FuzzBoost: Reinforcement Compiler Fuzzing

Xiaoting Li[1], Xiao Liu[2], Lingwei Chen[3], Rupesh Prajapati[4], and Dinghao Wu[4](✉)

[1] Visa Research, Palo Alto, CA, USA
xiaotili@visa.com
[2] Meta, Inc., Menlo Park, CA, USA
bamboo@fb.com
[3] Wright State University, Dayton, OH, USA
lingwei.chen@wright.edu
[4] Pennsylvania State University, University Park, PA, USA
{rxp338,duw12}@psu.edu

Abstract. Enforcing the correctness of compilers is important for the current computing systems. Fuzzing is an efficient way to find security vulnerabilities in software by repeatedly testing programs with enormous modified, or fuzzed input data. However, in the context of compilers, fuzzing is challenging because the inputs are pieces of code that are required to be both syntactically and semantically valid to pass front-end checks. Also, the fuzzed inputs are expected to be distinct enough to trigger abnormal crashes, memory leaks, or failing assertions that have not been detected before. In this paper, we formalize compiler fuzzing as a reinforcement learning problem and propose an automatic code synthesis framework called FuzzBoost to empower the input code mutations in the fuzzing process. In our learning system, we incorporate the deep Q-learning algorithm to perform multi-step code mutations in each training episode, and design a reward policy to assess the testing coverage information collected at runtime. By interacting with the system, the fuzzing agent learns to predict code mutation actions that maximizing the fuzzing rewards. We validate the effectiveness of our proposed approach and the preliminary evidence shows that our reinforcement fuzzing method can outperform the fuzzing baseline on production compilers. Our results also show that a pre-trained model can boost the fuzzing process for seed programs with similar patterns.

Keywords: Compilers · Fuzzing · Reinforcement learning

1 Introduction

Compilers are fundamental in the current computing system as they are part of the trust base of the machine. However, they contain bugs and it is non-trivial

X. Li, X. Liu and L. Chen—Work done while at PSU.

to verify all the vulnerabilities due to their large codebase. For example, GCC has about 15 million lines of code [27]. Fuzzing is an effective way to find security vulnerabilities in compilers by repeatedly testing the codes with randomly modified, or fuzzed inputs [28]. It plays an important role in quality assurance, software development, and vulnerability assessment over decades [8,9,19,30]. Many existing vulnerabilities are reported by fuzzing techniques [23]. Due to the unlimited search space and limited computing resources, existing fuzzing tools explore efficient strategies in fuzzing program inputs. Especially in the scenario of compiler testing, no one can exhaustively examine the entire input space, or traverse all the possible execution paths of target compilers in practice. Therefore, a variety of strategies are designed based on fuzzing heuristics to prioritize finding interesting inputs to be fuzzed. Such fuzzing heuristics may be a random selection, or trying to maximize a specific goal, such as code coverage [15], execution timeouts, and crashes [35].

Coverage-guided testing is widely adopted by fuzzers [10,33,36], which utilizes code coverage information as the search heuristic to generate new inputs from the fuzz action of a predefined list. These exhaustive bounded searches use domain-specific heuristics and are thereby limited in applicability and scalability. Additionally, they overlook the benefit from *past experiences* in historical mutations and cannot automatically learn the common knowledge that is shared in different input seeds generated during the fuzzing boosting process. Moreover, most coverage-guided frameworks only calculate the rewards/fitness after a single mutation is taken, which yet underestimates the power of a series of mutation combinations. For instance, state-of-the-art mutation-based methods like American Fuzzing Lop (AFL) [36] add newly generated fuzzing programs after one mutation according to defined search heuristics into the seed set for the next round of fuzzing. However, for coverage-guided fuzzing, testing coverage does not increase linearly. In other words, each of these mutations may not improve the testing efficacy incrementally. They can be rejected by lexical or semantic checks in the early stage of compilation. But a trace of mutations may trigger a giant improvement as it may increase the possibility of generating more diverse input programs to enhance the code coverage of compilers.

Faced with these challenges, we formalize compiler fuzzing as a reinforcement learning problem and propose FuzzBoost to integrate the superiority of reinforcement learning to the coverage-guided fuzzing. The design of FuzzBoost is inspired by the fact that fuzzing can be modeled as a learning process with a feedback loop where the model aims to learn the mutation heuristics based on the feedback (reward) from the runtime information for evaluating the quality of current input [5]. Reinforcement learning describes the learning process by an agent interacting with the environment to learn an optimal policy by trial and error. It is usually effective for sequential decision-making problems in natural and social sciences, and engineering [3,29]. Theoretically speaking, the problem of compiler fuzzing can be seen as a problem of program synthesis, the goal of which is to cover more paths, trigger more crashes or memory leaks in compilers' execution traces while compiling new generated codes. Specifically, we model compiler fuzzing as a multi-step decision-making process where a learning task progresses with a feedback loop. The fuzzing agent initially generates new inputs with little knowledge but random heuristics. The compiler iteratively runs with the newly fuzzed input. Based on the feedback of the environment, we capture runtime information gathered from binary instrumentation techniques to evaluate the quality of input seeds according to heuristics we define in our learning cycle.

