DeepGo: Predictive Directed Greybox Fuzzing

Peihong Lin*, Pengfei Wang*✉, Xu Zhou, Wei Xie, Gen Zhang, Kai Lu✉
College of Computer, National University of Defense Technology
{phlin22, pfwang, zhouxu, xiewei, zhanggen, kailu}@nudt.edu.cn

***Abstract*—Directed Greybox Fuzzing (DGF) is an effective approach designed to strengthen testing vulnerable code areas via predefined target sites. The state-of-the-art DGF techniques redefine and optimize the fitness metric to reach the target sites precisely and quickly. However, optimizations for fitness metrics are mainly based on heuristic algorithms, which usually rely on historical execution information and lack foresight on paths that have not been exercised yet. Thus, those hard-to-execute paths with complex constraints would hinder DGF from reaching the targets, making DGF less efficient.**

In this paper, we propose DeepGo, a predictive directed greybox fuzzer that can combine historical and predicted information to steer DGF to reach the target site via an optimal path. We first propose the *path transition model*, which models DGF as a process of reaching the target site through specific path transition sequences. The new seed generated by mutation would cause the path transition, and the path corresponding to the high-reward path transition sequence indicates a high likelihood of reaching the target site through it. Then, to predict the path transitions and the corresponding rewards, we use deep neural networks to construct a Virtual Ensemble Environment (VEE), which gradually imitates the path transition model and predicts the rewards of path transitions that have not been taken yet. To determine the optimal path, we develop a Reinforcement Learning for Fuzzing (RLF) model to generate the transition sequences with the highest sequence rewards. The RLF model can combine historical and predicted path transitions to generate the optimal path transition sequences, along with the policy to guide the mutation strategy of fuzzing. Finally, to exercise the high-reward path transition sequence, we propose the concept of an *action group*, which comprehensively optimizes the critical steps of fuzzing to realize the optimal path to reach the target efficiently. We evaluated DeepGo on 2 benchmark suites consisting of 25 programs with a total of 100 target sites. The experimental results show that DeepGo achieves 3.23×, 1.72×, 1.81×, and 4.83× speedup compared to AFLGo, BEACON, WindRanger, and ParmeSan, respectively in reaching target sites, and 2.61×, 3.32×, 2.43× and 2.53× speedup in exposing known vulnerabilities.

I. Introduction

Directed Greybox Fuzzing (DGF) [5] is an efficient technique designed for testing vulnerable code areas. By defining a measurable fitness metric, the directed greybox fuzzer can select promising seeds and give them more mutation chances to approach the target site gradually. For example, based on the call graph and control-flow graph information of the program under test (PUT), mainstream DGF techniques use the distance between inputs and target sites as the fitness metric to assist seed selection and seed energy assignment. Seeds that are closer to the target site are regarded as promising and prioritized. DGF spends most of its time on reaching these locations without wasting resources stressing unrelated parts, thus, it is particularly suitable for testing scenarios including patch testing [38], bug reproduction [35], and potential buggy code verification [52].

To enhance directedness and improve efficiency, the state-of-the-art DGF works leverage heuristic methods to redefine and optimize fitness metrics. For instance, some DGF techniques redefine the fitness metric based on trace similarity (e.g., Hawkeye [8]), data conditions (e.g., CAFL [25]), and data flow information (e.g., WindRanger [12]). Some directed fuzzers use the sequence-based approach to enhance directedness, such as extending the given target sequence (e.g., Berry [28]) and taking the use-after-free sequence as guidance (e.g., Lolly [17]). And some DGF techniques improve efficiency by pruning unreachable paths to the target (e.g., BEACON [18]) and constructing the queryable oracle to guide fuzz (MC^2 [43]). However, in fuzzing, the mutation of seeds makes the execution path uncertain. Those heuristic methods usually rely on historical execution information and lack foresight on paths that have not been exercised yet. For example, when using the basic block level distance to the target as the fitness metric, seeds with a shorter distance are prioritized without considering the path feasibility, as a result, those hard-to-execute paths with complex constraints would hinder DGF from reaching the target sites, making DGF less efficient. Therefore, in this paper, we aim to design a predicted directed greybox fuzzer that can foresee critical execution information and predict the *optimal path*. By combining the historical execution information and the predicted future execution information, the fuzzer can intelligently generate the optimal and viable path to the target site. By avoiding the infeasible and hard-to-execute paths, the fuzzer can reach the target site more precisely and efficiently.

For this purpose, we propose to *model DGF as a process of reaching the target site through specific path transition sequences* and name the model as the ***path transition model***. The new seed generated by mutation would cause the path transition, and we use *reward* to evaluate the immediate impact of path transitions on the fuzzer. We use *sequence reward* to evaluate the difficulty of reaching the target site through a sequence of path transitions. The path corresponding to the high-reward path transition sequence indicates a high likelihood of reaching the target site through it. Compared to previous heuristic methods, this model takes the difficulty of reaching the target site through different path transition sequences into consideration. Besides, by analyzing the sequence rewards of different path transition sequences, an optimal path with the

*These authors contributed equally to this work.

Network and Distributed System Security (NDSS) Symposium 2024
26 February - 1 March 2024, San Diego, CA, USA
ISBN 1-891562-93-2
https://dx.doi.org/10.14722/ndss.2024.24514.
www.ndss-symposium.org

highest sequence reward can be generated, which can be used to guide the fuzzer to the target site more efficiently. To achieve this goal, we need to address three challenges.

Challenge 1: How to predict path transitions that have not been taken? To generate the optimal path to the target site, we need to collect all the potential path transitions. However, the existing techniques can only collect information on the known path transitions, not suitable for future path transitions. In DGF, different mutations would cause different path transitions. Within the limited time budget, the fuzzer cannot try all mutations (e.g., using all mutators to mutate all seed bytes) on the seeds. Besides, randomly selecting mutators and bytes for mutation can result in inefficient mutators being chosen and key bytes being missed. Therefore, we should predict the potential path transitions and the corresponding rewards caused by the mutations that have not been taken.

Challenge 2: How to determine the optimal path among large numbers of path transitions? In the path transition model, a higher sequence reward indicates a higher likelihood of reaching the target site through the path transition sequence. The optimal path can be represented by a path transition sequence with the highest sequence reward. However, the combination of historical and predicted path transitions creates large numbers of path transition sequences with different sequence rewards, Thus, we should efficiently evaluate the sequence rewards of all path transition sequences and design a policy to guide the mutations to realize the optimal path.

Challenge 3: How to exercise the optimal path transition sequences by optimizing the fuzzing strategies? Since DGF still employs random mutation strategies, the paths covered by the fuzzer are random and constantly changing. However, based on the path transition model, to efficiently steer the fuzzer towards the target site via the optimal path transition sequence, we should comprehensively optimize the critical steps of fuzzing, such as seed selection, energy assignment, mutator schedule, looping cycles in havoc [53], and mutation location selection. Therefore, we should be able to optimize the critical steps of fuzzing simultaneously to exercise the optimal path transition sequences efficiently.

To address these challenges, we propose DeepGo, a predictive directed greybox fuzzer. Based on the path transition model, we extract the path covered by one seed, mutation on the seed, path transition, and seed value changes caused by path transition in the DGF to a four-tuple *(path, action, next_path, reward)* (see Section III). For **Challenge 1**, we use deep neural networks (DNNs) to construct a Virtual Ensemble Environment (VEE). Given a path and an action, the well-trained VEE can predict potential path transitions and the corresponding rewards. With the increasing path transition information provided to VEE, VEE can gradually imitate the path transition model and predict the path transitions caused by the mutations that have not been taken, which greatly improves the efficiency of DGF. For **Challenge 2**, we propose a Reinforcement Learning for Fuzzing (RLF) model to determine the optimal path. To give RLF foresight, we use a k-step branch rollout strategy to continuously obtain predicted path transitions from VEE. By combining the historical and predicted path transitions, the RLF model is trained to evaluate the expected sequence rewards of path transition sequences caused by different mutations and determine the highest sequence reward. Additionally, the model can learn a policy for the optimal path to guide the mutation strategy of fuzzing. For **Challenge 3**, to exercise the optimal path, we optimize fuzzing strategies based on the concept of *action group*. In the action group, we comprehensively consider the five critical steps, including seed selection, energy assignment, the selection of looping cycles in havoc [53], mutator schedule, and mutation location selection. Under the mutation policy generated by the RLF model, we use a Multi-elements Particle Swarm Optimization (MPSO) algorithm to optimize them simultaneously, to realize the desired mutations and generate the path transition sequences, and ultimately to reach the target site via the optimal path.

In summary, we mainly make the following contributions:

- We propose the path transition model, which models DGF as a process of reaching the target site through specific path transition sequences. Based on the path transition model, we use sequence reward as the fitness metric to evaluate the difficulty of reaching the target site through a sequence of path transitions.
- We construct the VEE, using DNNs to imitate the path transition model and predict potential path transitions and the corresponding rewards without exercising the path, which greatly improves efficiency.
- We propose the RLF model, which can combine the historical and predicted information to generate the optimal path to the target site. By avoiding the infeasible and hard-to-execute path, the optimal path with the highest sequence reward can guide the fuzzer to the target site efficiently and precisely.
- We optimize the mutation strategy of fuzzing at the granularity of the action group, which is more efficient than single-strategy optimization. In the action group, we consider the five critical steps of fuzzing and optimize them simultaneously with an MPSO algorithm.
- We implement and evaluate DeepGo. The evaluation results demonstrate that the VEE can predict the path transitions with high accuracy and DeepGo can reach the target sites faster than baseline fuzzers.
- The artifact of DeepGo is available on our website. https://gitee.com/paynelin/DeepGo.

II. Background

Directed Greybox Fuzzing. Following AFLGo [6], the existing DGF techniques calculate the distances between the inputs and predefined targets based on the call graph and control-flow graph and combine the distance with other indicators (e.g., condition complexity) to form the fitness metric. Then, at runtime, the directed greybox fuzzing techniques design different power schedules according to the fitness metric to assign more energy to the seeds that are preferred. It casts reachability as an optimization problem to minimize the distance between the generated inputs and the targets.

Deep Neural Networks. In recent years, Deep Neural Networks (DNNs) have demonstrated their ability to approximate complex non-linear and non-convex functions and imitate the environment in pattern recognition [32], [47]. DNNs have been used in some fuzzing works, such as NEUZZ [45], MTFuzz [44] and FUZZGUARD [59], to simulate program branching behavior. In summary, these works collect the mutated inputs

generated during the fuzzing process as the input of DNNs and record the covered program branches as the labels to train DNNs. By this means, the DNNs can simulate the program branching behavior and guide the fuzzing optimization. The evaluations of these works prove that DNNs are appropriate for simulating the fuzzing environment.

Reinforcement Learning. Reinforcement Learning (i.e., RL) [16], [34], [41] is commonly used to solve sequential decision problems (such as the Markov process). In the RL model, the agents would constantly take actions to interact with the environment and receive feedback, namely rewards. Based on the feedback, represented as a four-tuple $(state, action, next\_state, reward)$, from the environment, the RL model would optimize the action selection strategies (i.e., *policy* in RL) to obtain the maximum rewards. Following AFLGo, we can model DGF as the Markov process and apply the RL model to optimize the fuzzing strategies, such as the selection of mutation operators and mutation bytes.

Model-Based Policy Optimization [19]. Model-Based Policy Optimization (MBPO) is a model-based reinforcement learning method consisting of two modules: a virtual environment and a reinforcement learning network. Firstly, MBPO uses DNNs to create a virtual environment to replace the real environment. Secondly, MBPO allows the agent to interact with the virtual environment to obtain a large number of path transitions. Then, MBPO can continuously train the reinforcement learning network and learn the policy that maximizes rewards for the agent based on the path transitions. In this paper, we refer to some methods from MBPO, such as the k-step branch rollout strategy, to combine DNNs and RL.

