Reinforcement Learning-Based Fuzzing Technology

Zheng Zhang[1], Baojiang Cui[1(✉)], and Chen Chen[1,2]

[1] Beijing University of Posts and Telecommunications, Beijing, China
{zhangzheng232, cuibj}@bupt.edu.cn,
00152tenten@163.com
[2] Air Force Engineering University, Xian, China

Abstract. Fuzzing is a common vulnerability detection method in the modern software testing, which triggers potential vulnerabilities in the target program by generating variable input. However, traditional methods have the disadvantage of low code coverage due to the blind mutation of samples. To mitigate the problem, we model the process of traditional fuzzing as the Markov decision process and take use of the reinforcement learning algorithm to guide the direction of each step in the process of mutation to improve the quality of samples and the efficiency of fuzzing. In this paper, we implemented a general fuzzing system called RLFUZZ based on the reinforcement learning, taking the edge coverage as reward and using DDPG algorithm to maximize it. Experimental results show that DDPG-based RLFUZZ achieves greater edge coverage than baseline random mutation on LAVA-M dataset.

1 Introduction

Fuzzing is an efficient method applied in software testing that continuously feeds the target with randomly modified input to trigger vulnerabilities. Existing fuzzers explore different methods in input mutation, but none can exhaustively examine the entire input space in practice or search the whole execution paths in target programs because of the huge search space and constraints of computing resource. Therefore, current fuzzers utilize fuzzing heuristics to prioritize what fuzzing strategies to be taken to maximize a specific goal, such as crash [1], code coverage [2], etc.

Coverage-guided fuzzing is widely adopted by fuzzers that utilizes code coverage to guide the search for fuzzing strategy. However, common tools such as libFuzzer [3], still select randomly mutation strategy by default when processing the sample, so it is unable to generate samples to the direction of increasing code coverage. Therefore, we need to guide mutation according to coverage as feedback, intelligently select fuzzing strategy and mitigates the efficiency problem caused by the blindness of mutation.

Reinforcement learning refers to that the agent interacts with the environment by taking actions according to states and obtains the corresponding reward, then learns the optimal selection strategy of action to maximize the cumulative reward from historical experience [4]. For the problem of large state and action space, researchers introduce the neural network, using its excellent expression ability to calculate the value function of high-dimensional space, namely deep reinforcement learning (DRL).

L. Barolli et al. (Eds.): IMIS 2020, AISC 1195, pp. 244–253, 2021.
https://doi.org/10.1007/978-3-030-50399-4_24

In this paper, we implemented a general fuzzing system RLFUZZ based on the reinforcement learning. Fuzzing elements can be abstracted into state, action and reward, then reinforcement learning algorithm selects the appropriate action according to state to obtain the maximum reward, achieving the guided mutation of samples. We test RLFUZZ with different warmup steps and activation functions of hidden layers, and evaluate its performance with the baseline random mutation and some DQN algorithms on LAVA-M [5]. The results show that RLFUZZ has a significant improvement in edge coverage. We mainly make the following contributions: (1) We model the traditional fuzzing process as a multi-step MDP and propose to use Deep Deterministic Policy Gradient (DDPG) algorithm to choose a trace of high-reward mutation actions for given program input. (2) We implement the DDPG-based RLFUZZ that outperforms baseline random fuzzing on LAVA-M.

2 Related Work

Our work is influenced by mutation-based and machine learning-based fuzzing.

Mutation-Based Fuzzing. It changes the state of target program by constantly modifying the normal input to find vulnerabilities. For examples, PerfFuzz [6] uses multi-dimensional feedback and independently maximizes the execution counts for all program locations to increase the length of execution path and the execution number of most-hit branches in programs. FuzzFactory [7] further generalizes the above situation by domain specific instrumentation to provide a general solution framework for problems with different feedbacks. Fairfuzz [8] uses Branch Mask to guide the mutation to increase the execution probability of rare branches. However, traditional mutation-based fuzzing mutates samples randomly and generates a large number of invalid samples. So now researchers mostly guide the mutation based on feedback.