In this paper, we utilize seed programs from test suites of production compilers (GCC [11] in our research) to evaluate FuzzBoost. To demonstrate the effectiveness of our framework, we also compare it with a baseline fuzzing mechanism used in the system AFL [36], which is a widely-used fuzzing method. AFL applies mutation actions with a uniformly distributed strategy. From the results, FuzzBoost outperforms baseline random fuzzing with a higher coverage improvement on seed programs. Additionally, to better improve the efficiency of FuzzBoost on the fuzzing process, we conduct the experiments on a pre-trained model. As a result, our tool achieves a better fuzzing performance, which means that the fuzzing process can be boosted when we reuse the existing model for new seed programs in compiler fuzzing.

In summary, we make the following contributions:

- We integrate reinforcement learning to the compiler fuzzing problem and design a principled reinforcement fuzzing method to automatically generate new test seeds.
- We define reward functions to optimize the fuzzing goal and use a deep Q-learning algorithm to automatically learn a trace of high-reward mutations for given seeds which extensively leverage the knowledge in prior experiences. Our method is task-agnostic that does not rely on any other fuzzing techniques.
- We implement a prototyping tool called FuzzBoost and analyze real-world compiler fuzzing jobs. We conduct various analytical experiments and results demonstrate its testing efficacy.

2 Overview

Mutation-based fuzzing relies on generating new program inputs by mutating seed programs with heuristics. In the previous method [36], the designed fuzzer performs one-step manipulation on the provided input corpus. Then the fuzzer may select its collection of interesting fuzzed inputs after based on their performance, which is measured by capturing new crashes in the context of black-box fuzzing or capturing new path information in grey- or white-box fuzzing. However, it overlooks the potential of a trace of mutations in generating interesting fuzzed inputs, some intermediate states of which may not be good enough to attract interest or even break the compilation process due to lexical checks in early stages. Therefore, we re-model the problem as a multi-step decision-making problem that gives enough attention to these intermediate states being ignored in previous design models. Specifically, we formally model compiler fuzzing as a Markov decision-making process as described in Fig. 1.

As shown in the figure, in this multi-step decision-making process, there is an input mutation engine M, that performs a fuzzing action a, and subsequently

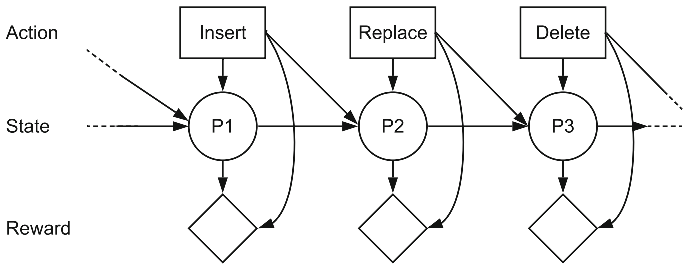


Fig. 1. Compiler fuzzing process

observes a new state s directly derived from the mutated program P_t by exercising the predicted action a on an original seed program P_{t-1}. It means that the input mutation engine predicts the program rewriting actions based on the extracted state from the seed program. After that, the engine can receive a reward r based on performed actions and system state transitions. With the given formalization, it is natural to use the Markov decision process (MDP) to model this problem. Therefore, we define the corresponding T-step finite horizon MDP as $M = (s_1, a_1, r_1, s_2, a_2, ..., s_T)$. Here s_t, a_t, r_t represent the state, action, and reward at time step $t = 1, ..., T-1$, respectively. To achieve the trace of the most effective rewrites for a seed program, we apply reinforcement learning methods [34] to deploy our formalization. Followed by prior footsteps [5,16], we use deep Q-learning algorithm [20,21] to learn the fuzzing engine.

In reinforcement learning, one episode is one complete sequence of actions that starts with an initial state configuration and ends with a terminal state. In the problem of compiler fuzzing, one episode can be formalized as generating a fuzzed program by performing one pre-designed mutation on an existing seed program (initial state), while the learning agent guides the mutation actions that aims to maximize the total reward it receives during the episode. Compared with those conventional mutation-based fuzz testing methods, we adopt the same methodology that using the coverage-guided heuristics to continuously select and generate the desired program generated from the seed set along the episode. The main difference is that, in our design, we *lazy-evaluate* the quality of the fuzzed inputs until it reaches the terminal state. To this end, our fuzzing process contains those intermediate states that might not be syntactically valid but can eventually contribute to high-quality fuzzed inputs.

Before we start the learning process, we randomly initialize a standard deep neural network. In the first episode, State 0 is represented as a program string P preprocessed from a seed program. To reduce the randomness and exhaustive space of mutation, we choose a substring of the whole input program to be our mutation target. Specifically, we extract a substring within a seed program with the window size (length of the substring) w and offset o. By observing this substring, the trained deep neural network automatically predict a mutation action to be taken in the next step. Feasible mutation actions on token-level include *insert* a token, *switch* two or more tokens, *replace* a token, or *change* the window position or size to enable another substring to observe and mutate. Once an action is decided, we run the compiler with the program after performing such a mutation and calculate the reward r of this new program with a record of the execution trace. Subsequently, it moves to the State 1 for further mutations. With the increased number of actions being taken, we deduct the reward by a discounted rate γ which is a value between 0 and 1 to enforce an expected fuzzing trace with fewer mutation actions. We iterate the mutation prediction and evaluation until a *terminal* state is achieved. During the learning process, there are four key elements in this process: *state*, *action*, *environment*, and *reward*.