Particle Swarm Optimization. Particle Swarm Optimization (PSO) algorithm [21] is an evolutionary computation technique that originated from the study of bird flocking behavior during foraging. It has been applied to improve the fuzzing efficiency in CGF (e.g., MOPT [31] uses the PSO algorithm to optimize the selection probability of mutation operators based on the fuzzing historical information). The main idea behind PSO is to find the optimal solution through collaboration and information sharing among individuals in the population. PSO algorithm uses massless particles to simulate birds in a flock, and these particles have two primary attributes: velocity and position. Velocity represents the speed of movement, and position represents the direction of movement. Each particle searches for the optimal solution individually in the search space and records it as the local best value. The local best values are then shared among the particles in the entire swarm to find the global best value as the optimal solution. All particles in the swarm would adjust their velocity and position based on the local best value and the global best value shared in the entire swarm.

III. PATH TRANSITION MODEL

In this paper, we propose the path transition model, which models DGF as a process of reaching the target site through specific path transition sequences. The new seeds generated by mutations would cause path transitions, and we use rewards to evaluate the immediate impact of path transitions on the fuzzer. The path transition sequence with the highest sequence reward determines the optimal path to the target. In this section, we map the key elements in DGF to the path transition model and quantify the effectiveness of path transitions and actions.

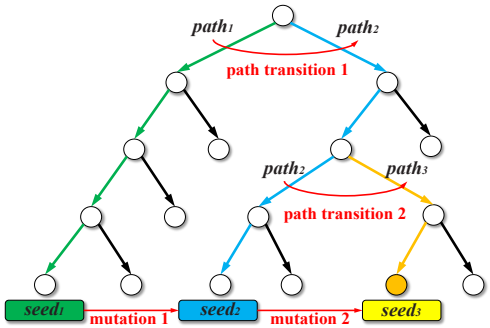


Fig. 1. Illustration of the path transition model.

A. Elements in Path Transition Model

Path. Each path corresponds to a seed in the seed queue of the fuzzer. We use `trace_bits` in AFL [6] to record the covered branches and branches' hit counts in the path and distinguish different paths.

Action. A fuzzer's action means to mutate a seed at a specific location. We pay attention to the location (i.e., bytes) where the mutation occurs instead of the mutator it uses. The fuzzer takes a series of actions to reach the target site gradually.

Path transition. Mutation on the seed causes the path transition if the execution path of the new input is different from that of the seed. If the new input's path is the same as the seed's, the mutation causes a self-path-transition.

Reward. The reward for a path transition represents the change in the seed's value caused by the path transition.

Policy. The policy is the strategy for the fuzzer to select actions in each path, represented as a list of probabilities corresponding to the actions. Under the policy, the fuzzer would select actions with different probabilities.

We use Fig. 1 to illustrate the path transition model. Fig. 1 shows the execution tree of an example program, where the nodes represent the basic blocks and the edges represent the basic block transition. The execution path $path_1$ marked green is covered by seed s_1, the execution path $path_2$ marked blue is covered by seed s_2, and the execution path $path_3$ marked yellow is covered by seed s_3. The node marked as orange is the target basic block, and $path_3$ is the optimal path that can reach the target site. During the fuzzing process, firstly, the fuzzer takes action to mutate seed s_1 to generate a new seed s_2, making $path_1$ transfers to $path_2$. Then, the fuzzer takes action to mutate seed s_2, making $path_2$ transfers to $path_3$ and reaching the target site. Through the two path transitions, the fuzzer generates the optimal path $path_3$ represented by a path transition sequence (*path transition 1*, *path transition 2*) that reaches the target.

B. Quantifying path transitions and actions

The fuzzer taking different actions to mutate seeds will cause different path transitions, we use *rewards* to quantify the effectiveness of path transitions and use *expected sequence rewards* to quantify the effectiveness of actions.

Seed value. We use seed value to evaluate different paths based on their contribution to the fuzzer reaching the target site. The seed value is calculated based on four characteristics of the seed corresponding to the path: (1) seed distance to the target, (2) the difficulty of satisfying the branch inversion, (3) execution speed, and (4) whether the seed is "favored". In DGF, a shorter seed distance implies that the fuzzer can reach the target site by satisfying fewer path constraints, and the lower difficulty implies that it is easier for the fuzzer to satisfy the path constraints leading to the target site. Moreover, during the seed execution process, the fuzzer may get trapped in loops (e.g., `while` and `for` loops), which reduces the execution speed and does not contribute to the fuzzer reaching the target site. Therefore, to enhance the fuzzing efficiency, we prefer seeds that have faster execution speed. Additionally, we prefer seeds that are marked as "favored" since these seeds can cover all the explored branches. By fuzzing the favored seeds, we can perform branch inversion on all explored branches to cover new branches leading to the target site. Following the methods of AFL [24] and AFLGo [6], elements of (1), (3), and (4) can be obtained by simply recording the execution information during fuzzing. Here, we mainly explain the concept of "difficulty" and its calculation.

We use branch probability to measure the difficulty of satisfying branch inversion, which is based on the statistics of branch hits. We first obtained information on the sibling branches of each branch during the static analysis. If the sibling branches of the covered branches are still uncovered (i.e., unexplored branches), satisfying the branch inversion to cover the unexplored branches would allow the fuzzer to transfer from the covered paths to new paths. Then, we count branch hits to calculate the branch probability of the unexplored branch. If the fuzzer consistently hits the covered branches with mutated inputs but cannot hit the unexplored branches, it indicates that the fuzzer has difficulties in satisfying the branch inversion. Finally, we quantify the difficulty of satisfying the branch inversion by branch probability:

$$P(ubr) = \frac{1}{\sum\limits_{br \in \phi(cond)} hit_{br} + 1} \tag{1}$$

Where ubr denotes an unexplored branch. We check whether the siblings of the covered branches are covered to find the unexplored branches. $\phi(cond)$ denotes the set of all branches under the same condition. hit_{br} denotes the branch hits recorded during fuzzing. $P(ubr)$ denotes the branch probability of the unexplored branches, which will not be 0 since we believe the unexplored branches always have the probability of being covered by fuzzing. We use the arithmetic mean of the branch probabilities of all unexplored branches in one seed's path to estimate the difficulty of satisfying the branch inversion:

$$ED_s = \frac{\sum_{br \in \Phi(s)} P(br)}{|\Phi(s)|} \tag{2}$$

Where s denotes the seed, ED_s denotes the estimated difficulty, and $\Phi(s)$ denotes the set containing all the unexplored branches in the seed's path.

Therefore, $distance$, $difficulty$, $execution\ speed$, and $favored$ can all be quantitatively measured and calculated to calculate the seed value. We use the Entropy Weight Method [26] to determine the weights of the four factors based on their information entropy. A small information entropy value makes a small factor weight, indicating the factor has a small impact on the overall evaluation of the seed value.

$$V^s(p_t) = W_1 \cdot d_s + W_2 \cdot ED_s + W_3 \cdot Ex_s + W_4 \cdot Fv_s \tag{3}$$

Where $V^s(p_t)$ denotes the seed value of path p_t, W_1, W_2, W_3, and W_4 are the weights calculated based on the Entropy Weight Method. d_s denotes the seed distance, Ex_s denotes the execution speed, and Fv_s indicates where it is $favored$, the value which could be 0 or 1. The Entropy Weight Method [26] is a commonly used multi-element comprehensive evaluation method. The steps for calculating the weights of different elements are as follows:

(1) Calculate the entropy of each element:

$$E_j = -\frac{1}{\ln(n)} \sum_{i=1}^{n} \frac{p_{ij}}{\ln(p_{ij})} \tag{4}$$

(2) Calculate the weight of each element:

$$W_j = \frac{1 - E_j}{n - \sum_{j=1}^{n} E_j} \tag{5}$$

(3) Normalize the weight of each element:

$$W_j' = \frac{W_j}{\sum_{j=1}^{n} w_j} \tag{6}$$

Where W_j' denotes the normalized weight of the j^{th} element. The core idea of the Entropy Weight Method is to determine the weight of each element by calculating their entropies, so as to conduct multi-element comprehensive evaluation.

Then, we calculate the reward based on the seed value to evaluate the effectiveness of the path transition:

$$r(p_t, a_t, p_{t+1}) = V^s(p_{t+1}) - V^s(p_t) \tag{7}$$

Where $r(p_t, a_t, p_{t+1})$ denotes the reward, a_t denotes the action that the fuzzer takes to transfer from path p_t to path p_{t+1}, $V^s(p_{t+1})$ and $V^s(p_t)$ denote the seed value of p_{t+1} and p_t. We use a four-tuple (p_t, a_t, p_{t+1}, r_t) to denote a path transition.

Expected sequence reward. In the path transition model, the path transitions caused by the actions selected according to the policy will affect the subsequent path transition sequences and thus affect reaching the target site. To evaluate the contribution of path transitions to reaching the target site, we define the expected sequence reward as *the expected sum of rewards of the path transition sequences generated by the fuzzer following a certain policy.* It can be computed recursively using the Bellman equation [4]:

$$Q_\pi(p, a) = \mathop{E}_{p' \sim P}[r(p, a, p') + \gamma V_\pi^t(p')] \tag{8}$$

Where $p' \sim P$ denotes the probability of transferring from path p to path p'. γ represents the discount factor, and the influence of subsequent path transitions on the expected sequence reward will gradually decrease. $r(p, a, p')$ denotes the reward of the path transition, $V_\pi^t(p')$ denotes the transition value of path p'. The calculation of the transition value of p' is:

$$V_\pi^t(p') = \begin{cases} 0, & if \ \ p = p_{ter} \\ \sum_a \pi(a|p) \cdot Q_\pi(p', a), & others \end{cases} \tag{9}$$

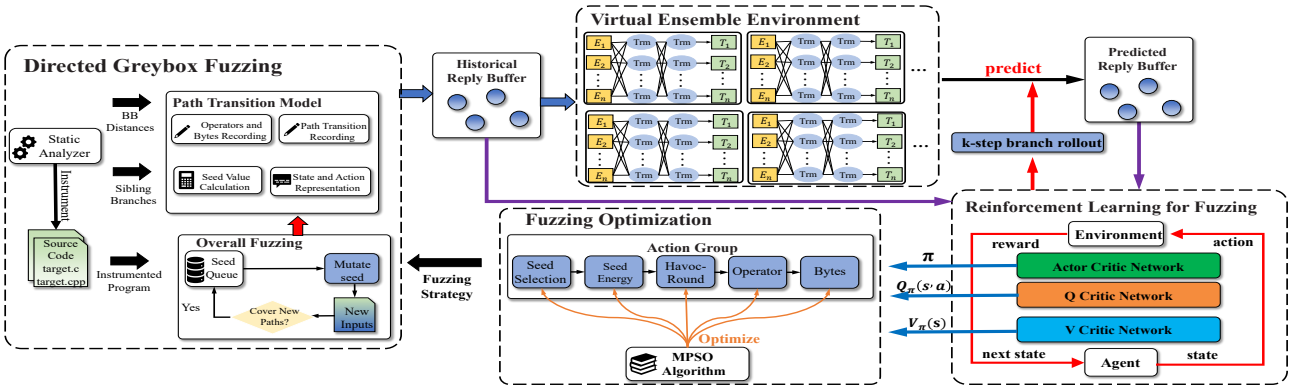


Fig. 2. The overview of DeepGo.

Where p_{ter} denotes the terminal path. We consider the path as the terminal path if all actions of the path can only cause self-path-transition. π represents the policy to select actions (e.g., in AFL, the fuzzer employs the random policy to select actions for mutation). $\pi(a|p)$ represents the probability of path p selecting action a under the policy π. If p' is the terminal state, its transition value is 0. If not, its transition value is equal to the weighted average of the expected sequence rewards of all actions. The Bellman equation is used to recursively update the transition value and expected sequence reward until the path is the terminal path. By maximizing the expected sequence reward, the fuzzer can learn a policy that maximizes the long-term cumulative reward.

IV. METHOD

A. Overview of DeepGo

Based on the path transition model, we design a predictable directed greybox fuzzer, DeepGo. DeepGo uses DNNs to predict potential path transitions and the corresponding rewards. Then, it uses reinforcement learning to combine historical and predicted path transitions to obtain the optimal path transition sequence with the corresponding policy. Finally, based on the action group, it optimizes the fuzzing strategies comprehensively to exercise the optimal path transition sequences. As Fig. 2 shows, DeepGo mainly consists of four components.