Machine Learning-Based Fuzzing. Recently machine learning methods have been introduced into fuzzing to improve the traditional mutation-based fuzzing [9]. V-Fuzz [10] uses Graph Neural Network (GNN) to predict the possible location of vulnerabilities in the target program to guide the mutation. NeuFuzz [11] uses Deep Neural Network (DNN) to optimize the selection strategy of samples and improve the coverage of vulnerable path. Security researchers have also done relevant research on DRL. Böttinger et al. [12] uses Deep Q-learning to improve the code coverage and consuming time. FuzzerGym [13] combines libFuzzer with Deep Double Q-learning to improve the code line coverage. REINAM [14] uses reinforcement learning to generate input grammars to improve the quality of samples. Kuznetsov et al. [15] uses Deep Q-learning to reduce the number of mutations required to detect vulnerability by 30%. Although introduction of DRL into fuzzing has become a research trend, current research still focuses on value-based reinforcement learning algorithm, which is hard to deal with the problem of large state and action space.

3 Reinforcement Learning

This section gives necessary background on reinforcement learning. Solution of reinforcement learning problem is actually the optimal strategy π^* for maximum reward, which can be obtained by solving the Bellman Eq. 1 with state s, immediate reward R_s, future discount factor γ and transfer probability $P_{ss'}$.

$$v(s) = R_s + \gamma \sum_{s' \in S} P_{ss'} v(s'). \tag{1}$$

Methods for solving reinforcement learning tasks based on function approximation include value-based, policy-based, etc. Value-based method is difficult to solve the problem of converge or continuous action space. So, we choose policy-based method to solve tasks, such as Policy Gradient (PG). DDPG algorithm introduces DNN into Deterministic Policy Gradient (DPG) algorithm and uses Actor-Critic as the basic architecture. It well balances the exploration of state space S and action space A and calculation efficiency. Next, we introduce PG and DDPG in turn.

Policy Gradient. The method fits policy $\pi_\theta(a|s)$ through function approximation to obtain the probability of a under s with parameter θ. The target function $J(\theta)$ of π is set as the expectation about r. In order to maximize $J(\theta)$, θ is updated by stochastic gradient ascent (SGA) algorithm. In Policy Gradient Theorem [16]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s \sim \rho^\pi, a \sim \pi_\theta}[\nabla_\theta log \pi_\theta(a|s) Q^\pi(s,a)]. \tag{2}$$

$\nabla_\theta log \pi_\theta(a|s)$ represents the evaluation function that can be calculated by Softmax strategy for discrete tasks or Gauss strategy for continuous tasks. $Q^\pi(s,a)$ represents the action value function of π and can be solved by Actor-Critic method. Actor selects a based on probability and critic evaluates it with a score, then actor modifies the probability of subsequent a based on the score. In other words, actor is responsible for updating π_θ and θ according to the action value function of critic and critic is responsible for updating $Q_w(s,a)$ and w by $Q_w(s,a) \approx Q^\pi(s,a)$. The Actor-Critic uses Time Difference (TD) method to update the value function step by step, which does not need complete experience trajectories but is difficult to converge.

Deep Deterministic Policy Gradient. PG needs to sample the high-dimensional action space to obtain the specific action at every step, which consumes computing resource in abundance. Silver et al. [17] prove the existence of deterministic policy gradient by strict mathematical derivation, that a given state s corresponds to an optimal $a = \mu_\theta(s)$ without sampling. Lillicrap et al. [18] propose DDPG algorithm with following improvements compared with DPG: (1) DNN is used to represent policy function μ and action value function Q. (2) Experience replay removes the temporal correlation and dependence among the data of state transition and the bias caused by function approximation. (3) Dual network architecture in both actor and critic makes the learning process more stable and easier to converge. After training on a mini batch of data, the parameters of online network are updated through gradient ascent or descent algorithm and the parameters of target network are updated through soft update algorithm.

In general, the goal of DDPG algorithm training is to maximize the target function J and minimize the loss of value network. It can solve the problem with high-dimensional state and action space. In the traditional fuzzing process, there are various mutation samples and strategies, so the MDP modeled from it also has high-dimensional state and action space. Therefore, we choose DDPG algorithm to solve the process to improve the traditional fuzzing by guiding the mutation of samples.

4 Design and Implementation

In the MDP modeled from fuzzing, state can be represented by samples as their bytes, bits [19] and substrings [12]. Action can be mutation strategies, such as functions, XOR matrix, character list, etc. Reward is calculated from running information of target program, such as coverage based on basic block, edge, code line, etc. Different methods of calculating reward have a great influence on reinforcement learning.