2.1 State

A state S is a concrete configuration in the environment. As defined in MDP, each process has one state and when the process proceeds, the state updates. In the case of compiler fuzzing, the agent learns to interact with a given seed program. We define the state as a function regarding a given input seed program P. The interaction is performed upon the observation of selected substring within such an input, which is viewed as a series of consecutive token symbols. Formally, let Σ denote a finite set of symbols. The set of possible program inputs I in this language is defined by the Kleen closure $I := \Sigma^*$. For an input program string $P = (p_1, p_2, ..., p_n) \in I$, let

$$S(P) := \{(p_{1+i}, p_{2+i}, ..., p_{m+i}) \mid i \geq 0,\ m + i \leq n\} \qquad (1)$$

denote the set of all substrings of P. We define the states of the Markov decision process to be I and I is a union set of $S(P)$. Thus, we have $P \in I$ denotes an input program and $P_0 \in S(P) \subset I$ is a substring of this input seed program. The entire state space of a seed program is $S(P)$, which is theoretically infinite since permutations in this language I can increase after mutation. In other words, the seed program can be converted to any other valid programs.

2.2 Action

Action A is the set of all possible mutation actions that the agent can perform. In most cases, actions are deterministic and should be chosen among a pre-defined list. In compiler fuzzing, we define the set of possible action A of the MDP to be pre-designed rewrite rules on the extracted substrings $S(P_o)$. The rewrite rules are designed in accord with the extracted substring and predicted type. To be specific, we categorize rewrites from two perspectives, i.e., the extracted content and the extraction window, so the agent can predict which type and on which position an action should be performed on the current input.

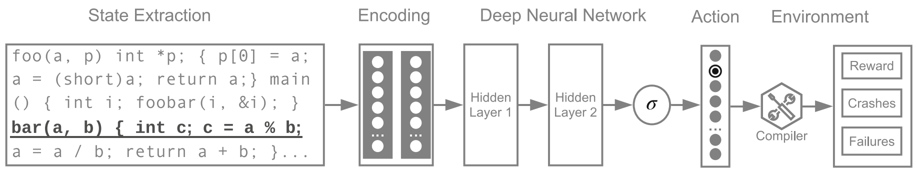
The rewrites of extracted content are performed on the token-level which include *insertion*, *replacement*, *re-ordering*, *deletion* and *replication*. These rewriting rules conform with the C language lexical requirements. For *insertion*, we append new tokens after the predicted index according to production rules; that is, if the last token is an operator, we randomly sample a token from the existing identifiers as its next. For *deletion*, we delete the token located at the predicted index. For *replacement*, we replace the token at the predicted index with another token randomly sampled from sets of tokens with same characteristic; e.g., if this token is a keyword of C, we select another keyword for replacement. Note, the keyword and operator token set are predefined, while identifier token set is generated by parsing the seed program. For the second type, they are designed to make a change on the extraction windows. Atomic mutations include window *left shift* and *right shift*, and window size *up* and *down* with one token length either from left or right side for each. Each of these actions do not change the input program but motivate the diversity of the extracted substring $S(P_o)$ and covers more states in the mutation space. For both types of mutations, the time step increases until the termination state is triggered on the current episode. We define a *terminate* action to early stop the mutation episode. That is, the mutation agent can proactively terminate a mutation episode while observing an extracted substring.

2.3 Environment

The environment is the world that the agent evaluates each action. The environment takes the current state and action as the input, and then outputs the reward of performing such action and calculates the next state after executing the action. In compiler fuzzing, the environment is the compiler or verifier. To observe more detailed information about the fuzzing efficacy, we develop a tool called FuzzBoost based on program execution traces. In this respect, we record dynamic traces while running the testing production compilers, i.e., GCC, on generated programs. In compiler construction, a basic block of an execution trace is defined as a straight-line code sequence with no branches except for the entry and exit points, which is considered as one of the important atomic units to measure code coverage. In our method, we capture all the unique basic blocks $B(\mathrm{T}_P)$ concerning each execution trace T_P and calculate a store of all the unique basic blocks covered by the existing test suite I' to represent our measure of interest. In our implementation of FuzzBoost, the program execution trace is collected by Pin [18], a widely-used dynamic binary instrumentation tool. Pin provides infrastructures to intercept and instrument the execution trace of a binary. During execution, Pin inserts the instrumentation code into the input program and recompile the output with a Just-In-Time (JIT) compiler. We develop a plug-in of Pin to log the executed instructions. Additionally, we develop another coverage analysis tool based on the execution trace to report all the basic blocks touched so far. It also reports whether new basic blocks are triggered by the fuzzed program and the number of new covered blocks as well. Furthermore, our environment also logs and reports abnormal crashes, memory leaks, or failing assertions of compilers with the assistance of internal errors alarms from the compiling messages.

2.4 Reward

Designing a good reward function to facilitate learning and maintaining the optimal policy is the key goal in our framework. Rewards provide evaluative



```
foo(a, p) int *p; { p[0] = a;
a = (short)a; return a;} main
() { int i; foobar(i, &i); }
bar(a, b) { int c; c = a % b;
a = a / b; return a + b; }...
```