Directed Greybox Fuzzing Component. The DGF component continuously mutates seeds to generate inputs for reaching the target sites. This component contains a static analyzer and a fuzzer. At compile time, the static analyzer calculates the basic-block-level distance (BB distance), records the sibling branches of each branch, and instruments the target program. Once the fuzzing campaign is launched, the fuzzer continuously mutates seeds to test the program. Notably, the path transition model is incorporated into the DGF component.

Virtual Ensemble Environment. VEE is used to predict the potential path transitions and the corresponding rewards. VEE consists of DNNs and shares the *historical reply buffer* and *predicted reply buffer* with the Reinforcement Learning for Fuzzing component. The historical replay buffer and predicted reply buffer are both data buffers that store the four-tuples, i.e., $(path, action, next\_path, reward)$. Given a tuple, $(path, action)$, the trained VEE would predict the next path and reward of the action, represented as $(next\_path, reward)$, according to the probabilities of different path transitions.

Reinforcement Learning for Fuzzing Model. This model leverages the reinforcement learning model to combine the historical path transitions and predicted path transitions to learn the policy that maximizes sequence rewards. The RLF model consists of the *Actor network*, *Q-Critic network*, and *V-Critic network*. After training, the Q-Critic network can evaluate the expected sequence reward $Q_\pi(p, a)$ caused by each action, the V-Critic network can evaluate the transition value $V_\pi^t(p)$ of each path, and the Actor network can learn the policy π to maximize sequence rewards.

Fuzzing Optimization Component. This component exercises the optimal path transition sequences by optimizing the fuzzing strategies. Based on the action group, we can comprehensively optimize critical steps of fuzzing, and we use the Multi-elements Particle Swarm Optimization algorithm to optimize the elements of an action group simultaneously.

We divide the fuzzing process of DeepGo into different fuzzing cycles, with each cycle lasting approximately 20 minutes. In each fuzzing cycle, DeepGo needs to conduct four tasks, including (1) using the fuzzer to test programs and provide historical path transitions to train VEE and RLF model, (2) VEE providing predicted path transitions to train RLF model, (3) RLF providing transition values, expected sequence rewards, and policy to the FO component, and (4) the FO component using the MPSO algorithm to optimize the action group and providing the optimization strategies of fuzzing to DGF. After DeepGo completes these four tasks, it will enter the next fuzzing cycle and repeat these four tasks.

B. Virtual Ensemble Environment

To design a policy that can guide the fuzzer to optimal paths, the fuzzer has to obtain rewards for all path transitions caused by actions taken in paths. However, the fuzzer cannot take all actions in all paths within a limited time budget (e.g., 24 hours). To predict the potential path transitions and rewards caused by actions that have not yet been taken, we design VEE to imitate the path transition model and predict the potential path transitions and the corresponding rewards.

1) Input and output encoding: Before training VEE, we should encode the path, action, and reward. We design the encoding method of VEE for two purposes. First, VEE predicts the potential path transitions for new paths based on the recorded path transitions already taken in explored paths. Second, to improve VEE's training efficiency, we need to use

low-dimensional vectors to represent paths and actions. For example, mapping the `trace_bits` to a 65536-dimensional vector to represent the path will significantly slow down the training of DNNs. Thus, while ensuring that different paths and actions can be clearly distinguished, we try to use low-dimensional vectors to represent paths, actions, and rewards.

Path. We apply the Coupled Data Embedding (CDE) [20] algorithm to encode the path (represented by `trace_bit`) in AFL as a 20-dimensional continuous vector and normalize the values of each dimension. CDE is used to represent discrete vectors as continuous vectors while preserving the main features that distinguish paths. According to the method of CDE, a 20-dimensional vector can both distinguish the main features of different paths and reduce the dimension of paths to improve the training efficiency of DNNs. If two paths represented by `trace_bits` are more similar, the Euclidean distance between the corresponding 20-dimensional vectors is closer.

Action. We encode the action based on the mutation location. Take the example from Fig. 1, seed s_1 represents a seed of 100 bytes, if the fuzzer selects the 4^{th} byte of s_1 as the mutation location, regardless of which mutator is selected, the fuzzer will mutate s_1 starting from the 4^{th} byte, and the encoding of this action is 4/100=0.04. The value of all actions is also normalized.

Reward. The reward is a scalar which is calculated according to Equation 7.

Employing this encoding method enables VEE to predict the probability and reward of potential path transitions based on historical fuzzing information. VEE is trained with the four-tuples, namely $(path, action, next\_path, reward)$, in which action represents the bytes of a seed where the mutations can occur, not a concrete mutation. Thus, the same action with different mutators would cause different path transitions with different probabilities. By analyzing the historical information of mutations on one byte, VEE can predict the path transitions caused by taking this action, i.e., mutating this byte with different mutators. For bytes that have been mutated, we use VEE to predict the probabilities and rewards of different path transitions caused by mutating the byte using different mutators. For seeds or bytes that have not been mutated yet, we use seeds with similar structures or bytes with similar offsets to predict the probabilities and rewards of path transitions caused by mutation actions. The similarity is measured based on the CDE encoding method. If the 20-dimensional vectors representing two seeds have a shorter Euclidean distance, it indicates that the two seeds are more similar. Additionally, within a seed, similar encoded values of two actions indicate a similar offset of the corresponding bytes.

2) Training of VEE: We use DNN to construct VEE to imitate the path transition behavior in the path transition model. Formally, let f:$(path, action) \longrightarrow (next\_path, reward)$ denote the DNNs that takes the tuple $(path, action)$ as input and outputs the tuple $(next\_path, reward)$. We use θ to denote the trainable weight parameters of DNN and train DNN with a set of training samples (X, Y), where X denotes a set of inputs and Y denotes the corresponding outputs. In the path transition model, since the same action on the same path may cause different path transitions with different probabilities, the path transition model is essentially a probabilistic model. Therefore, when designing DNN f:$(path, action) \longrightarrow (next\_path, reward)$ and loss function, we mainly address VEE's aleatoric and epistemic uncertainties to improve its prediction accuracy.

Aleatoric uncertainty. Aleatoric uncertainty arises from the unpredictability of path transitions in fuzzing. For example, using different mutators for the same mutation location may cause different path transitions. To capture the aleatoric uncertainty, we use the Gaussian probability distribution of the next paths and rewards to predict probabilities of different path transitions and the corresponding rewards that may be caused by action a_t taken in path p_t. We use the trainable weight parameters to represent the Gaussian probability distribution of the next path p_{t+1} and reward r_t can be represented according to the input tuple (p_t, a_t):

$$P(p_{t+1}, r_t | p_t, a_t, \theta) = N(\mu_\theta(p_t, a_t), \Sigma_\theta(p_t, a_t)) \tag{10}$$

Where N represents the Gaussian distribution and μ_θ represents the mean of Gaussian distribution. We use Σ_θ to represent the variance, which indicates uncertainty about the mean. We define the loss function between the output of the DNN and the ground truth label $y \in Y$ in the training set as:

$$\mathcal{L}(\theta) = \sum_{n=1}^{N} [\mu_\theta(s_n, a_n) - s_{n+1}]^T \Sigma_\theta^{-1}(s_n, a_n) [\mu_\theta(s_n, a_n) - s_{n+1}] + \log\det \Sigma_\theta(s_n, a_n) \tag{11}$$

The training task is to find the weight parameters $\hat{\theta}$ of the DNN f to minimize the loss.

Epistemic uncertainty. Epistemic uncertainty results from the random sampling method employed by most DNNs. Since a single DNN cannot sample all the training data, there may be areas where the DNN has epistemic defects and cannot accurately predict outputs. To capture epistemic uncertainty, we adopted the Probabilistic Ensembles with Trajectory Sampling algorithm (PETS) [11] to aggregate all DNNs into a virtual ensemble environment. We use the same random sampling method to train DNNs by sampling four-tuples from the historical replay buffer. All DNNs generate predicted outputs represented by Gaussian probability distributions, and their results are weighted to produce a final prediction. This prediction can be described as:

$$P(p_{t+1}, r_t | p_t, a_t, \theta) = \frac{1}{n} \sum_{i=1}^{n} P(p_{t+1}, r_t | p_t, a_t, \theta_i) \tag{12}$$

Where n represents the number of DNNs in VEE. Taking into account the training speed, GPU memory limit, and VEE's prediction accuracy, we use 6 identical DNNs to construct VEE and adopt the average of the probabilities from 6 DNNs as the model prediction. By taking the weighted average of all DNNs, we can alleviate the cognitive defects of a single DNN due to sampling randomness in specific areas.

3) Determine the path transition from the predicted distribution: Given an input (p_t, a_t), DNNs output the potential path transitions and the corresponding rewards, denoted as (p_{t+1}, r_t), with a Gaussian probability distribution. However, in DGF, mutating the seed will only cause a specific path transition. Thus, we use a random sampling method to determine the next path p_{t+1} based on the predicted probabilities of different path transitions.

C. Reinforcement Learning for Fuzzing Model

To steer the fuzzer toward the target site more efficiently, namely through the high-reward path transition sequences, we need to learn a policy to select actions to maximize sequence rewards. Due to the huge number of paths, actions, and path transitions in the path transition model, it is difficult to use traditional mathematical methods such as Dynamic Programming (DP) [3] to learn the policy and calculate the expected sequence rewards. Therefore, we develop the Reinforcement Learning for Fuzzing (RLF) model, which is based on the reinforcement learning algorithm Soft Actor-Critic (SAC) [15].

1) The design of RLF model: As Fig. 2 shows, following the structure of SAC, the RLF model consists of a *Actor network*, a *Q-Critic network*, and a *V-Critic network*. During the training process, we train the Q-Critic network to evaluate the expected sequence reward and train the V-Critic network to evaluate the transition value of each path. With the expected sequence rewards and transition values, the Actor network is trained to optimize the policy to increase the probability of selecting actions with high expected sequence rewards.

2) Training data processing and collecting: we treat the path transition model as the environment in reinforcement learning and map the *fuzzer*, *path*, *action*, *path transition*, and *reward* in the path transition model to *agent*, *state*, *action*, *state transition*, and *reward* in reinforcement learning. Moreover, RLF reuses VEE's encoding method for path, action, and reward. Since we combine historical path transitions and predicted path transitions to train RLF to give the RLF model foresight, we collect historical path transitions and predicted path transitions in different ways.

Collecting historical path transitions. Since the environment of DGF is different from that of the traditional reinforcement learning process where state transitions are serially generated and represented as $(s_0, a_0, s_1, a_1, ..., a_{n-1}, s_n)$, the training data processing of RLF is different from that of the traditional reinforcement learning (e.g., SAC). In DGF, the fuzzer might mutate one seed multiple times, resulting in different path transitions from the same path. This means that the fuzzer may stay on a certain path and take different actions to cause different path transitions. Therefore, it is not feasible for the RLF model to obtain a complete path transition sequence and evaluate the current policy's effectiveness by calculating its sequence reward within a short period (e.g., several seconds). Based on this consideration, in each fuzzing cycle, the fuzzer selects actions according to the policy to cause path transitions. The historical path transitions are stored in the historical reply buffer and loaded by the RLF model at the end of the fuzzing cycle to train RLF during the next fuzzing cycle.

Collecting predicted path transitions. We use VEE to imitate the path transition model and employ the k-step branch rollout strategy to obtain predicted path transitions that have not yet occurred in the path transition model. In the k-step branch rollout strategy, the RLF model is regarded as the agent and it selects a sequence of actions at each path according to the initial policy to cause k path transitions, generating a new k-length path transition sequence. We use Fig. 3 to illustrate the process of the k-step branch rollout strategy. Suppose $(p_0, a_0, p_1, a_1 ... p_i, a_i, ... a_{n-1}, p_n)$ is a historical path transition sequence in the path transition model. We take

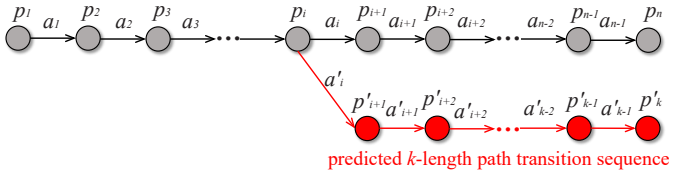


Fig. 3. k-step branch rollout strategy.

p_i as the starting point and use RLF's policy π to select a sequence of actions $a'_i, a'_{i+1}, ..., a'_{i+k-1}$ to cause k path transitions, generating a new k-length path transition sequence represented as $(p_i, a'_i, r_i, p'_{i+1})$, $(p'_{i+1}, a'_{i+1}, r'_{i+1}, p'_{i+2})$, ... , $(p'_{i+k-1}, a'_{i+k-1}, r'_{i+k-1}, p'_{i+k})$. Here, k is a hyperparameter that would impact the accuracy of VEE's predictions and the foresight of the RLF model's designed policy.