Based on the above analysis, we design RLFUZZ as shown in Fig. 1 to implement the general fuzzing based on reinforcement learning. Reinforcement learning network in the left selects action according to state and reward from right fuzzing target environment and return it back. Python interface layer chooses mutation strategy according to the action to mutate samples, and then transfers the mutated to target program and obtains the status after execution through shared memory and C/C++ proxy, then calculates reward and returns it to reinforcement learning network.

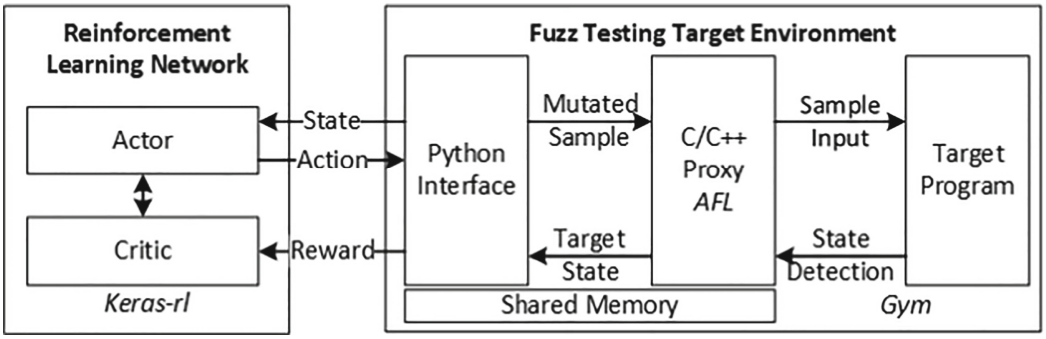

Fig. 1. The high-level architecture of RLFUZZ system.

In this section, we model the traditional fuzzing as MDP, including state, action and reward and describe their implementation as well as DDPG model in RLFUZZ.

4.1 State

For the universality and scalability of the system, we specify the state space as all sample data in the form of byte array with value range as $[0, 255]$. The maximum length M is specified as different value in specific environments. If the size of sample data is less than M, then 0 is used to complete it. According to FuzzerGym [13], in order to maximize the position of finding new paths through code, it is better to select empty seed during initialization. So, we set the initial sample empty, $s_0 = [0] * M$.

RLFUZZ maintains a list of effective samples EFF_input and adds one that generates new path to it. If last mutated sample generates a new path, it is selected as current state s_i and added to the list. Otherwise, s_i is set as one randomly selected from EFF_input. This effectively mitigates the problem of limited state exploration.

4.2 Action

There are a variety of mutation methods in fuzzing. For the sake of performance, we designate action space as a series of optional functions as shown in Table 1.

Table 1. The functions of mutation in action space.

Action	Description
Mutate_EraseBytes	Delete random bytes
Mutate_InsertByte	Insert a random byte
Mutate_InsertRepeatedBytes	Insert at least 3 identical random bytes
Mutate_ChangeByte	Change a random byte
Mutate_ChangeBit	Flip a random bit of a byte
Mutate_ShuffleBytes	Rearrange random serial bytes
Mutate_ChangeASCIIInteger	Change a random integer
Mutate_ChangeBinaryInteger	Change several random bytes
Mutate_CopyPart	Copy or insert random bytes

RLFUZZ takes these functions as A and maintains a dictionary of function pointers funcMap with a unified calling interface. Reinforcement learning algorithm selects action index number and transfers it to back end through action interface. Then the back end calls specific function to achieve mutation on the sample.

4.3 Reward

In traditional fuzzing, the criteria of measuring the success is whether samples trigger vulnerabilities. However, the generation of effective samples is generally a long process of continuous exploration. So, researchers utilize code coverage to assist in evaluating the effectiveness of samples. Instead of using simple basic blocks, AFL [20] uses edge coverage that provides relatively more insight into the execution path of the program. Therefore, we take the edge coverage as reward. Each unit of memory storing execution status in AFL records the execution number of an edge jumping from basic block bb_i to bb_j. Suppose the memory of execution status is denoted as SHM, then $r = \frac{N(SHM)}{L(SHM)}$. $L(SHM)$ represents the total capacity of SHM, and $N(SHM)$ represents the number of non-zero units in SHM, meaning the number of edges executed at least once.