Fig. 2. Fuzz action prediction in the reinforcement learning process of compiler fuzzing

feedback to guide an RL agent to make decisions. However, rewards can be very sparse so that it is challenging for the learning problems. In the game of Go, a reward only occurs at the end of a game. In such cases, the learning process can converge slowly because of the sparse motivations. We solve this challenge by giving every mutation step a reward, so the goal of agent is to maximize the accumulated rewards until one episode terminates at step T,

$$R = \sum_{t=0}^{T} \gamma_t r_{t+1}(P), \tag{2}$$

where $\gamma_t \in (0, 1)$ indicates a discount factor to gradually deduct the reward in the future. $r_{t+1}(P)$ represents the reward of generated program P at step $t + 1$. In fuzz testing, the possible rewarding heuristics are program coverage, new crashes, timeout, etc. They aim at enlarging the analyzed surface in the target programs being fuzzed and digging into the program traces accordingly that are more suspicious. In compiler fuzzing, we adopt testing coverage as the reward to motivate the learning towards a vulnerability search on more areas of the compiler's code. However, unlike conventional definitions for coverage, which are usually line/function/branch coverage that require expensive computing resources to calculate, we define the reward based on the ratio of unique basic blocks covered by a certain fuzzed seed program P at step t to the entire unique basic blocks of its mutated test suites I' along the episode;

$$r(P, I') := B(\mathrm{T}_P) / \sum_{\rho \in I'} B(\mathrm{T}_\rho), \tag{3}$$

where $B(\mathrm{T}_P)$ is the number of unique basic blocks in the execution trace of a program P and $I' \subset I$ is the programs generated from this test suite. This stepwise reward r is a continuous scalar value that has a range of $(0, 1]$, where 1 is achieved when a specific execution trace covers all the basic blocks that have been tested so far by its existing fuzzed cases. The designed reward motivates the mutation steps towards the training goal: improving the compiler testing coverage by selecting a critical subsequence inside a seed program and enforcing simple mutations in a trace.

3 Designed Framework

To start a deep Q-learning process for compiler fuzzing, we propose FuzzBoost which adopts a deep neural network with two layers connected with non-linear activation functions. We build this end-to-end learning framework with the environment reward calculated based on dynamic trace analysis. In this section, we present the overall learning process for FuzzBoost by illustrating an iteration of fuzz action prediction in the reinforcement learning process for compiler fuzzing as shown in Fig. 2.

3.1 Initialization

We start with an initial input seed $P \in I$, where the choice of P is not constrained but can be any C program even not well-formed ones. We employ the GCC test suite as our sampling pool and randomly selected programs to be our seed inputs. We propose to use a neural network as the Q function to mimic the reasoning for input mutation of compiler fuzzing. This deep neural network maps states (embedding of an extracted substring from seed programs) to Q outputs for all actions A. Due to the lack of heuristics at the very beginning, the neural network is randomly initialized and reinforcely optimize the model parameters θ from the environment feedbacks, i.e., rewards, by maximizing the code mutation rewards in the episode training.

3.2 State Extraction

FuzzBoost observes a substring within a seed program to predict actions to perform. The substring is extracted from the seed program by the customized window and encoded as $S(P)$. In Sect. 2.1, we define the states of our Markov decision process to be $I = \Sigma*$. To be more specific, it is a substring P' at offset $o \in 0, ..., |P| - |P'|$ and of window size $|P'|$. To make the extracted state tractable, we define actions in Sect. 2.2 to shift and resize the window. By performing window-related actions, the fuzzing agent can see the whole program by partially observing fragments consecutively. In other words, FuzzBoost learns to select the most critical piece of code to mutate incrementally during the training process. After the sequence is extracted, we use a word embedding model to abstract the sequence into a fixed-dimensional vector for training.

3.3 Deep Q-Network

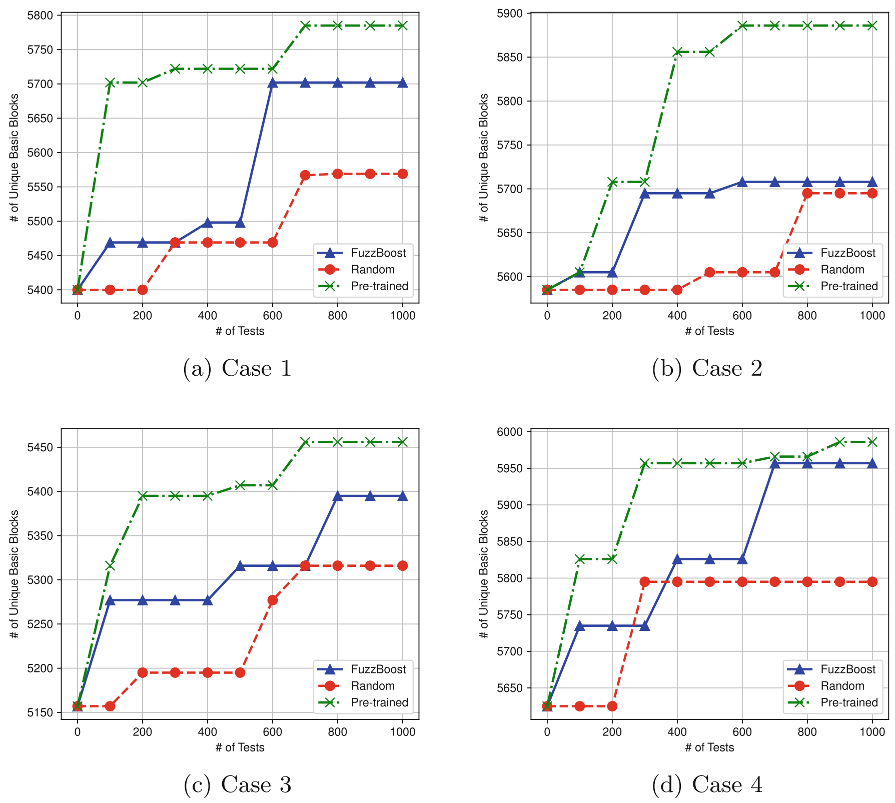
We implement the Q-learning module based on Tensorflow [1] 1.14. The deep neural network used for prediction is a forward neural network with two hidden layers connected with non-linear activation functions. The two hidden layers contain 100 and 512 hidden units respectively, and are fully connected with an input layer with 100 units (which is the max window size for input substring) and an output layer with 10 units (which is the size of action space). The goal of the training is to maximize the expected reward. Since the MDP is a finite horizon in our practical design, we adopt a discount rate $\gamma = 0.9$ to address the long-term reward. We set the learning rate $\alpha = 0.001$ to achieve our best-tuned results. We use the decayed epsilon-greedy strategy for exploration in the reinforcement learning iteration, that is, the ϵ value is set up to 1 at the very beginning and decays over time until a min value, 0.01 in our configuration, is reached. In this scenario, with the probability $1 - \epsilon$, the agent selects an action $a = argmax_{a'} Q(s_t, a_t)$, which is the estimated optimum by the on-training neural network. In the meanwhile, with probability ϵ, the agent explores any other actions with a uniformly distributed choice within the action space $|A|$.