The predictions may need to cover multiple test cases during the k-step branch rollout process. Across different test cases, the same actions defined by the locations of bytes may be *misaligned* (i.e., the location of the mutated bytes in one testcase might be misaligned with the mutated bytes in other test cases owing to the different testcase length). To handle the misalignment, we use $mutation\_ID \wedge (path\_ID >> 2)$ to differentiate between different intermediate testcases generated during the process. For example, for testcase1 and testcase2 that cover the same path, we still assign them different IDs, so that we can distinguish mutations on different intermediate testcases. In this way, even if the same byte of testcase1 and testcase2 is mutated, we do not consider the probabilities and rewards associated with the path transitions caused by these two mutations to be the same.

3) Training of the RLF model: The RLF model combines historical path transitions and predicted path transitions to train the Actor network, Q-Critic network, and V-Critic network. In DGF, we aim for the fuzzer to adopt actions with high expected sequence rewards while also exploring different actions to cause new path transitions. Therefore, in RLF, we apply `entropy` to measure the randomness of selecting actions. Assuming we select actions in state s_t based on the policy π, and the probabilities of selecting actions follow a distribution denoted as $\pi(\cdot|s_t)$, the entropy of the action is calculated as:

$$H(\pi(\cdot|s_t)) = \mathop{\mathbb{E}}_{a\sim\pi(\cdot|s_t)}[-\log \pi(\cdot|s_t)(a)] \tag{13}$$

In the RLF model, the objective of the Actor network is to learn a policy π^* that maximizes the reward and entropy.

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[\sum_t \underbrace{r(s_t, a_t, s_{t+1})}_{\text{reward}} + \alpha \underbrace{H(\pi(\cdot \mid s_t))}_{\text{entropy}}] \tag{14}$$

Where α is the coefficient that balances exploration and exploitation, as proposed in SAC. Then, we construct the objective function for the Q-Critic network and the objective function for V-Critic network to train the parameters ω of the Q-Critic network and the parameters ϕ of the V-Critic network.

$$J_{v^\pi}(\phi) = \frac{1}{2}(\gamma r_t + V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t))^2 \tag{15}$$

$$J_{Q^\pi}(\omega) = (r_t + \gamma Q^\pi_\omega(s_{t+1}, a_{t+1}) - Q^\pi_\omega(s_t, a_t))^2 \tag{16}$$

For Actor network, we train the parameters σ of the Actor network to maximize the expected sequence rewards of actions, so as to maximize the sequence rewards.

$$J_\pi(\sigma) = \max_\sigma Q(s_t, \pi_\sigma(\cdot|s_t)) \tag{17}$$

By minimizing these three objective functions, we train the parameters in the three networks to compute the expected sequence rewards, and transition values, and design the policy for selecting actions that can maximize the sequence rewards. The well-trained RLF model provides two types of optimization information to the FO component: (1) the estimated expected sequence rewards and estimated transition value, and (2) the policy for selecting actions in each path.

D. Optimize Fuzzing Strategies Based on Action Group

To guide the fuzzer to exercise the optimal path transition sequences with the highest sequence rewards, we need to optimize fuzzing strategies based on the feedback information from the RLF model. In recent years, state-of-the-art techniques have been proposed to optimize a single fuzzing strategy, such as seed selection [6], mutator schedule [31], and mutation location selection [45]. However, optimizing a single fuzzing strategy may not significantly guide the fuzzer toward the optimal path transition sequences. Therefore, we propose the concept of the action group that is composed of five elements and attempt to comprehensively optimize multiple fuzzing strategies. Besides, we propose the Multi-elements Particle Swarm Optimization (MPSO) algorithm to optimize the elements in the action group simultaneously.

1) The concept of action group: We define the action group as a tuple consisting of five elements.

Seed-selection (denoted as **SS**). Representing the probability of a seed being selected to fuzz by the fuzzer.

Seed-energy (denoted as **SE**). Representing the energy assigned to the seed, which determines the number of mutations that can be applied to the seed during the havoc stage.

Havoc-round (denoted as **HR**). Representing the number of looping rounds used to select different mutators and bytes during the havoc stage. All of the selected mutators and bytes are integrated into a single havoc action. The value of the havoc-round may be 2, 4, 8, 16, 32, 64, or 128.

Mutator (denoted as **MT**). Representing the mutator selected to mutate the seed. Similar to AFL [24], DeepGo preserves 16 different types of mutators.

Location (denoted as **LC**). Representing the mutation location of the seed that is selected to mutate.

Each action group is represented as a 27-dimensional vector consisting of the 5 elements. As shown in Fig. 4, SS and SE are both 1-dimensional vectors. The value of **SS** represents the probability within a range of [0, 1]. The fuzzer would select seeds to fuzz based on **SS**. The value of **SE** represents the energy assigned to the seed and the fuzzer calculates the mutation times of seeds based on **SE**. **HR** is a 7-dimensional vector, where each dimension represents the probability of selecting one of the seven different havoc-round values (i.e., 2, 4, 8, 16, 32, 64, and 128). The fuzzer samples the number of looping rounds used to select different mutators and mutation locations during the havoc stage based on **HR**. **MT** is a 16-dimensional vector, where each dimension represents the probability of selecting one of the 16 different types of mutators. **LC** is a 2-dimensional vector, where the first dimension represents the probability of selecting the optimal locations, and the second

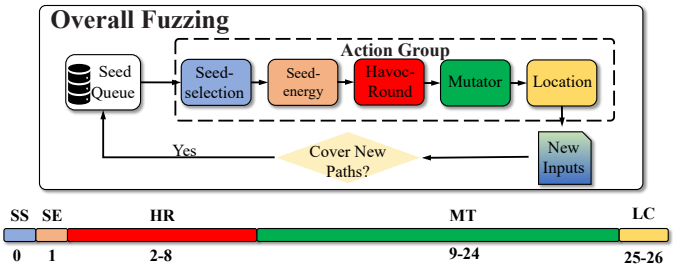


Fig. 4. The layout of elements in the action group.

dimension represents the probability of selecting common locations. We classify the mutation locations of seeds into two categories: *optimal locations* and *common locations*. The optimal locations include the mutation locations selected by the RLF model's policy with a probability of greater than 80%, while common locations include all other mutation locations. Based on the **MT** and **LC**, the fuzzer samples the mutators and the type of mutation location. The fuzzer constantly mutates seeds to generate new inputs according to the five elements in the action group. As Fig. 4 shows, we represent each element as a vector and concatenate the five vectors into one vector to represent the action group of a seed.

2) Multi-elements Particle Swarm Optimization algorithm: To optimize the five elements in the action group simultaneously, we treat each action group as a particle represented by a 27-dimensional vector and view the optimization of the action group as a multi-element optimization problem. We propose a Multi-elements Particle Swarm Optimization (i.e., MPSO) algorithm to realize the optimization. The optimization of the action group can guide the fuzzer toward the desired path transition sequences with high sequence rewards.

As we have introduced in Section II, the PSO algorithm uses massless particles to simulate birds in a flock. Each particle searches for the optimal solution individually and records it as the local best value. The local best values are then shared among the particles in the entire swarm to find the global best value as the optimal solution. These particles have two primary attributes: *velocity* and *position*. Velocity represents the speed of movement, and position represents the direction of movement. All particles in the swarm would constantly adjust their velocity and position based on the local best value and the global best value shared in the entire swarm:

$$v_i = w \times v_i + r \times (lbest_i - x_i) + r \times (gbest_i - x_i) \quad (18)$$

$$x_i = x_i + v_i \quad (19)$$

Where v_i represents the velocity of the i^{th} particle, x_i represents the position of the i^{th} particle, r represents the random displacement weight within the range of [0,1], $lbest_i$ represents the local best position found by the i^{th} particle and $gbest_i$ represents the global best position found by all particles. The inertia factor ω is a non-negative value, where a larger value results in stronger global optimization and weaker local optimization ability in the PSO algorithm. In this paper, we apply the Linearly Decreasing Inertia Weight (LDIW) method [46] to set the value of ω as:

$$\omega = \omega_{ini} - (\omega_{ini} - \omega_{end}) \times (g/G) \quad (20)$$

Where ω_{ini} and ω_{end} denote the initial and final inertia values, respectively. Following the LDIW, ω_{ini} and ω_{end} are typically

set to 0.4 and 0.9. g denotes the number of iterations completed by the current particle, which is equal to the number of mutations performed by the fuzzer on the corresponding seed. G denotes the maximum iteration number, which is equal to the total mutation times calculated according to the seed energy. In DeepGo, the mutation times of seeds are determined by the energy assigned to the seeds.

Local best position and local efficiency. In the MPSO algorithm, each particle has its own local best position `lbest` and local efficiency eff_{local}. Given a particle, position x_1 is deemed superior to position x_2 only if the local efficiency obtained by the particle at x_1 is greater than that at x_2. We use local efficiency to quantify the mutation efficiency of fuzzing strategies composed of the five elements for one specific particle. The local efficiency is calculated according to the averaged expected sequence rewards of all the mutations taken by the fuzzer in the path corresponding to the particle. In DGF, each mutation on the seed would cause a path transition from path p to path p'. According to Equation (5), in one path transition, we use $r + \gamma V_\pi^t(p')$ to evaluate the expected sequence rewards. Based on this, we calculate local efficiency.

$$eff_{local} = \frac{\sum_{i=1}^{g} r_i + \gamma V_\pi^t(p')}{g} \tag{21}$$

Where $V_\pi^t(p')$ denotes transition value of path p' covered by the i^{th} mutation.

Global best position and global efficiency. We use global efficiency to quantify the fuzzing efficiency of the fuzzer. Since the global efficiency of the fuzzer depends on the local efficiency of different particles and the relationship between particles, we determine the global position of all particles based on the fuzzing efficiency within a fuzzing cycle. A particle is currently in the global best position (`gbest`) only if the fuzzing efficiency of the fuzzer is higher than that of any other position. We use the average sequence rewards during the fuzzing cycle to evaluate the global efficiency:

$$eff_{global} = \frac{\sum_{j=1}^{U} \sum_{i=1}^{g_j} r_i + \gamma V_\pi^t(p'_j)}{\sum_{j=1}^{U} g_j} \tag{22}$$

Where eff_{global} denotes the global efficiency, s_j denotes j^{th} seed, g_j denotes the total number of mutations for s_j, and U denotes the number of seeds that have been fuzzed in the current fuzzing cycle.

During the fuzzing process, we calculate the local efficiency and global efficiency of particles according to Equations (21) and (22), and record the `lbest` and `gbest`. We update the particles' spatial positions according to Equations (18), (19), and (20), moving all particles towards the direction of `lbest` and `gbest`. Through this approach, we optimize the action groups of all seeds to guide the fuzzer toward the target via the optimal path transition sequences.

The process of MPSO is shown in Algorithm 1, where s denotes the seed, p denotes the particle, $\Omega_{(s,p)}$ denotes the set containing all seeds and their corresponding particles. The function $Prob\_Sel_s$ determines whether to fuzz the seed based on the **SS**. mn_i denotes the number of mutation times of the seed, and the function `Cal_MN` calculates the number of mutation times for the seed based on **SE**. The functions $Prob\_Sel_h$, $Prob\_Sel_l$, and $Prob\_Sel_m$ probabilistically select the corresponding values of hr for **HR**, lc for mutation location based on **LC**, and mt for mutator based on **MT**.