RLFUZZ uses *fork server* mode of AFL to instrument and call the target program. Gym sends sample data to AFL, which sends SHM and program termination status

code back. Gym then calculates r and returns it to reinforcement learning network through python interface of reward.

4.4 DDPG Network

We use *keras-rl* to build the DDPG model. The network of actor in the left of Fig. 2 consists of 2 hidden layers with 1024 and 128 nodes. The size of input layer is the size of state space and the size of output layer is the size of action space. The activation of last layer is *softmax*. The network of critic in the right of Fig. 2 also consists of 2 hidden layers with 1024 and 128 nodes. Its input layer size is the size of state space and action space. and the size of output layer is 1. The activation of its last layer is *sigmoid*.

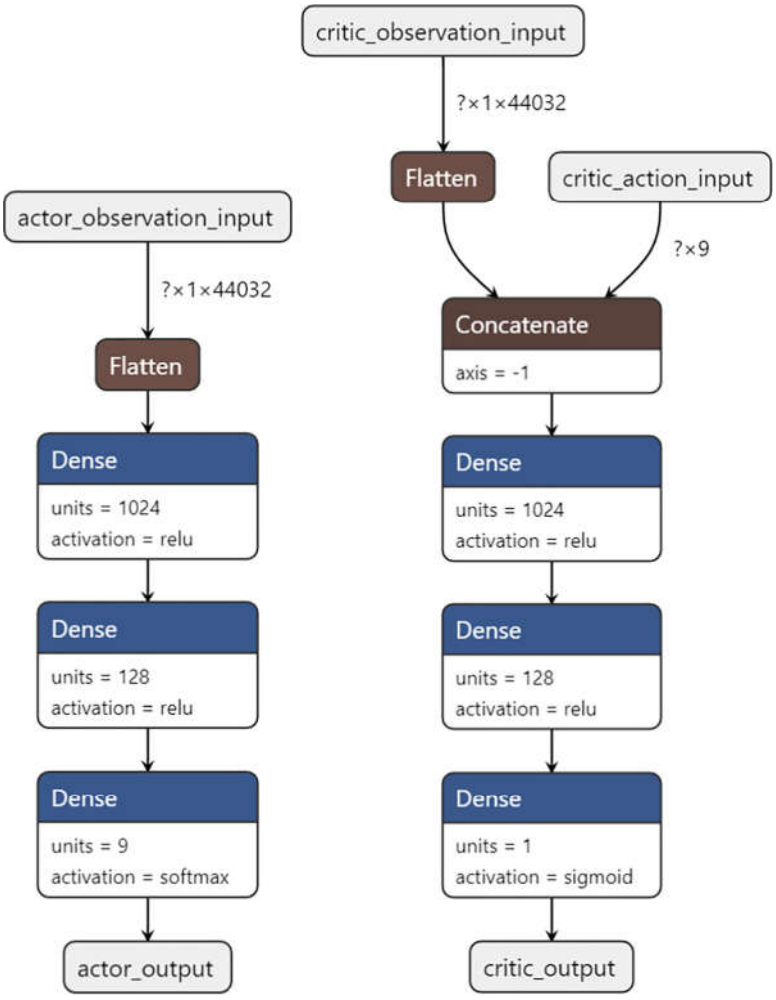

Fig. 2. The specific structures of actor and critic online network in an experiment on base64.

We set the size of ring buffer for experience replay as 100000 for storing interaction data. The Ornstein Uhlenbeck stochastic process [21] is introduced as random noise for action exploration with its size set the size of action space and $\theta = 0.15$, $\mu = 0$, $\sigma = 0.3$. DDPG updates critic network with $\gamma = 0.99$ and soft updates target network

with a small amplitude at each time step, that is $\theta^{Q'} = \tau\theta^{Q} + (1-\tau)\theta^{Q'}$ with $\tau = 0.001$. We choose the Adam optimizer with learning rate as 0.001.

5 Experiments and Evaluation

This section describes the experiments for evaluating RLFUZZ and analyzes relevant results in two parts. The first includes comparison of different warmup steps and activation functions in DDPG. The second includes experiments on LAVA-M dataset. All were performed on E5-2630 V4 2.20 GHz with 384 GB of RAM, including keras-rl 0.4.2, gym 0.15.4 and AFL 2.52b on 64-bit Ubuntu 16.04.6 LTS.