3.4 Termination

A mutation episode terminates when the agent detects a terminal state. In our design, we define three conditions that may trigger the terminal state of mutating the seed program: (1) the agent executes the "terminate" action from the neural network prediction; (2) the generated program reaches a maximum number of mutation steps; or (3) the agent generates an invalid action that triggers miscellaneous effects during the reward calculation. The first type of termination will cut the program mutation actively by FuzzBoost while the latter two are passively ended with pre-defined policies. Theoretically, the mutation trace can be generated as long as possible to achieve enough diversity. But in practice, to excessively improve the testing efficacy, we empirically set up the mutation trace length to be 20 actions to enforce our agent to learn within the shortest path. To catch the found bugs/vulnerabilities, we log and report abnormal crashes, memory leaks, or failing assertions of compilers with the assistance of internal errors alarms from the compiling messages. Moreover, in our design, all the programs that have achieved higher code coverage are kept to be the seeds and waiting for another round of fuzzing, otherwise removed from the seed pool. Therefore, the agent can still explore the entire language set even with the restricted length of learning traces during an episode. The methodology applied in our mechanism is the same as conventional coverage-guided fuzzing methods but has made mutation traces *longer* in one round (compared with 1 step in conventional fuzzing) and *predictable* by a neural network (compared with purely random in conventional fuzzing).

4 Experiments

In our research, we propose a reinforcement learning framework FuzzBoost that incrementally trains a deep neural network to predict mutation actions on a given seed program to improve the compiler testing coverage effectively. We evaluate the performance of FuzzBoost on a seed input set gathered from the GCC test suites. We randomly sample 20 C programs in the test suite as our benchmark dataset, more specifically, from the *gcc.c-torture* repository. The window size is set to be 50 to extract the substring inputs. We run FuzzBoost for four weeks to test its fuzzing efficacy and compare with the baseline random fuzzing method used in a popular tool (AFL) [36]. We also conduct an empirical analysis on starting the compiler fuzzing with a pre-trained model to investigate if it can boost our process. All measurements are performed on i7-7700T 2.90 Ghz with 12 GB of RAM.



(a) Case 1 (b) Case 2

(c) Case 3 (d) Case 4

Fig. 3. Number of unique basic blocks covered by generated test suites

4.1 Fuzzing Efficacy

In our design, to improve the efficiency in this end-to-end learning process, we use an approximation of the code coverage improvement to describe the coverage information, which is the accumulated number of unique basic blocks being executed with the generated new test cases. In order to show that FuzzBoost learning algorithm learns to perform high-reward actions given a seed input observation, we compare the improved testing efficacy against a baseline with random action selection policy. The choice of the baseline method is uniformly distributed among the action space A and we terminate the actions with the same methodologies as our method described in Sect. 3.4. Random mutation is widely used in software fuzzing tools [36] which is proven to be effective while a good heuristic, such as coverage-guided, is designed.

Comparison: We perform the experiments with our method FuzzBoost and baseline method Random-based mutation strategy to fuzz each of the programs from the sampling pool. We respectively generate 1,000 new tests from seed programs for both strategies and record the accumulated number of unique basic

Table 1. Coverage improvements with different window size

Window size	50	60	70	80	90	100
Coverage improvement (%)	37.14	36.11	30.29	28.95	28.07	27.94

blocks along the execution trace. On average, our proposed method FuzzBoost achieves higher testing coverage by 37.14% than the Random-based mutation method in terms of the number of the accumulated unique basic blocks on the seed programs. We randomly select four seed programs and illustrate the coverage improvement of comparisons between baseline method and FuzzBoost in Fig. 3. The results in each sub-figure represent the number of unique basic blocks that different amount of test programs trigger in the compiler. We can see that FuzzBoost gradually increases the code coverage as the model being trained to mutate programs more effectively. Our method obviously outperforms the baseline for all cases, among which the most and least improvements, 79.17% (case 1, seed1.c) and 12.24% (case 2, seed2.c) respectively, are achieved. We also observe that FuzzBoost improves the code coverage with a faster speed than the baseline. We believe this is because our method can learn to fuzz more efficiently and generate interesting test suites with fewer mutation actions.