Algorithm 1 MPSO Algorithm

Input: $\Omega_{(s,p)}$
Output: U_s, $\Omega_{(s,p')}$

```
1: Initial(Ω_(s,p))
2: while fuzzing do
3:     for (s_i, p_i) in Ω_(s,p) do
4:         if Prob_Sel_s(p_i(SS)) == True then
5:             mn_i ←Cal_MN(p_i(SE))
6:             for j in mn_i do
7:                 hr_j ← $Prob_Sel_h(p_i(HR)), hm_j ←<>
8:                 for k in hr_j do
9:                     lc_k ← Prob_Sel_l(p_i(LC)),
10:                    mt_k ← Prob_Sel_m(p_i(MT)),
11:                    hm_j ← hm_j ∪ (lc_k, mt_k)
12:                end for
13:                new_input = Mutate(hm_j, s_i)
14:                eff_local, eff_global = Cal_eff(s_i, new_input)
15:                Update(lbest, gbest, p_i)
16:            end for
17:        end if
18:    end for
19: end while
```

At first, we initialize the five elements in all the particles (Line 1). For **SS** and **SE**, according to the seeds' characteristics (e.g., the bitmap size and execution speed), AFL [24] calculates the probabilities of seeds being fuzzed and the energy assigned to seeds. We use the probabilities and the seed energy obtained by AFL's method as the initial value of **SS** and **SE**. For **HR**, **LC**, and **MT**, we use the average probability as the initial value of their spatial position (e.g., 1/16 for each mutator). During the fuzzing process, the fuzzer selects seeds to fuzz according to **SS** (Line 4), calculates the mutation times according to **SE** (Line 5), selects the values of **HR** (i.e., hr_j) (Line 7), selects the seed bytes for mutation (i.e., lc_k) according to **LC** (Line 9), and selects the mutators (i.e., mt_k) according to **MT** (Line 10). In each hr_j cycle, the mt_k and lc_k will be combined into the havoc mutation (i.e., hm_j) (Line 11), and the fuzzer will use hm_j to mutate the seed s_i and generate a new_input (Line 13). Then, MPSO would calculate the local efficiency and global efficiency (Line 14), and update `lbest`, `gbest` and the position of the particle p_i (Line 15). Notably, the value of all dimensions of the particle will constantly change according to Equation (18), (19), and (20) to update the spatial position of the particle, allowing the particle to move to the `lbest` and `gbest`. For instance, if one particle has low local efficiency which results in reducing the global efficiency of the fuzzer, according to the Equation (18), (19) and (20), the **SS** and the **SE** will gradually decrease during the process of MPSO. Once the FO component first receives the feedback information from the RLF model, it will launch the MPSO algorithm until the end of fuzzing.

V. Implementation

The implementation of DeepGo mainly consists of three components: the fuzzer, the VEE, and the RLF model. For the static analyzer in the fuzzer, we leverage LLVM 11.0.

We use the LLVM IR to instrument the program and obtain information about basic-block-level distance, sibling branches, etc. The fuzzer is built on AFLGo, with 2100 lines of C code, and the VEE and the RLF model are implemented with about 1300 lines of Python code.

In detail, the DNNs of VEE are implemented in Pytorch-1.13.0 with five fully connected layers. The hidden layer uses Swish as its activation function. The DNNs are trained for 500 epochs and we use the tensorboard tool to automatically monitor the loss values to determine if they converge to a small value. If the loss values of VEE and RLF have converged, the DNNs will automatically stop training. The networks in the RLF model consist of three fully-connected layers. For the hyperparameters of the RLF model, referring to the learning rate settings for Q-Critic network, V-Critic network, and Actor network in SAC, we set them to 0.005 to ensure the learning efficiency and convergence of the RLF model (in the experimental process, all three networks can converge quickly).

Notably, when using DeepGo to test programs, the fuzzing process, the training of the models, and the prediction of the path transition are performed concurrently. We use an extra GPU to train the VEE and RLF models based on the information collected throughout the fuzzing process. All the time spent on training and prediction of the VEE and RLF models is counted in the time budget (indicated as wall clock time) for the fuzzing process.

VI. Evaluation

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

RQ1: What about the performance of DeepGo in terms of reaching the target code locations?

RQ2: What about the performance of DeepGo in terms of exposing the known vulnerabilities?

RQ3: What about VEE's performance in predicting probabilities and rewards of path transitions?

RQ4: Can the RLF model and FO component guide the fuzzer to path transition sequences with high sequence rewards?

RQ5: How do the VEE, RLF, and FO components contribute to the overall performance of DeepGo?

A. Evaluation Setup

Evaluation Criteria. We 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 that can reach the 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. An observed crash indicates that the fuzzer has successfully exposed the vulnerability.

Evaluation Benchmarks. We selected two datasets that are widely used by state-of-the-art DGF techniques (e.g., WindRanger [12], BEACON [18]).

(1) UniBench [27] 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++ [13] 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.

(2) AFLGo testsuite [6] was proposed in AFLGo's paper and website to evaluate the directness of DGF. It had been used as a benchmark by the state-of-the-art DGF techniques to verify the bug reproduction capabilities. To answer RQ2, we selected the AFLGo testsuite as the benchmark to verify the capability of exposing known vulnerabilities.

Baselines. In our evaluation, we compared DeepGo with the state-of-the-art directed greybox fuzzers that were publicly available at the time of writing this paper, including WindRanger, BEACON, ParmeSan, and AFLGo.

Experiment Settings. We conducted the experiments on the machine equipped with Intel(R) Xeon(R) Gold 6133 CPU @ 2.50GHz 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 from UniBench and AFLGo testsuite, we used the seeds in BenchMark's recommended seed corpus as initial seeds. For experimental results analysis, we utilize the Mann-Whitney U test (p-value) to measure the statistical significance. In addition, we use the Vargha-Delaney statistic ($\hat{A}_{12}$) [1] to measure the probability of one technique performing better than another.

B. Reaching Target Sites

To answer RQ1, we tested the 20 programs from UniBench, with a total of 80 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 III in the Appendix. In Table III, 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, DeepGo can reach the most (73/80) target sites compared to AFLGo (22/80), BEACON (11/80), WindRanger (19/80), and ParmeSan (9/80) within the time budget. Moreover, on most of the target sites

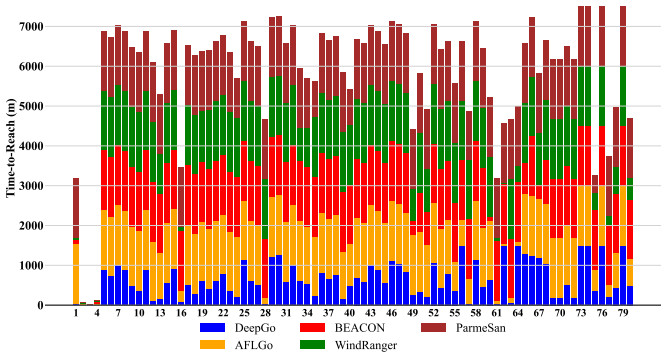


Fig. 5. TTR of DeepGo and baseline fuzzers on the UniBench.

(67/80), DeepGo outperforms all other fuzzers and achieves the shortest TTR. In terms of mean TTR of reaching the target sites, DeepGo demonstrates 3.23×, 1.72×, 1.81×, and 4.83× speedup compared to AFLGo, BEACON, WindRanger, and ParmeSan, respectively. We conducted both the Mann-Whitney U test (p-value) and the Vargha-Delaney test ($\hat{A}_{12}$) that all the p-values are less than 0.05, and the mean $\hat{A}_{12}$ against AFLGo, BEACON, WindRanger, and ParmeSan are 0.86, 0.81, 0.83, and 0.89, respectively. Based on the above analysis, we can conclude that **DeepGo 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. 5, the x-axis represents the target site ID (1-80), 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 min in Fig. 5. From the figure, we can clearly see that the blue bars are much shorter than the other bars, which means that DeepGo can reach most of the target sites faster than the baseline fuzzers.

C. 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 I. As Table I shows, among the 20 vulnerabilities, DeepGo (19) exposed the most compared to AFLGo (14), BEACON (13), WindRanger (16), and ParmeSan (14). Besides, on most of the target sites (14/20), DeepGo outperformed all the baseline fuzzers and achieved the shortest TTE. With respect to the mean TTE of exposing vulnerabilities, DeepGo demonstrated 2.61×, 3.32×, 2.43× and 2.53× speedup compared to AFLGo, BEACON, WindRanger, and ParmeSan, respectively. All p-values were less than 0.05, and the mean $\hat{A}12$ against AFLGo, BEACON, WindRanger, and ParmeSan were 0.79, 0.72, 0.75, and 0.81, respectively. Based on the above analysis, we can conclude that **DeepGo can expose known vulnerabilities faster than the baseline fuzzers**.

D. The effectiveness of VEE

To answer RQ3, we analyzed the predictions made by VEE when DeepGo tested the 20 programs from UniBench. Given

TABLE I. THE RESULTS OF TTE ON AFLGO TESTSUITE

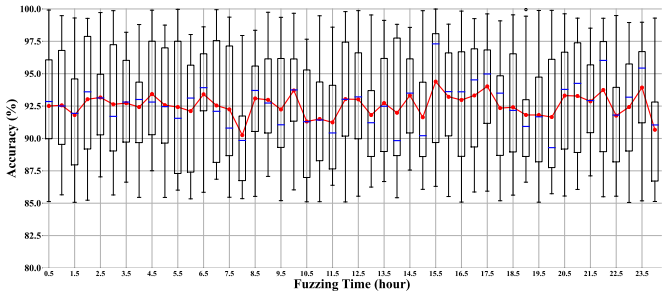
Prog.	CVE-ID	AFLGo	BEACON	WindRa	ParmeS	DeepGo
$\text{binutils}_{2.26}$	2016-4487	2.33m	0.63m	1.21m	0.95m	1.34m
	2016-4488	4.23m	32.1m	3.32m	2.62m	2.69m
	2016-4489	3.36m	2.98m	5.88m	2.31m	1.23m
	2016-4490	1.15m	2.35m	2.63m	0.82m	1.97m
	2016-4491	448m	258m	298m	212m	129m
	2016-4492	10.8m	43.6m	7.47m	4.33m	6.94m
	2016-6131	348m	292m	318m	244m	68.1m
$\text{libming}_{4.48}$	2018-8807	331m	267m	171m	301m	101m
	2018-8962	234m	163m	121m	198m	54.8m
	2018-11095	T.O.	914m	1311m	T.O.	812m
	2018-11225	T.O.	438m	996m	T.O.	128m
$\text{LibPNG}_{1.5.1}$	2011-2501	10.2m	N/A	7.81m	4.53m	3.46m
	2011-3328	69.1m	N/A	49.3m	193m	17.5m
	2015-8540	0.88m	N/A	0.96m	3.41m	5.65m
$\text{xmllint}_{2.9.4}$	2017-9047	T.O.	T.O.	T.O.	T.O.	783m
	2017-9048	T.O.	T.O.	T.O.	T.O.	1389m
	2017-9049	T.O.	T.O.	T.O.	T.O.	T.O.
	2017-9050	T.O.	T.O.	T.O.	T.O.	911m
$\text{Lrzip}_{0.631}$	2017-8846	348m	156m	223m	466m	131m
	2018-11496	201m	98.1m	169m	126m	78.9m
speedup		**2.61×**	**3.32×**	**2.43×**	**2.53×**	**-**
mean $\hat{A}_{12}$		**0.79**	**0.72**	**0.75**	**0.81**	**-**
mean p-values		0.018	0.032	0.026	0.011	**-**

an input (p_t, a_t), VEE would predict the outputs (p_{t+1}, r_t) with varying probabilities and rewards (i.e., predicted probabilities and predicted rewards). Meanwhile, during the fuzzing process, the fuzzer takes action a_t in path p_t would cause path transitions with varying probabilities and rewards (i.e., real probabilities and real rewards.). To analyze the prediction accuracy of VEE, we defined the metric **Accuracy** as:

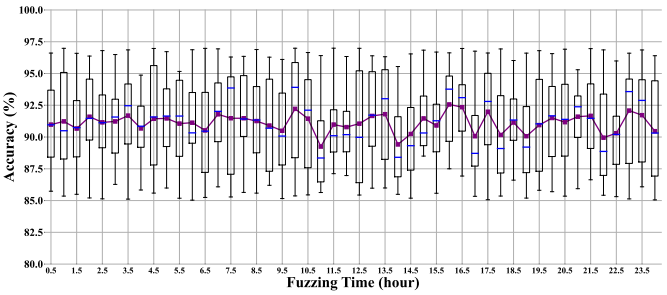
$$Accuracy = 1 - \frac{|real\_value - predicted\_value|}{|real\_value|} \quad (23)$$

Where $real\_value$ denotes the real probability or real reward, $predicted\_value$ denotes the predicted probability or predicted reward. The higher value of Accuracy indicates the higher accuracy of the prediction. During the testing of each program from UniBench, we calculated the two accuracies of each path transition every 30 minutes. Since DeepGo repeated the testing of the 20 programs five times, at each time point, we calculated the average accuracy of predicted probability (**AAPP**) and the average accuracy of predicted reward (**AAPR**) of all path transitions of all the programs, to evaluate the prediction accuracy of VEE based on Equation 23.