5.1 DDPG Model

Different warmup steps and activation functions of model have a serious impact on efficiency. DDPG model needs to execute warmup steps to fill the experience pool for network updating parameters afterwards. Activation function introduces nonlinear factors into network, affecting its expression ability and convergence speed. Therefore, we analyze their comparative experimental results and select the optimal for RLFUZZ. We denote the sum of values in *SHM* as Number of Basic Block Edge (NBBE) and save it to the log of RLFUZZ during the running of base64 with *-d* from LAVA-M.

Warmup Steps. We carried out 10 comparison experiments, setting warmup steps as 10%,…,100% of total 20000 steps. According to the distribution histogram of NBBE in Fig. 3, 10% of total steps outperform others and the NBBE of its mutated samples generated mainly distribute in $(324.2, 362.0)$, while the NBBE generated by others mostly concentrates in $(210.8, 229.7)$. 10% of total generates more effective samples with larger edge coverage, which is conducive to improving the efficiency of fuzzing. So RLFUZZ takes 10% of total steps as its warmup steps.

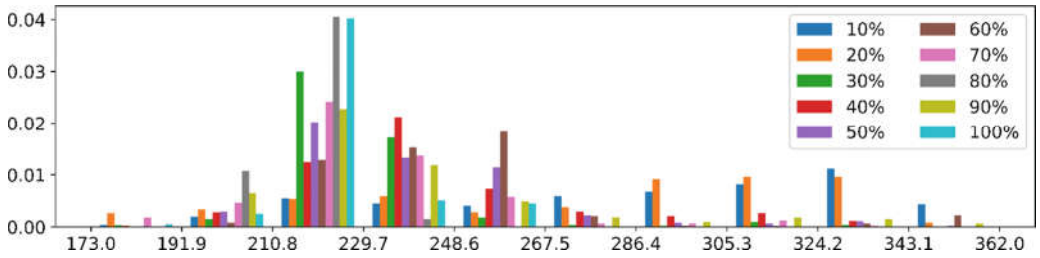

Fig. 3. The distribution histogram of NBBE of all samples generated with different warmup steps.

Activation Functions. We have done 9 experiments with setting activation function of hidden layers in actor and critic as elu, selu, softplus, softsign, relu, tanh, sigmoid, hard_sigmoid and linear. The number of total steps is 5000 and the number of warmup steps is 2000. The distribution histogram of NBBE is as shown in Fig. 4. Only the NBBE of samples generated by relu concentrates in $(283.6, 354.0)$, while the NBBE of

samples generated by other activation functions concentrates in $(213.2, 230.8)$. Relu outperforms others in NBBE, so RLFUZZ uses *relu* as its activation function.

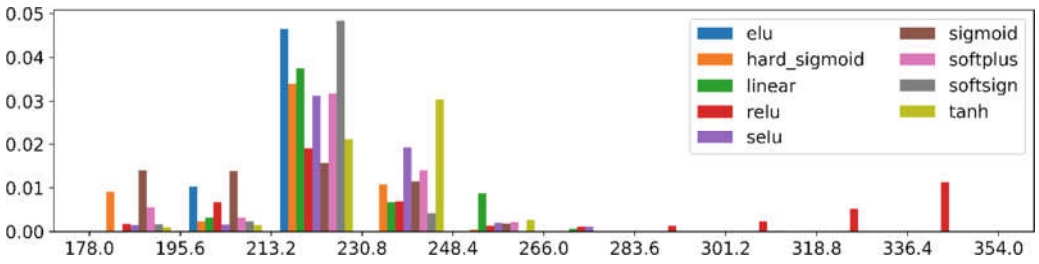

Fig. 4. The distribution histogram of NBBE of all samples with different activation functions.

5.2 LAVA-M Dataset

LAVA-M is widely used in fuzzing research and it contains source codes with inserted multiple vulnerabilities of 4 Linux programs and corresponding samples. We choose the random mutation strategy as baseline, meaning interaction with environment by random actions. In view of existing research that DQN performs well in fuzzing, so we choose three DQN algorithms in experiments. Non-random algorithms all perform 2000 warmup steps and 20000 steps in total. We choose DDPG, DQN, Double DQN and Duel DQN and random strategy to experiment on base64, md5sum and uniq.