Window Size: Since the size of each seed program varies, and, arguably, the limited window size may restrict the diversity of mutation trace and thus put a constraint on exploring the entire seed program. As a result, the seed program cannot be thoroughly observed or mutated accordingly after one episode of fuzzing. In this part, we analyze the impact of the current framework with different window sizes on model effectiveness. We increase the initial window size $w = |P'|$ from 50 to 100 and measure the average coverage improvement to compare against the baseline strategy on seed1.c as the seeds in sample pool are generally short. Table 1 shows the experimental results. We can see the coverage improvement decreases while increasing the window size of the initially extracted substring. That is, smaller substrings are better to start with and to mutate the program than larger ones in our method. Our interpretation is that small windows narrow down the mutation space and thus reduce the action randomness, which may increase the possibility of learning a high-quality mutation trace for the model, especially when the model is highly under-trained in the beginning stage. It also indicates that our model is trained to learn better moves of small windows and accordingly select better action to improve coverage. Also, it is worth noting that the ultimate goal for fuzz testing is not the exploration of entire programs, but making control-flow changes within limited observations to boost the fuzzing process.

End State: We set up the compiler fuzzing as an end-to-end reinforcement learning framework. Unlike the problem of Go, the end state of FuzzBoost is not deterministic in all cases. In our design, we hard-code a limit on the length of mutation traces from the computation cost point of view, but theoretically, the traces can be endless to gain enough randomness and achieve the higher

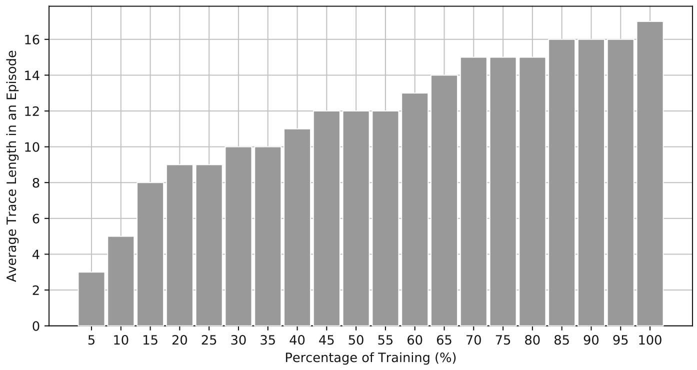


Fig. 4. Mutation length during training

reward. In the process of optimization, we provide the learning agent an action to actively terminate the episode which varies across the learning stage. Thus, to analyze how the end state evolves, we record the distribution of mutation trace lengths under different training stages. Figure 4 presents the average trace length distributions along the learning process over all training seeds. From the result, we can see that, as the training goes on, mutation trace lengths are increasing gradually. In this respect, the reward expectation of learned mutation actions are positive in a form that reinforces the model to dig more mutation opportunities in one episode to maximize the fuzzing reward.

4.2 Boosting with Pre-training

Our trained fuzzing tool learns to constantly accumulate the prior experience by training on the seed programs. This naturally lead us to the question for the sake of resource cost: given an agent which is pre-trained on seed programs $P_{train} = p_i \sim P$, can it improve testing efficiency than learning from scratch? To answer that, we use the same experimental setting as the experiments in Sect. 4.1 and reuse the seed programs from the initial 20 seed programs and craft another 9 α-equivalent programs for each seed respectively. We call a program P' is α-equivalent to program P when we only perform bound variable renaming on P. We randomly pick 80% of them to serve as P_{train} to learn an agent and the rest 20% are used for P_{test}. After pre-training on P_{train}, we save the model and use it on P_{test} to continue the trial-and-error reinforcement compiler fuzzing.

The fuzzing results under such a pre-trained model are shown in Fig. 3 and compared with the performance of FuzzBoost learned with an initially arbitrary model. The coverage improvement for the case of pre-trained model increases drastically towards the highest coverage against the one trained from scratch despite the small improvements in one of the seed programs (case 4). In addition, as the training goes on, the pre-trained model can find useful action in mutation space more quickly and generate fuzzed programs with high testing coverage.

```
1  foo (a, p)
2       int *p;
3  { p[0] = a;
4    a = (short) a;
5    return a;
6  }
7  main () {
8    int i;
9
10   foobar (i, &i);
11
12
13 }
14 foobar (a, b) {
15   int c;
16   c = a % b;
17   a = a / b;
18   return a + b;
19 }
```

Listing 1.1. Original

```
1  foo (a, p)
2       int *p;
3  { p[0] = foobar(a,p);
4    p = (short) a;
5    return a;
6  }
7  main () {
8    int i;
9    for (int a=8; a>0; a--) {
10     foobar (i, &i);
11   }
12   foobar(i, &i);
13 }
14 foobar (a, b) {
15   int c;
16   c = a % b;
17   a = c / b;
18   return a + b;
19 }
```

Listing 1.2. Mutated

4.3 Mutation Example

In this part, to demonstrate how effective FuzzBoost can achieve in program mutations for compiler fuzzing, we showcase the topmost utilized mutations in the following example. We present an original seed (on the left) and its corresponding new generations after mutations (on the right). We discuss each of these abstracted edits involved in the trace of atomic mutations. These edits help explain what is learned by the model during the reinforcement learning process. It should be noted that these mutations are not accomplished within one episode, while we just use this one example to illustrate what the most used mutations are and how they look like.