To reflect the results straightforward, we use the line chart to visualize the results. In Fig. 6, the x-axis represents the fuzzing time (from 0-hour to 24-hour) and the y-axis represents the accuracies of AAPP and AAPR which are both within the range of [0, 1]. As shown in Fig. 6, from 0.5-hour to 24-hour, the accuracies of AAPP and AAPR are all greater than 80%, and the means of AAPP and AAPR at the 48 time points are 92.57% and 91.10% respectively. Meanwhile, we used box charts to show the AAPPs and AAPRs on different programs at different time points. The blue line represents the median, and the length of the box represents the distribution range. From the box charts, we can see that the AAPPs and AAPRs do not vary significantly ($\leq 15\%$) among different programs, and maintain high value ($\geq 80\%$) at each time point. This suggests that VEE can limit the deviation when imitating the



(a) AAPP



(b) AAPR

Fig. 6. The accuracies of rewards and probabilities predicted by VEE on programs from UniBench.

path transition model and predict the probabilities and rewards of path transitions with high accuracy.

E. The effectiveness of the RLF model and FO component

To answer RQ4, we collected rewards of path transitions that occurred during the fuzzing process, to calculate the average reward of path transitions in the path transition sequence to show whether the RLF model and the FO components can guide the fuzzer to exercise the optimal path transition sequences with high sequence rewards. Since DeepGo's optimizations on the fuzzing strategies rely on the RLF model and the FO component, removing either of them will render DeepGo's optimizations ineffective, thus, we removed both of them, thereby forming a new fuzzer called **DeepGo-r**. We used DeepGo-r and DeepGo to test the 20 programs from UniBench and calculated the reward of all path transitions for all the programs every 30 minutes. Then, at each time point, we obtain the average reward (i.e., **AR**) of the 20 programs.

To reflect the results in a straight way, we use the line chart to visualize the results. In Fig. 7, the x-axis represents the fuzzing time (from 0.5-hour to 24-hour) and the y-axis represents the values of AR, which is within the range of [-1, -1]. As shown in Fig. 7, from 0.5-hour to 24-hour, the overall

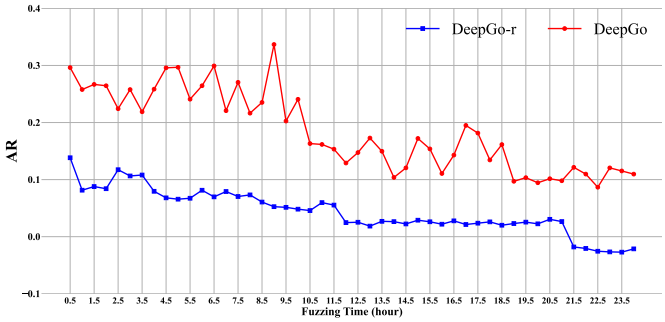


Fig. 7. The value of ARs on programs from UniBench.

TABLE II. INTERMEDIATE DATA ANALYSIS USING DIFFERENT SEEDS

Fuzzers	**RSeed**	**PRseed**	**RPath**
AFLGo	2311	52.4%	58
BEACON	312	79.8%	23
WindRanger	2532	64%	41
ParmeSan	1463	43.8%	19
DeepGo	2788	72.6%	261

trend of AR is decreasing, which is consistent with the fact that the fuzzer is getting difficult to find new seeds as the fuzzing proceeds. DeepGo's AR was significantly higher than that of DeepGo-r. At each time point, DeepGo's AR was, on average, 4.26× higher than DeepGo-r's AR. This indicates that the RLF model and the FO component can guide the fuzzer to exercise the optimal path transition sequences and reach the target site more quickly.

F. Ablation study

To answer RQ5, we conducted the ablation study to demonstrate the impact of VEE, RLF, and FO on DeepGo's performance. To demonstrate that VEE, the RLF model, and the FO component can enhance directedness, we remove VEE from DeepGo and form a new tool **DeepGo-V** and also run DeepGo-v and DeepGo-r on UniBench for the TTR experiment. Detailed results of DeepGo-v and DeepGo-r are listed in Table III in the Appendix. According to the TTR results, DeepGo (73/80) can reach much more target sites than DeepGo-v (32/80) and DeepGo-r (18/80), respectively. Moreover, DeepGo outperforms DeepGo-v and DeepGo-r by 2.05× and 3.72×, respectively, in the average TTR of reaching the target sites. The p-values are 0.013 and 0.006, and the mean $\hat{A}_{12}$ against DeepGo-v and DeepGo-r is 0.83 and 0.90, respectively. These results demonstrate that VEE, RLF, and the FO component have significant impacts on reducing TTR.

We also use intermediate data to prove that DeepGo can avoid the infeasible and hard-to-execute paths. Firstly, we consider a seed to be a reachable seed (i.e., **Rseed**) if its execution path covers reachable basic blocks (BB distance ≥ 0). We calculate the average number of reachable seeds generated by different fuzzers when testing different programs, as well as 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 hard-to-execute paths. Secondly, we measure the total number of paths taken by the fuzzer to reach target sites (i.e., reached paths, **Rpath**) during testing 20 programs from UniBench. For the same target site, different paths can yield different testing results. By counting the total number of reached paths, we can determine if DeepGo can discover more paths to reach the target site compared to other fuzzers, thus enabling more diverse testing of the target site. From the results in Table II, we can obtain two conclusions. Firstly, by comparing the Rseeds and Rpaths of all fuzzers, we can see that DeepGo can generate more reachable seeds within the same time budget. Secondly, although BEACON has a higher Rpath, it has the lowest Rseed among all fuzzers due to pruning some reachable paths to the target site. Apart from BEACON, DeepGo has the highest PRseed among all fuzzers. **These three intermediate metrics suggest that DeepGo can generate optimal and viable paths to the target site by avoiding infeasible and hard-to-execute paths.**

G. Case study of DeepGo's performance

To show why DeepGo can reach the target sites faster, we used the target site `get_audio.c`:1605 in $\texttt{lame}_{3.99.5}$ as an example to conduct a case study. As listing 1 shows, Line 24 is set as the target site. Since the path constraints at Line 2, Line 16, and Line 23 are all hard-to-satisfy path constraints, AFLGo, Windranger failed to reach the target site within the 24-hour time budget while DeepGo reached the target site (Line 24) at the 282^{nd} minute. As for AFLGo and WindRanger, AFLGo could reach Line 11-12, and WindRanger could reach Line 17. However, neither of them could satisfy the path constraints at Line 2, Line 9-10, Line 16, and Line 23 simultaneously to reach the target site. In Listing 1, conditions in different code locations are associated with different seed bytes (e.g., the values of $global.snd\_file$ in Line 2 and $global\_reader.input\_format$ in Line 11 are determined by different seed bytes) and mutating the relevant bytes would cause different path transitions and rewards. Since AFLGo and WindRanger cannot predict the path transitions and rewards, they cannot avoid using actions that cause low-reward path transitions. Consequently, most of the new inputs generated by AFLGo and WindRanger could not cover Line 11-12 and Line 18, and thus failed to reach the target site (Line 24).

To identify such associations, during the testing process, the associations between the actions (mutation bytes) and the path transitions would be recorded. By querying the paths containing certain code locations, we can obtain the corresponding actions associated with them, thereby obtaining the correspondence between code locations and actions. Specifically, DeepGo can record the mappings between (path, action) and (next_path, reward) to train VEE. If the path transitions caused by mutating a certain byte can guide DeepGo to cover interesting code locations (e.g., code locations closer to the target sites), the path transition and its corresponding action (mutation bytes) will be given a high reward. By querying the rewards of different path transitions, we can determine which bytes to mutate for interesting code locations.

To prove the above analysis, we calculated the expected sequence rewards of taking all actions (i.e., mutating all bytes) and the selection probabilities of the RLF for each action after DeepGo reached Line 12 and presented the results in Fig. 8. In Fig. 8, the x-axis represents the encoding of actions, and the y-axis represents the normalized value of the expected sequence rewards and selection probabilities of actions. `Line 2`, `Line 9-10`, and `Line 16` represent three actions with encodings of 0.226, 0.41, and 0.618, respectively, which mutate the bytes related to the constraint condition in

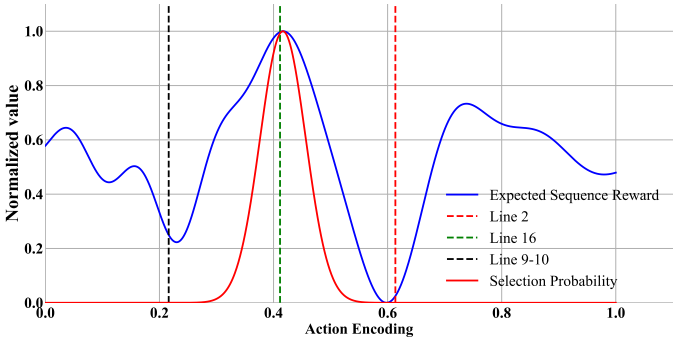


Fig. 8. The expected sequence rewards and selection probability of actions.

```
1  int init_infile(){
2   if (global.snd_file == 0) {
3    global.music_in = open_wave_file(gfp, inPath);
4   }
5   else
6     ...
7  }
8  static FILE * open_wave_file(){
9   if (global_reader.input_format != sf_raw &&
10          (global_reader.input_format != sf_ogg){
11   global_reader.input_format =
12          parse_file_header(gfp,musicin);
13  }
14 }
15 static int parse_file_header(){
16  if (type == IFF_ID_FORM) {
17   int const ret = parse_aiff_header(gfp, sf);
18  }
19  else
20    ...
21 }
22 static int parse_aiff_header(){
23  if (dataType == IFF_ID_2CBE) {
24   global.pcmswapbytes = !global_reader.swapbytes;
25  }
26  else
27   ...
28 }
```

Listing 1. Example of the target site *get_audio.c:1605* in $\texttt{lame}_{3.99.5}$

Line 2, Line 9-10, and Line 16, respectively. The blue line represents the distribution of the expected sequence rewards of actions, and the red line represents the Gaussian probability distribution of the RLF model for selecting actions. As the blue line shows, the normalized values of the expected sequence rewards of these three actions are 0.232, 0.994, and 0.061, respectively. It means that if the fuzzer takes actions at `Line 2` and `Line 9-10`, the caused path transitions will make the fuzzer further away from the target site. As for the red line, the Gaussian probabilities of `Line 2` and `Line 9-10` are close to 0, while that of `Line 16` is close to 1. Thus, when selecting actions using Gaussian sampling, the RLF model tends to select the actions near `Line 16` rather than selecting `Line 2` and `Line 9-10`. By this means, DeepGo selects actions with high expected sequence rewards with a high probability, which helps avoid low-reward path transitions that go further away from the target site.

