USENIX Security 20. USENIX Association, pp. 2325–2342.

Chen, Peng, Chen, Hao, 2018. Angora: efficient fuzzing by principled search. In: 2018 IEEE Symposium on Security and Privacy. SP 2018, Proceedings, San Francisco, California, USA, 21–23 May 2018. IEEE Computer Society, pp. 711–725.

Chen, Peng, Liu, Jianzhong, Chen, Hao, 2019. Matryoshka: fuzzing deeply nested branches. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. CCS 2019, London, UK, November 11–15, 2019. ACM, pp. 499–513.

Chen, Yaohui, Li, Peng, Xu, Jun, Guo, Shengjian, Zhou, Rundong, Zhang, Yulong, Wei, Tao, Lu, Long, 2020b. 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, pp. 1580–1596.

Du, Zhengjie, Li, Yuekang, Liu, Yang, Mao, Bing, 2022. WindRanger: a directed greybox fuzzer driven by DeviationBasic blocks. In: ICSE '22: 44th International Conference on Software Engineering. ACM.

Fioraldi, Andrea, Maier, Dominik, Eißfeldt, Heiko, Heuse, Marc, 2020. AFL++: combining incremental steps of fuzzing research. In: 14th USENIX Workshop on Offensive Technologies. WOOT 20. USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi.

Gan, Shuitao, Zhang, Chao, Chen, Peng, Zhao, Bodong, Qin, Xiaojun, Wu, Dong, Chen, Zuoning, 2020. GREYONE: data flow sensitive fuzzing. In: 29th USENIX Security Symposium. USENIX Security 20. USENIX Association, pp. 2577–2594. https://www.usenix.org/conference/usenixsecurity20/presentation/gan.

Ganesh, Vijay, Leek, Tim, Rinard, Martin, 2009. Taint-based directed whitebox fuzzing. In: 2009 IEEE 31st International Conference on Software Engineering, pp. 474–484.

Huang, Heqing, Guo, Yiyuan, Shi, Qingkai, Yao, Peisen, Wu, Rongxin, Zhang, Charles, 2022. Beacon: directed grey-box fuzzing with provable path pruning. In: The 43rd IEEE Symposium on Security and Privacy. S&P'22.

Kim, Juhwan, Yun, Joobeom, 2019. Poster: directed hybrid fuzzing on binary code. In: The 2019 ACM SIGSAC Conference.

Lattimore, Tor, 2016. Regret analysis of the finite-horizon gittins index strategy for multi-armed bandits. In: Feldman, Vitaly, Rakhlin, Alexander, Shamir, Ohad (Eds.), Proceedings of the 29th Conference on Learning Theory. COLT 2016, New York, USA, June 23–26, 2016. In: JMLR Workshop and Conference Proceedings, vol. 49, pp. 1214–1245. http://proceedings.mlr.press/v49/lattimore16.html.

lcamtuf, 2023. American fuzzy lop (AFL) fuzzer. https://lcamtuf.coredump.cx/afl/.

Lee, Gwangmu, Shim, Woochul, Lee, Byoungyoung, 2021. Constraint-guided directed greybox fuzzing. In: 30th USENIX Security Symposium. USENIX Security 21. USENIX Association, pp. 3559–3576. https://www.usenix.org/conference/usenixsecurity21/presentation/lee-gwangmu.

Lemieux, Caroline, Sen, Koushik, 2018. FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. ASE 2018, Montpellier, France, September 3–7, 2018. ACM, pp. 475–485.

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

Liang, Hongliang, Zhang, Yini, Yu, Yue, Xie, Zhuosi, Jiang, Lin, 2019. Sequence Coverage Directed Greybox Fuzzing (ICPC '19). IEEE Press.

Liang, Hongliang, Jiang, Lin, Ai, Lu, Wei, Jinyi, 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, pp. 127–137.

Ma, Kin Keung, Khoo, Yit Phang, Foster, Jeffrey S., Hicks, Michael, 2011. Directed symbolic execution. In: Static Analysis – 18th International Symposium. SAS 2011, Venice, Italy, September 14–16, 2011, Proceedings.

Marinescu, Paul Dan, Cadar, Cristian, 2013. KATCH: high-coverage testing of software patches. In: Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering. ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18–26, 2013. ACM, pp. 235–245.

Nguyen, Manh-Dung, Bardin, Sébastien, Bonichon, Richard, Groz, Roland, Lemerre, Matthieu, 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, pp. 47–62. https://www.usenix.org/conference/raid2020/presentation/nguyen.

Noller, Yannic, Kersten, Rody, Pasareanu, Corina S., 2019. Badger: complexity analysis with fuzzing and symbolic execution. In: Becker, Steffen, Bogicevic, Ivan, Herzwurm, Georg, Wagner, Stefan (Eds.), Software Engineering and Software Management. SE/SWM 2019, Stuttgart, Germany, February 18–22, 2019. In: LNI, vol. P-292. GI, pp. 65–66.

Noller, Yannic, Pasareanu, Corina S., Böhme, Marcel, Sun, Youcheng, Nguyen, Hoang Lam, Grunske, Lars, 2020. HyDiff: hybrid differential software analysis. In: Rothermel, Gregg, Bae, Doo-Hwan (Eds.), ICSE '20: 42nd International Conference on Software Engineering. Seoul, South Korea, 27 June–19 July, 2020. ACM, pp. 1273–1285.

Peng, Jiaqi, Li, Feng, Liu, Bingchang, Xu, Lili, Liu, Binghong, Chen, Kai, Huo, Wei, 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, pp. 605–616.