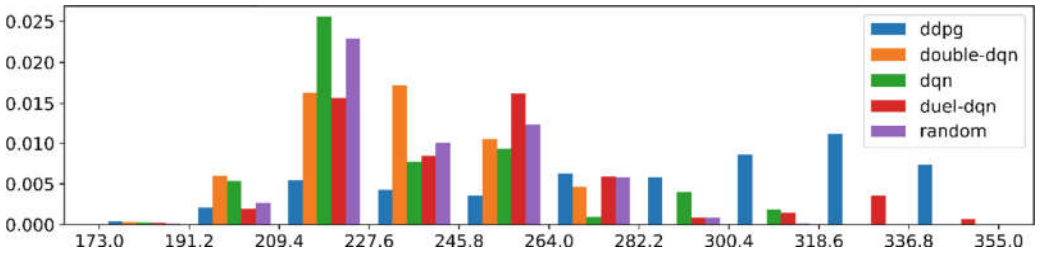

Fig. 5. The distribution histogram of NBBE of all generated samples on base64.

The figures are the distribution histograms of NBBE from each experiment. On base64 in Fig. 5, the NBBE of DDPG-generated samples mainly distributes in $(300.4, 355.0)$, while DQN and random in $(209.4, 227.6)$, Double DQN in $(209.4, 245.8)$ and Duel DQN in $(227.6, 282.2)$. On md5sum in Fig. 6, the NBBE of DDPG-generated samples mostly distributes in $(313.8, 335.0)$, while others mainly distribute in $(229.0, 250.2)$. On uniq in Fig. 7, the NBBE of DQN and random-generated samples predominantly distribute in $(196.4, 205.8)$, while others concentrate in $(224.6, 234.0)$. Although DDPG is less than Duel and Double DQN, the difference between DDPG and Duel DQN is 11.8% and the difference between DDPG and Double DQN is only 6.7%, also DDPG gets the maximum value 234.

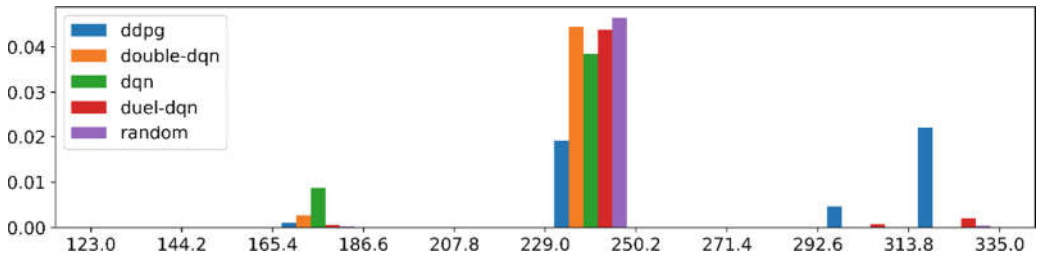

Fig. 6. The distribution histogram of NBBE of all generated samples on md5sum.

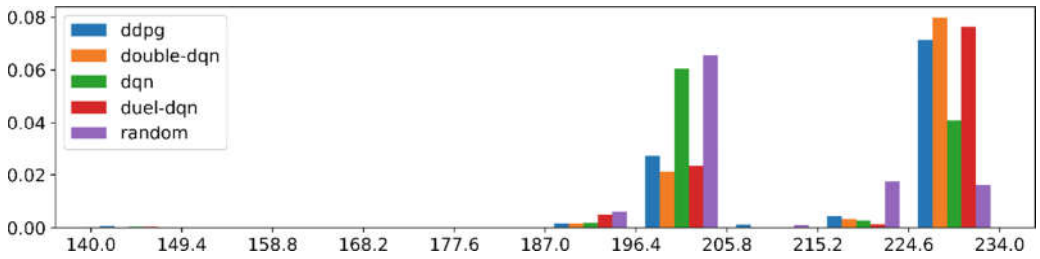

Fig. 7. The distribution histogram of NBBE of all generated samples on uniq.

In conclusion, RLFUZZ with DDPG can generate samples with greater edge coverage on the whole, which has a significant improvement compared with baseline random strategy and even outperforms DQN algorithms in some aspects.