VII. Discussion

Settings of hyperparameters. The hyperparameter γ and k both affect DeepGo's TTR. Firstly, in the training of the RLF model, the hyperparameter γ is used to balance the influence of current rewards and subsequent rewards on the transition value and expected sequence rewards of the current state and its actions. Only focusing on current rewards, RLF may get stuck in local optima. On the contrary, overly emphasizing subsequent rewards may cause low-reward path transitions for the current path, thereby reducing the directed efficiency. Thus, the setting of the hyperparameter γ would influence the Q-values, V-values, and policy of the RLF model, ultimately affecting DeepGo's TTR. Secondly, in the k-step branch rollout strategy, the hyperparameter k is used to generate the k-length path transition sequence. As k increases, VEE is able to predict more path transitions, enabling RLF to have more foresight in designing policies. However, a high value of k

may lead to a decrease in the accuracy of VEE's predicted path transitions and rewards, misleading the RLF's policy to low-reward path transition sequences. Therefore, the setting of the hyperparameter k would affect the prediction accuracy of VEE and the foresight of RLF's policy, ultimately impacting the TTR of DeepGo.

To observe the impact of varying γ and k on DeepGo, we conducted experiments by setting γ to 0.9, 0.8, 0.7, 0.6, 0.5, 0.4, 0.3, 0.2, and 0.1, and setting k to 1, 2, 3, 4, 5, 6, 7, 8, and 9. We then utilized DeepGo with different hyperparameter configurations to test 20 programs from UniBench and recorded the mean TTR for each test case. In order to visually illustrate the impact of γ and k settings on DeepGo's TTR, we use a 3D chart to showcase the variations of TTR as γ and k change. According to Fig. 9, we can draw three conclusions. Firstly, the minimum TTR is achieved when γ is set to 0.8 and the k is set to 4, which is marked as a red point. Secondly, if the value of γ is between [0.5, 0.9], and the value of k is between [3, 5], the setting of γ and k has a relatively small impact on TTR (TTR changed within the range of [648, 688]). Thirdly, TTRs are higher when γ is greater than 0.5 compared to when γ is less than 0.5, and TTRs are higher when k is less than 5 compared to when k is greater than 5. This reflects the fact that since the length of path transition sequences in the fuzzing is generally less than 20 based on statistical fuzzing information, we should pay more attention to the influence of the current rewards of the path. Moreover, the setting of k should balance the prediction and the foresight with an appropriate value.

Selection of targets. When selecting targets for the evaluation, we ran AFL++ for 48 hours because we believe that DGF techniques can faster reach the predefined targets than CGF techniques [13], [24], [55], [56]. Therefore, we believe that even if some targets are reached within more than 24 hours but less than 48 hours by CGF, they are still likely to be reached by DGF within 24 hours. For example, in our evaluations, out of the 51 targets that took AFL++ more than 24 hours to reach, 45 of them were reached by one or more directed fuzzers within 24 hours. We use AFL++ rather than "regular" AFL to set the targets because AFL++ provides more comprehensive results, allowing for a detailed recording of the time required for CGF to reach different targets. With the information provided by AFL++, we can reproduce the code locations and the time cost to reach these locations, which is necessary for us to select the targets. In contrast, AFL does not provide such detailed information.

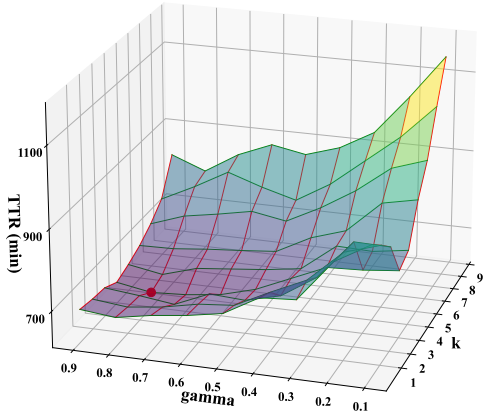


Fig. 9. The impact of hyperparameter settings on TTR.

VIII. Related Work

Directed Symbolic Execution (DSE) mostly relies on symbolic execution engines such as KLEE [7], KATCH [33], and BugRedux [2] to reach the target sites. Using program analysis and constraint solving, DSE can generate inputs that effectively penetrate through the path constraints toward the target sites. Although some state-of-the-art works, such as symcc [40], symqemu [39], and JigSaw [9], have been proposed to develop the symbolic execution, the heavyweight program analysis, path-explosion problem, and constraint solving of DSE still limit its scalability.

Directed Grey-box Fuzzing (DGF) calculates the distances between the seeds and pre-defined targets to prioritize the seeds closer to the targets, which casts reachability as an optimation problem to minimize the distance between the seeds and their targets. Based on AFLGo's idea, Hawkeye [8], LOLLY [29], Berry [28], UAFL [50], and CAFL [25] proposed the new fitness metrics such as trace similarity, and sequence similarity, to enhance directedness and detect hard to manifest vulnerabilities. FuzzGuard [59] filters out the unreachable seeds and BEACON [18] prunes the infeasible paths, which are effective methods to improve the efficiency of DGF. MC^2 [43] models DGF as an oracle-guided search problem to find a target-reaching input, which accelerates the speed of reaching the targets. FISHFUZZ [58] enables fuzzers to seamlessly scale among thousands of targets and prioritize seeds toward interesting locations, thus achieving more comprehensive program testing. SemFuzz [12], [54] analyzes the data-flow information and semantic information to generate valid input. Parmesan [37], V-Fuzz [37], and SAVIOR [10] utilize the sanitizers, such as ASAN [42] and UBSan [30], to label the potential buggy code as the target sites and steer the DGF to test the target sites. Hydiff [23], SAVIOR [10] and Badger [36] prioritize the seeds that may cause the specific program bug locations as the target sites, and then prioritize symbolic execution of the seeds that are reachable from more target sites. DrillerGO [22] and Berry [28] combine the precision of DSE and the scalability of DGF to mitigate their individual weaknesses. However, DGF still suffers from being difficult to penetrate through the hard-to-satisfy path constraints. DeepGo foresees critical execution information and predicts the optimal path. By combining the historical execution information and the predicted future execution information, DeepGo can intelligently generate the optimal and viable path to the target site. By avoiding the infeasible and hard-to-execute paths, the fuzzer can reach the target site more precisely and efficiently.

AI-Based Grey-box Fuzzing. Previous state-of-the-art works [14], [45], [48], [49], [51], [59] apply AI techniques to augment the greybox fuzzing techniques. Among these works, NEUZZ [45] and MTFUZZ [44] introduce gradient-descent based approaches to augment coverage-guided grey-box fuzzing by approximating the PUT's discrete branching behavior. AthenaTest [49] uses the local transformer-based networks to extract features of seeds from the corpus and generate test cases. DYNFuzz [57] and FuzzGuard [59] build models based on neural networks to predict whether the seeds are reachable to the target sites and filter out the unreachable seeds to enhance directedness. However, the existing AI-Based

Grey-box Fuzzing techniques only optimize the selection of mutated bytes or seeds. They cannot comprehensively optimize all fuzzing strategies, making the fuzzer less intelligent in generating optimal paths to reach the target sites.

IX. Conclusion

In this paper, we propose DeepGo, a predictive directed greybox fuzzer that can combine historical and predicted information to steer DGF to reach the target site via an optimal path. DeepGo constructs the Virtual Ensemble Environment, which uses DNNs to imitate the path transition model and predict the rewards of potential path transitions. Using the RLF model, DeepGo combines the historical and predicted path transitions to determine the path transition sequence with the highest sequence rewards to generate optimal paths. Based on the MPSO algorithm, DeepGo optimizes the action group and exercises the high-reward path transition sequence to realize the optimal path. DeepGo is evaluated on 100 target sites of 25 real-world programs from 2 datasets, the experiment results show that DeepGo outperforms the state-of-the-art directed fuzzers (AFLGo, BEACON, WindRanger, and ParmeSan) in reaching target sites and exposing known vulnerabilities. Moreover, DeepGo also shows high accuracy in predicting the path transitions that have not been taken yet.

Acknowledgment

This work is partially supported by the National Key Research and Development Program of China under Grant No. 2021YFB0300101, the National University of Defense Technology Research Project (ZK20-17, ZK20-09, ZK23-14), the National Natural Science Foundation China (62272472, 61902405, U22B2005, 61972412, 62306328), the HUNAN Province Natural Science Foundation (2021JJ40692), and the National High-level Personnel for Defense Technology Program (2017-JCJQ-ZQ-013).

References

[1] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "The non-stochastic multiarmed bandit problem," *SIAM Journal on Computing*, vol. 32, no. 1, pp. 48–77, 2002.

[2] P. Auer, N. Cesa-Bianchi, Y. Freund, and R. E. Schapire, "Gambling in a rigged casino: The adversarial multi-armed bandit problem," *Electron. Colloquium Comput. Complex.*, no. 68, 2000.

[3] L. A. Baxter, "Markov decision processes: Discrete stochastic dynamic programming," *Technometrics*, vol. 37, no. 3, pp. 353–353, 1995.

[4] R. Bellman, "A markovian decision process," *Indiana University Mathematics Journal*, vol. 6, no. 4, p. 15, 1957.

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

[6] M. Böhme. (2023) Directed greybox fuzzing with afl. https://github.com/ aflgo/aflgo.

[7] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," in *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. USENIX Association, 2008, pp. 209–224.

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

[9] J. Chen, J. Wang, C. Song, , and H. Yin, "Jigsaw: Efficient and scalable path constraints fuzzing," in *43rd IEEE Symposium on Security and Privacy (Oakland'22)*, May 2022.

[10] Y. Chen, P. Li, J. Xu, S. Guo, R. Zhou, Y. Zhang, T. Wei, and L. Lu, "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, 2020, pp. 1580–1596.

[11] K. Chua, R. Calandra, R. McAllister, and S. Levine, "Deep reinforcement learning in a handful of trials using probabilistic dynamics models," in *Advances in Neural Information Processing Systems*, vol. 31, 2018.

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

[13] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[14] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," *IEEE Computer Society*, 2017.

[15] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," in *Proceedings of the 35th International Conference on Machine Learning*, 2018.

[16] H. v. Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 2094–2100.

[17] L. Hongliang, Z. Yini, Y. Yue, X. Zhuosi, and J. Lin, "Sequence coverage directed greybox fuzzing," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019.

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

[19] M. Janner, J. Fu, M. Zhang, and S. Levine, "When to trust your model: Model-based policy optimization," in *Advances in Neural Information Processing Systems*, 2019.

[20] S. Jian, L. Cao, G. Pang, L. Kai, and G. Hang, "Embedding-based representation of categorical data by hierarchical value coupling learning," in *International Joint Conference on Artificial Intelligence*, 2017.

[21] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Icnn95-international Conference on Neural Networks*, 1995.

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

[23] T. Lattimore, "Regret analysis of the finite-horizon gittins index strategy for multi-armed bandits," in *Proceedings of the 29th Conference on Learning Theory, COLT 2016, New York, USA, June 23-26, 2016*, ser. JMLR Workshop and Conference Proceedings, V. Feldman, A. Rakhlin, and O. Shamir, Eds., vol. 49. JMLR.org, 2016, pp. 1214–1245.

[24] lcamtuf. (2023) American fuzzy lop (afl) fuzzer. https://lcamtuf.coredump.cx/ afl/.

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

[26] X. Li, K. Wang, L. Liu, J. Xin, H. Yang, and C. Gao, "Application of the entropy weight and topsis method in safety evaluation of coal mines," *Procedia Engineering*, vol. 26, pp. 2085–2091, 2011, iSMSSE2011. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877705811052532

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

[28] H. Liang, L. Jiang, L. Ai, and J. Wei, "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, 2020, pp. 127–137. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054807

[29] H. Liang, Y. Zhang, Y. Yu, Z. Xie, and L. Jiang, "Sequence coverage directed greybox fuzzing," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 2019, pp. 249–259.

[30] LLVM/Clang. Clang 9 documentation c. undefined behavior sanitizer. http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. 2023.

[31] C. Lyu, S. Ji, C. Zhang, Y. Li, W. Lee, Y. Song, and R. A. Beyah, "Mopt: optimized mutation scheduling for fuzzers," in *USENIX Security Symposium*, 2019.

[32] A. Mahendran and A. Vedaldi, "Understanding deep image representations by inverting them," *IEEE*, 2015.

[33] P. D. Marinescu and C. Cadar, "Katch: High-coverage testing of software patches," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 235–245.

[34] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," 2013, nIPS Deep Learning Workshop 2013.