Example: By observing the results, we find: (1) the top most chosen mutation is *insertion*. Usually, the fuzzing engine tries to insert statements with keywords that do not exist in the original seed file. As shown in *line 9* to *line 11* in the *mutated file*, the fuzzing engine tries to insert a `for` statement into the seed file. By inserting these non-existing tokens, the compiler should execute the lexical analysis in a way that has not been used before; (2) the second chosen mutation is *replication* that the fuzzing engine tries to replicate statements locally as shown in *line 12* in the *mutated file*. The replication will trigger the compiler to optimize code which will improve the testing coverage; (3) the third chosen mutation is *replacement* that can replace a variable (`a`) with a function call (`foobar(a,p)`) as in *line 3* or replace a variable (`a`) with another existing variable (`p`). The replacement either makes the statement more complex to parse, causes exception handlings such as typecast, or changes the control-flow of the seed file, all of which will make the compilation different from the original paths, thereby increasing the testing coverage.

5 Discussion

It is critical to compare with related works, but we find it difficult to perform apple-to-apple comparisons. For instance, generation-based fuzzing tools, such as DeepSmith [7] and Learn&Fuzz [12], craft new programs from scratch other than mutating seed programs while our tool is built on mutation-based fuzzing that rely on seed programs to achieve the whole-program validity. Moreover, some previous methods [7,12] generate a bunch of new programs which usually get rejected at an early stage in compilation and therefore leads to a inefficient and shallow testing procedure. AFL [36] can generate new fuzzed inputs in a very fast way as it only conducts one-step random mutation on seed programs each time. However, it does not suit for compiler fuzzing because its mutation mechanism deals with random changes on inputs without considering their structure context. Compiler requires highly-structured and syntax-aware inputs, so we only compare our tool with its mutation heuristic in the paper. For NEUZZ [26], it is grey-box fuzzing that relies on the coverage analysis on target applications. But for compiler testing, the computation cost for code edge coverage is very high, and that is why we use $\#BasicBlocks$ tested as an approximation.

In this work, we do not claim our tool is better than others. Instead, we reveal our insight of leveraging the superiority of reinforcement learning for compiler fuzzing to efficiently solve a multi-step mutation-based fuzzing problem. In our mechanism, we lazy-evaluate the mutation results and consider those intermediate states in the mutation traces to explore code coverage in a deeper way. Our designed rewriting rules in mutation actions incorporate the structure context of programs, thus our fuzzed inputs can better conform with the syntax requirements of programming languages. What's more, the mutations can realize the comprehensive search in the large space to iteratively guiding the tool for the final fuzzing goals. Our experimental results and analysis comprehensively demonstrate the effectiveness of our compiler fuzzing tool.

6 Related Work

Our study is related to deep reinforcement learning and mutation-based fuzzing.

Deep Reinforcement Learning: Despite the popularity in solving the game of Go, reinforcement learning is also widely adopted as a powerful technique for program synthesis [2,5,13,16,17,22,26]. Bunel et al. [6] perform reinforcement learning on top of a supervised model with an objective that explicitly maximizes the likelihood of generating semantically correct programs. Böttinger et al. [5] use a deep Q-learning network to learn a grammar description for inputs to perform generation-based fuzzing. Researchers also propose Neurally Directed Program Search (NDPS) [31], for solving the challenging non-smooth optimization problem of finding a programmatic policy with maximal reward. Existing projects that adopt deep reinforcement learning for program synthesis focus on semantic goals toward synthesis tasks. Our target is to generate source programs that are well-formed but contain different syntactic features, which are similar to the work from Böttinger et al. [5] that aims at PDF parser fuzzing. But differently, in our design, we consider the improvement of testing coverage of compilers as the reward for reinforcement learning.

Mutation-Based Fuzzing: Mutation-based fuzzing contains two important decisions: 1) where to mutate, and 2) what new value to use for the mutation [24]. Generally mutation-based fuzzers are not aware of the expected input format or specifications, and they cannot select mutations very wisely [25]. It generates new inputs by blindly modifying the provided seeds. A well-known fuzzer that is mutation-based is called AFL [36] which randomly mutates seed inputs and incrementally add new seeds into the set with respect to defined heuristics. Several boosting techniques are proposed to improve the efficiency of mutation-based fuzzing. AFLFast [4] boosts up original AFL fuzzer by focusing on low-frequency paths that allow the fuzzer to explore more paths with limited time. Skyfire [32] applies grammar in existing seed inputs for fuzzing programs that take highly-structured inputs. Kargen and Shahmehri [14] perform mutations on the machine code instead of on a well-formed input to produce high-coverage inputs. DeepFuzz [17] utilizes an RNN-based model to generate new well-formed C programs for compiler fuzzing based on existing testsuites. In this paper, our method boosts the mutation process by using a deep neural network to predict the mutation without any training data.

7 Conclusion

In this paper, we propose FuzzBoost, a deep reinforcement learning framework to fuzz off-the-shelf compilers by generating new programs with coverage-guided dynamics. Our proposed end-to-end learning framework learns to select a trace of best mutation actions in each round towards high code coverage and performs automatically without any human supervision. It improves the testing coverage on a seed set from the GCC test suites and outperforms the baseline fuzzing agent with a random selection strategy. Moreover, we demonstrate that a pre-trained agent in our framework can generalize the strategy to new seed instances to expedite the fuzzing process, which is much faster than starting from scratch.

Acknowledgement. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used for this research. This research was supported in part by the National Science Foundation (NSF) grant CNS-1652790 and the Office of Naval Research (ONR) grant N00014-17-1-2894.