6 Conclusion

In order to mitigate the problem of low code coverage caused by blindness of mutation in traditional fuzzing, in this paper, we model the fuzzing process as the MDP and implement RLFUZZ, an end-to-end general fuzzing system based on reinforcement learning with sample as state, mutation function as action and edge coverage as reward. It improves the traditional fuzzing by guiding the mutation in each step to reduce the generation of invalid samples. Experiments on LAVA-M dataset show that DDPG algorithm outperforms the baseline random mutation strategy and DQN algorithms in some aspects. Future research should inquire into this further with more algorithms.

References

1. Vinesh, N., Rawat, S., Bos, H., Giuffrida, C., Sethumadhavan, M.: Confuzz—a concurrency fuzzer. In: First International Conference on Sustainable Technologies for Computational Intelligence, pp. 667–691. Springer (2020)
2. Padhye, R., Lemieux, C., Sen, K.: JQF: coverage-guided property-based testing in Java. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019) (2019). https://doi.org/10.1145/3293882.3339002
3. Serebryany, K.: Libfuzzer a library for coverage-guided fuzz testing. LLVM project (2015)

4. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
5. Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., Ulrich, F., Whelan, R.: LAVA: large-scale automated vulnerability addition. In: 2016 IEEE Symposium on Security and Privacy (SP), pp. 110–121. IEEE (2016)
6. Lemieux, C., Padhye, R., Sen, K., Song, D.: PerfFuzz: automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 254–265. ACM (2018)
7. Padhye, R., Lemieux, C., Sen, K., Simon, L., Vijayakumar, H.: FuzzFactory: domain-specific fuzzing with waypoints. In: Proceedings of the ACM on Programming Languages vol. 3, no. OOPSLA, pp. 174 (2019)
8. Lemieux, C., Sen, K.: FairFuzz: a targeted mutation strategy for increasing greybox fuzz testing coverage. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 475–485. ACM (2018)
9. Wang, Y., Jia, P., Liu, L., Liu, J.: A systematic review of fuzzing based on machine learning techniques. arXiv preprint. arXiv:1908.01262 (2019)
10. Li, Y., Ji, S., Lv, C., Chen, Y., Chen, J., Gu, Q., Wu, C.: V-Fuzz: Vulnerability-oriented evolutionary fuzzing. arXiv preprint. arXiv:1901.01142 (2019)
11. Wang, Y., Wu, Z., Wei, Q., Wang, Q.: NeuFuzz: efficient fuzzing with deep neural network. IEEE Access **7**, 36340–36352 (2019)
12. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. In: 2018 IEEE Security¨ and Privacy Workshops (SPW), pp. 116–122. IEEE (2018)
13. Drozd, W., Wagner, M.D.: FuzzerGym: a competitive framework for fuzzing and learning. arXiv preprint. arXiv:1807.07490 (2018)
14. Wu, Z., Johnson, E., Yang, W., Bastani, O., Song, D., Peng, J., Xie, T.: REINAM: reinforcement learning for input-grammar inference. In: Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 488–498. ACM (2019)
15. Kuznetsov, A., Yeromin, Y., Shapoval, O., Chernov, K., Popova, M., Serdukov, K.: Automated software vulnerability testing using deep learning methods. In: 2019 IEEE 2nd Ukraine Conference on Electrical and Computer Engineering (UKRCON), pp. 837–841. IEEE (2019)
16. Sutton, R.S., McAllester, D.A., Singh, S.P., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Advances in Neural Information Processing Systems, pp. 1057–1063 (2000)
17. Silver, D., Lever, G., Heess, N., Degris, T., Wierstra, D., Riedmiller, M.: Deterministic policy gradient algorithms (2014)
18. Lillicrap, T.P., Hunt, J.J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., Wierstra, D.: Continuous control with deep reinforcement learning. arXiv preprint. arXiv:1509.02971 (2015)
19. Rajpal, M., Blum, W., Singh, R.: Not all bytes are equal: neural byte sieve for fuzzing. arXiv preprint. arXiv:1711.04596 (2017)
20. Zalewski, M.: American fuzzy lop - a security-oriented fuzzer. https://github.com/google/AFL
21. Uhlenbeck, G.E., Ornstein, L.S.: On the theory of the Brownian motion. Phys. Rev. **36**(5), 823 (1930)