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

[36] Y. Noller, R. Kersten, and C. S. Pasareanu, "Badger: Complexity analysis with fuzzing and symbolic execution," in *Software Engineering and Software Management, SE/SWM 2019, Stuttgart, Germany, February 18-22, 2019*, ser. LNI, S. Becker, I. Bogicevic, G. Herzwurm, and S. Wagner, Eds., vol. P-292. GI, 2019, pp. 65–66. [Online]. Available: https://doi.org/10.18420/se2019-16

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

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

[39] S. Poeplau and A. elien Francillon, "Symqemu: Compilation-based symbolic execution for binaries," in *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021*. The Internet Society, 2021. [Online]. Available: https://www.ndss-symposium.org/ndss-paper/symqemu-compilation-based-symbolic-execution-for-binaries/

[40] S. Poeplau and A. Francillon, "Symbolic execution with symcc: Don't interpret, compile!" in *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, S. Capkun and F. Roesner, Eds. USENIX Association, 2020, pp. 181–198. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/poeplau

[41] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," *Computer Science*, 2015.

[42] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "AddressSanitizer: A fast address sanity checker," in *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, 2012, pp. 309–318. [Online]. Available: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany

[43] A. Shah, D. She, S. Sadhu, K. Singal, P. Coffman, and S. Jana, "Mc2: Rigorous and efficient directed greybox fuzzing," ser. CCS '22, Los Angeles, CA, USA, 2022. [Online]. Available: https://doi.org/10.1145/3548606.3560648

[44] D. She, R. Krishna, L. Yan, S. Jana, and B. Ray, "Mtfuzz: fuzzing with a multi-task neural network," in *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*. ACM, 2020, pp. 737–749. [Online]. Available: https://doi.org/10.1145/3368089.3409723

[45] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 803–817.

[46] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, 1998, pp. 69–73.

[47] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *Computer ence*, 2013.

[48] S. Sivakorn, G. Argyros, K. Pei, A. D. Keromytis, and S. Jana, "Hvlearn: Automated black-box analysis of hostname verification in ssl/tls implementations," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017.

[49] M. Tufano, D. Drain, A. Svyatkovskiy, S. K. Deng, and N. Sundaresan, "Unit test case generation with transformers and focal context," arXiv, May 2021.

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

[51] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *The 38th IEEE Symposium on Security and Privacy(S&P'17)*. USENIX Association, 2017.

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

[53] M. Wu, L. Jiang, J. Xiang, Y. Huang, H. Cui, L. Zhang, and Y. Zhang, "One fuzzing strategy to rule them all," in *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 2022, pp. 1634–1645. [Online]. Available: https://doi.org/10.1145/3510003.3510174

[54] W. You, P. Zong, K. Chen, X. Wang, X. Liao, P. Bian, and B. Liang, "Semfuzz: Semantics-based automatic generation of proof-of-concept exploits," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2017, pp. 2139–2154. [Online]. Available: https://doi.org/10.1145/3133956.3134085

[55] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "EcoFuzz: Adaptive Energy-Saving greybox fuzzing as a variant of the adversarial Multi-Armed bandit," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020.

[56] G. Zhang, P. Wang, T. Yue, X. Kong, S. Huang, X. Zhou, and K. Lu, "Mobfuzz: Adaptive multi-objective optimization in gray-box fuzzing," 2022. [Online]. Available: https://api.semanticscholar.org/CorpusID:248224859

[57] L. Zhaoji, W. Tianyuan, Z. Ziqiang, and et al., "Directed grey-box fuzzing technology based on lstm and dynamic strategy," *Computer Engineering and Applications*, vol. 58, no. 18, pp. 147–153, 2022.

[58] H. Zheng, J. Zhang, Y. Huang, Z. Ren, H. Wang, and C. Cao, "Fishfuzz: Catch deeper bugs by throwing larger nets," in *32th USENIX Security Symposium (USENIX Security 23)*. USENIX Association, Aug. 2023.

[59] P. Zong, T. Lv, D. Wang, Z. Deng, R. Liang, and K. Chen, "FuzzGuard: Filtering out unreachable inputs in directed grey-box fuzzing through deep learning," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2255–2269. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/zong

APPENDIX

TABLE III. THE TTR RESULTS ON PROGRAMS FROM UNIBENCH

No	Prog	Version	Target sites	AFLGo	BEACON	WindRanger	ParmeSan	DeepGo-v	DeepGo-r	DeepGo
1	cflow	1.6	parser.c:281	T.O.	99.4m	61.1m	T.O.	58.1m	T.O.	41.8m
2			c.c:1783	12.8m	22.1m	6.45m	10.1m	12.2m	16.4m	11.2m
3			parser.c:1223	1.22m	0.83m	2.44m	6.23m	6.16m	4.76m	8.23m
4			parser.c:108	12.8m	68.1m	8.32m	6.76m	13.5m	17.2m	16.1m
5	mp42aac	Bento4 1.5.1-628	Ap4AvccAtom.cpp:82	T.O.	N/A	T.O.	N/A	T.O.	T.O.	891m
6			Ap4TrunAtom.cpp:139	T.O.	N/A	T.O.	N/A	T.O.	T.O.	723m
7			Ap4SbgpAtom.cpp:81	T.O.	N/A	T.O.	N/A	T.O.	T.O.	1022m
8			Ap4AtomFactory.cpp:490	T.O.	N/A	T.O.	N/A	T.O.	T.O.	878m
9	jhead	3.00	exif.c:1339	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	472m
10			iptc.c:143	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	355m
11			iptc.c:91	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	892m
12			makernote.c:174	T.O.	N/A	T.O.	T.O.	671m	T.O.	98.1m
13	mp3gain	1.5.2	layer3.c:1116	1142m	N/A	984m	N/A	871m	T.O.	172m
14			mp3gain.c:602	T.O.	N/A	T.O.	N/A	T.O.	T.O.	572m
15			interface.c:663	T.O.	N/A	T.O.	N/A	T.O.	T.O.	912m
16			apetag.c:341	290m	N/A	91.2m	N/A	164m	345m	72.8m
17	lame	3.99.5	bitstream.c:823	T.O.	N/A	T.O.	N/A	T.O.	T.O.	521m
18			lame.c:2148	T.O.	N/A	T.O.	N/A	T.O.	T.O.	291m
19			uantize_pvt.c:441	T.O.	N/A	1269m	N/A	T.O.	T.O.	599m
20			get_audio.c:1605	T.O.	N/A	T.O.	N/A	932m	T.O.	412m
21	imginfo	jasper 2.0.12	jp2_cod.c:841	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	619m
22			jp2_cod.c:636	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	776m
23			jas_stream.c:823	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	351m
24			jpc_dec.c:1393	T.O.	N/A	T.O.	984m	641m	T.O.	211m
25	gdk-pixbuf-pixdata	gdk-pixbuf 2.31.1	gdk-pixbuf-loader.c:387	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	1126m
26			io-qtif.c:511	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	622m
27			io-jpeg.c:691	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	498m
28			io-tga.c:360	126m	T.O.	T.O.	N/A	87.3m	172m	48.7m
29	jq	1.5	jv_dtoa.c:3122	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	1223m
30			jv_dtoa.c:2004	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	1267m
31			jv_dtoa.c:2518	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	587m
32			jv_unicode.c:42	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	1024m
33	tcpdump	4.8.1	print-aodv.c:259	T.O.	N/A	843m	T.O.	T.O.	T.O.	622m
34			print-ntp.c:412	1436m	N/A	974m	1239m	1213m	T.O.	542m
35			print-rsvp.c:1252	T.O.	N/A	T.O.	889m	338m	T.O.	223m
36			print-l2tp.c:606	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	821m
37	tic	ncurses 6.1	captoinfo.c:189	T.O.	N/A	N/A	T.O.	T.O.	T.O.	662m
38			alloc_entry.c:141	T.O.	N/A	N/A	T.O.	T.O.	T.O.	761m
39			name_match.c:111	1186m	N/A	N/A	T.O.	251m	1368m	155m
40			entries.c:78	1038m	N/A	N/A	883m	749m	1411m	495m
41	flvmeta	1.2.1	json.c:1036	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	682m
42			api.c:718	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	577m
43			flvmeta.c:1023	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	1021m
44			check.c:769	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	874m
45	tiffsplit	libtiff 3.9.7	tif_ojpeg.c:1277	T.O.	N/A	T.O.	N/A	T.O.	T.O.	561m
46			tif_read.c:335	T.O.	N/A	T.O.	N/A	T.O.	T.O.	1123m
47			tif_jbig.c:277	T.O.	N/A	T.O.	N/A	T.O.	T.O.	1046m
48			tif_dirread.c:1977	T.O.	N/A	T.O.	N/A	1068m	T.O.	825m
49	nm	binutils-5279478	tekhex.c:325	T.O.	364m	798m	N/A	369m	T.O.	264m
50			elf.c:8793	T.O.	986m	T.O.	N/A	665m	T.O.	342m
51			dwarf2.c:2378	1313m	831m	1065m	N/A	942m	T.O.	212m
52			elf-properties.c:51	T.O.	T.O.	T.O.	N/A	T.O.	T.O.	1062m
53	pdftotext	4.00	XRef.cc:645	T.O.	N/A	T.O.	N/A	T.O.	T.O.	427m
54			GfxFont.cc:1337	1345m	N/A	T.O.	N/A	1143m	1423m	785m
55			Stream.cc:1004	725m	N/A	T.O.	N/A	498m	911m	352m
56			GfxFont.cc:1643	637m	N/A	T.O.	N/A	T.O.	876m	T.O.
57	sqlite3	SQLite 3.8.9	pager.c:5017	617m	N/A	N/A	1214m	325m	810m	44.1m
58			func.c:1029	T.O.	T.O.	N/A	T.O.	T.O.	T.O.	1126m
59			insert.c:1498	T.O.	T.O.	N/A	T.O.	310m	T.O.	452m
60			vdbe.c:1984	T.O.	89.6m	N/A	T.O.	T.O.	T.O.	628m
61	exiv2	0.26	tiffcomposite.cpp:82	73.1m	N/A	68.1m	N/A	57.2m	93.2m	42.1m
62			XMPMeta-Parse.cpp:847	37.5m	N/A	21.4m	N/A	T.O.	79.5m	T.O.
63			tiffvisitor.cpp:1044	102m	N/A	T.O.	N/A	89.2m	112m	69.5m
64			XMPMeta-Parse.cpp:896	86.7m	N/A	421m	N/A	T.O.	99.1m	T.O.
65	objdump	binutils-2.28	elf.c:9509	T.O.	782m	T.O.	T.O.	T.O.	T.O.	1294m
66			section.c:936	T.O.	T.O.	T.O.	T.O.	T.O.	T.O.	1244m
67			bfd.c:1108	T.O.	361m	1288m	T.O.	T.O.	T.O.	1175m
68			bfdio.c:262	T.O.	1123m	T.O.	T.O.	T.O.	T.O.	1032m
69	ffmpeg	4.0.1	rawdec.c:268	T.O.	N/A	N/A	T.O.	221m	T.O.	183m
70			decode.c:557	T.O.	N/A	N/A	N/A	338m	T.O.	182m
71			dump.c:632	T.O.	N/A	N/A	N/A	673m	T.O.	498m
72			utils.c	T.O.	N/A	N/A	N/A	238m	T.O.	178m
73	mujs	1.0.2	jsrun.c:572	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.
74			jsgc.c:47	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.
75			jsdump.c:292	532m	N/A	421m	433m	488m	T.O.	361m
76			jsvalue.c:362	T.O.	N/A	T.O.	T.O.	T.O.	T.O.	T.O.
77	swftools	0.9.2	initcode.c:242	324m	N/A	223m	N/A	298m	401m	196m
78			png.c:410	871m	N/A	681m	N/A	501m	1004m	431m
79			poly.c:137	T.O.	N/A	T.O.	N/A	T.O.	T.O.	T.O.
80			jpeg2swf.c:257	677m	N/A	541m	N/A	611m	881m	481m
		speedup		**3.23×**	**1.72×**	**1.81×**	**4.83×**	**2.05×**	**4.26×**	**-**
		mean $\hat{A}_{12}$		**0.86**	**0.81**	**0.83**	**0.89**	**0.83**	**0.90**	**-**
		mean p-values		0.008	0.032	0.016	0.001	0.013	0.006	**-**