References

1. Abadi, M., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2016), pp. 265–283 (2016)

2. Becker, S., Abdelnur, H., State, R., Engel, T.: An autonomic testing framework for IPv6 configuration protocols. In: Stiller, B., De Turck, F. (eds.) AIMS 2010. LNCS, vol. 6155, pp. 65–76. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13986-4_7
3. Bertsekas, D.P., Tsitsiklis, J.N.: Neuro-dynamic programming: an overview. In: Proceedings of the 34th IEEE Conference on Decision and Control, Piscataway, NJ, pp. 560–564. IEEE Publications (1995)
4. Böhme, M., Pham, V.T., Roychoudhury, A.: Coverage-based greybox fuzzing as Markov chain. IEEE Trans. Softw. Eng. **45**(5), 489–506 (2017)
5. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 116–122. IEEE (2018)
6. Bunel, R., Hausknecht, M., Devlin, J., Singh, R., Kohli, P.: Leveraging grammar and reinforcement learning for neural program synthesis. arXiv preprint arXiv:1805.04276 (2018)
7. Cummins, C., Petoumenos, P., Murray, A., Leather, H.: Compiler fuzzing through deep learning. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 95–105. ISSTA (2018)
8. Duran, J.W., Ntafos, S.: A report on random testing. In: ICSE, pp. 179–183 (1981)
9. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Trans. Softw. Eng. **SE-10**(4), 438–444 (1984)
10. Gan, S., et al.: CollAFL: path sensitive fuzzing. In: 2018 IEEE Symposium on Security and Privacy, pp. 679–696. IEEE (2018)
11. GCC, The GNU Compiler Collection. gcc.gnu.org (2019). http://gcc.gnu.org/
12. Godefroid, P., Peleg, H., Singh, R.: Learn&Fuzz: machine learning for input fuzzing. In: Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pp. 50–59. IEEE Press (2017)
13. Kaelbling, L.P., Littman, M.L., Moore, A.W.: Reinforcement learning: a survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
14. Kargén, U., Shahmehri, N.: Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 10th Joint Meeting on Foundations of Software Engineering, pp. 782–792. ACM (2015)
15. Kifetew, F.M., Tiella, R., Tonella, P.: Combining stochastic grammars and genetic programming for coverage testing at the system level. In: Le Goues, C., Yoo, S. (eds.) SSBSE 2014. LNCS, vol. 8636, pp. 138–152. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09940-8_10
16. Li, X., Liu, X., Chen, L., Prajapati, R., Wu, D.: ALPHAPROG: reinforcement generation of valid programs for compiler fuzzing. In: Proceedings of the Thirty-Fourth Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-2022) (2022)
17. Liu, X., Li, X., Prajapati, R., Wu, D.: DeepFuzz: automatic generation of syntax valid C programs for fuzz testing. In: Proceedings of the 33rd AAAI Conference on Artificial Intelligence (2019)
18. Luk, C.K.: Pin: building customized program analysis tools with dynamic instrumentation. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 190–200 (2005)
19. Miller, B.P., Fredriksen, L., So, B.: An empirical study of the reliability of UNIX utilities. Commun. ACM **33**(12), 32–44 (1990)
20. Mnih, V., et al.: Playing Atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)

21. Mnih, V., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529 (2015)
22. Rajpal, M., Blum, W., Singh, R.: Not all bytes are equal: neural byte sieve for fuzzing. arXiv preprint arXiv:1711.04596 (2017)
23. Rash, M.: A collection of vulnerabilities discovered by the AFL fuzzer (AFL-fuzz) (2019). https://github.com/mrash/afl-cve
24. Rawat, S., Jain, V., Kumar, A., Cojocar, L., Giuffrida, C., Bos, H.: VUzzer: application-aware evolutionary fuzzing. In: NDSS, vol. 17, pp. 1–14 (2017)
25. Saavedra, G.J., Rodhouse, K.N., Dunlavy, D.M., Kegelmeyer, P.W.: A review of machine learning applications in fuzzing. arXiv preprint arXiv:1906.11133 (2019)
26. She, D., Pei, K., Epstein, D., Yang, J., Ray, B., Jana, S.: NEUZZ: efficient fuzzing with neural program smoothing. In: 2019 IEEE Symposium on Security and Privacy, pp. 803–817. IEEE (2019)
27. Sun, C., Le, V., Zhang, Q., Su, Z.: Toward understanding compiler bugs in GCC and LLVM. In: ISSTA, pp. 294–305 (2016)
28. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute Force Vulnerability Discovery. Pearson Education, London (2007)
29. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (2018)
30. Takanen, A., Demott, J.D., Miller, C., Kettunen, A.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, Norwood (2018)
31. Verma, A., Murali, V., Singh, R., Kohli, P., Chaudhuri, S.: Programmatically interpretable reinforcement learning. arXiv preprint arXiv:1804.02477 (2018)
32. Wang, J., Chen, B., Wei, L., Liu, Y.: Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy, pp. 579–594 (2017)
33. Wang, M., et al.: SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceedings, pp. 61–64. ACM (2018)
34. Watkins, C.J., Dayan, P.: Q-learning. Mach. Learn. **8**(3–4), 279–292 (1992)
35. You, W., Liu, X., Ma, S., Perry, D., Zhang, X., Liang, B.: SLF: fuzzing without valid seed inputs. In: Proceedings of the 41st International Conference on Software Engineering, ICSE (2019)
36. Zalewski, M.: American fuzzy lop (2014)