Plackett, R.L., 1958. Studies in the history of probability and statistics: VII. The principle of the arithmetic mean. Biometrika 45 (1–2), 130–135. https://doi.org/10.1093/biomet/45.1-2.130.

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

Serebryany, Konstantin, Bruening, Derek, Potapenko, Alexander, Vyukov, Dmitriy, 2012. AddressSanitizer: a fast address sanity checker. In: 2012 USENIX Annual Technical Conference. USENIX ATC 12. USENIX Association, Boston, MA, pp. 309–318. https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

Shah, Abhishek, She, Dongdong, Sadhu, Samanway, Singal, Krish, Coffman, Peter, Jana, Suman, 2022. MC2: rigorous and efficient directed greybox fuzzing (CCS '22). Los Angeles, CA, USA. https://doi.org/10.1145/3548606.3560648.

Shin, Y., Williams, L., 2013. Can traditional fault prediction models be used for vulnerability prediction? Empir. Softw. Eng. 18 (1), 25–59.

Wang, Haijun, Xie, Xiaofei, Li, Yi, Wen, Cheng, Li, Yuekang, Liu, Yang, Qin, Shengchao, Chen, Hongxu, Sui, Yulei, 2020a. 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, pp. 999–1010.

Wang, Xinyu, Sun, Jun, Chen, Zhenbang, Zhang, Peixin, Wang, Jingyi, Lin, Yun, 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, pp. 291–302.

Wang, Y., Jia, X., Liu, Y., Zeng, K., Su, P., 2020b. Not all coverage measurements are equal: fuzzing by coverage accounting for input prioritization. In: Network and Distributed System Security Symposium.

Wen, Cheng, Wang, Haijun, Li, Yuekang, Qin, Shengchao, Liu, Yang, Xu, Zhiwu, Chen, Hongxu, Xie, Xiaofei, Pu, Geguang, Liu, Ting, 2020. MemLock: memory usage guided fuzzing. In: ICSE '20: 42nd International Conference on Software Engineering. Seoul, South Korea, 27 June–19 July, 2020. ACM, pp. 765–777.

Yang, Guowei, Rungta, Neha, Khurshid, Sarfraz, Person, Suzette, 2011. Directed incremental symbolic execution. In: ACM SIGPLAN Notices: A Monthly Publication of the Special Interest Group on Programming Languages.

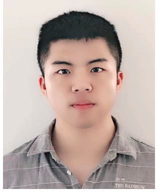
Yue, Tai, Wang, Pengfei, Tang, Yong, Wang, Enze, Yu, Bo, Lu, Kai, Zhou, Xu, 2020. EcoFuzz: adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit. In: 29th USENIX Security Symposium. USENIX Security 20. USENIX Association, pp. 2307–2324. https://www.usenix.org/conference/usenixsecurity20/presentation/yue.

Yun, Insu, Lee, Sangho, Xu, Meng, Jang, Yeongjin, Kim, Taesoo, 2018. QSYM: a practical concolic execution engine tailored for hybrid fuzzing. In: Enck, William, Felt, Adrienne Porter (Eds.), 27th USENIX Security Symposium. USENIX Security 2018, Baltimore, MD, USA, August 15–17, 2018. USENIX Association, pp. 745–761. https://www.usenix.org/conference/usenixsecurity18/presentation/yun.

Zhang, Gen, Wang, Pengfei, Yue, Tai, Kong, Xiangdong, Huang, Shan, Zhou, Xu, Lu, Kai., 2022. MobFuzz: adaptive multi-objective optimization in gray-box fuzzing. In: Proceedings 2022 Network and Distributed System Security Symposium. https://api.semanticscholar.org/CorpusID:248224859.

Zhao, Lei, Duan, Yue, Yin, Heng, Xuan, Jifeng, 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-myway-probabilistic-path-prioritization-for-hybrid-fuzzing/.

Zong, Peiyuan, Lv, Tao, Wang, Dawei, Deng, Zizhuang, Liang, Ruigang, Chen, Kai, 2020. FuzzGuard: filtering out unreachable inputs in directed grey-box fuzzing through deep learning. In: 29th USENIX Security Symposium. USENIX Security 20. USENIX Association, pp. 2255–2269. https://www.usenix.org/conference/usenixsecurity20/presentation/zong.



Peihong Lin received his B.S. and M.S. degree in the College of Command and Control Engineering from PLA Army Engineering University, China, in 2018 and 2022. Currently, he is pursuing the Ph.D degree in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and software testing.

Pengfei Wang received his B.S., M.S., and Ph.D degrees in computer science and technology, in 2011, 2013, and 2018 respectively, from the College of Computer, National University of Defense Technology, Changsha. He is now an associate professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems and software testing.

Xu Zhou received his BS, MS, and Ph.D degree in the School of Computer Science from National University of Defense Technology, China, in 2007, 2009, and 2013, respectively. He is now an associate professor in the School of Computer Science, National University of Defense Technology. His research interests include operating system and security.

Wei Xie received his Ph.D degrees in 2014 from the College of Electronic Science and Engineering, National University of Defense Technology, Changsha. He is now an associate professor in the College of Computer, NUDT. His research interests include security of Web, IoT, and AI.

Kai Lu received his B.S. degree and Ph.D. degree in 1995 and 1999, respectively, both in computer science and technology, from the College of Computer, National University of Defense Technology, Changsha. He is now a professor in the College of Computer, National University of Defense Technology, Changsha. His research interests include operating systems, parallel computing, and security.

Gen Zhang received his Ph.D. degree in computer science and technology in 2022 from National University of Defense Technology, Changsha. His research interests include fuzzing and testing.