RiverFuzzRL - an open-source tool to experiment with reinforcement learning for fuzzing

Ciprian Paduraru
Department of Computer Science,
University of Bucharest, Romania
ciprian.paduraru@unibuc.ro

Miruna Paduraru
Department of Computer Science
University of Bucharest, Romania
miruna.paduraru@drd.unibuc.ro

Alin Stefanescu
Department of Computer Science,
University of Bucharest, Romania
alin.stefanescu@unibuc.ro

***Abstract*—Combining fuzzing techniques and reinforcement learning could be an important direction in software testing. However, there is a gap in support for experimentation in this field, as there are no open-source tools to let academia and industry to perform experiments easily. The purpose of this paper is to fill this gap by introducing a new framework, named RiverFuzzRL, on top of our already mature framework for AI-guided fuzzing, River. We provide out-of-the-box implementations for users to choose from or customize for their test target. The work presented here is performed on testing binaries and does not require access to the source code, but it can be easily adapted to other types of software testing as well. We also discuss the challenges faced, opportunities, and factors that are important for performance, as seen in the evaluation.**

***Index Terms*—fuzzing, reinforcement learning, binary analysis, symbolic execution, open-source tool**

1. Introduction

Fuzzing techniques obtained important results in software testing [1]. Lately, the trend is to use artificial intelligence (AI) to guide the fuzzing process for better results. Reinforcement learning (RL) is an interesting field of machine learning (ML), which, if applicable, could lead to agents that take intelligent decisions with less human supervision. The combination of RL and fuzzing could be valuable, but there are several challenges to be addressed in order to make the experimentation process easier for both researchers and industry practitioners.

The main purpose of our work is to describe an open-source tool that can be used to improve fuzzing using RL strategies. Over the previous work in the field we provide more flexibility for the user control over the fuzzing process. First, there is no open-source solution that provides an RL environment to do fuzzing, other than our work in [2], which uses a contextual bandits solution to optimize the inputs selection for a generational concolic execution type.

In this paper we address the problem of implementing an open-source generic fuzzing solution for binary programs. There are a couple of reasons why we target binary programs especially, despite their difficulty. First, it is related to our plans to move towards testing IoT software, where most of the solutions run in embedded systems and LLVM or code recompilation are difficult to apply. Second, there are many scenarios where companies need to integrate software, provided directly as a binary, from third-party providers (firmware), without access to source code. Thus, it is impossible to test such components considering recompilation to inject own testing code mechanisms in the final build. Having said that, the novelties presented from both theoretical and practical points of view in this paper can also be applied to white-box testing where access to the entire source code is provided.

Our contributions are the following:

- It is the first open-source tool that provides a solution for experimenting with fuzzing and RL.
- Our solution allows customization of the RL environment from different dimensions, which is superior to the work in the literature: state representation, action and rewards customization, model selection, and training algorithms. A full comparison is presented in Section 4.
- We allow stateful programs testing, i.e., the programs under test are evaluated under a sequence of inputs, so there is no need to reset the program state each time.
- A combination of a classic forward tracing process with a symbolic constraints propagation is also possible in our environment. This was not documented previously in the literature for reinforcement learning and fuzzing processes.
- The existing work is limited only to Deep Q-Network (DQN) models evaluation. A novelty in our work is the separation of the two concerns: (a) defining a testing environment that allows users to frame their fuzzing ideas as an RL formulation, and (b) defining the RL methods used to train and use that environment. As an example to show how customizable our fuzzing tool is in terms of methods for training models, we link a public library of RL algorithms, TF-Agents [3].

River - an open-source testing framework. We base our work on our River open-source tool available at https://river.cs.unibuc.ro. Currently, we investigate methods for fuzz testing guided by AI algorithms, testing deployment of software over IoT devices, and collaborate with industry to bring our tools intro practical use. The solution is cross-platform, working at the moment for guided fuzzing for binaries on x86, x64, ARM32, and ARM64. In this paper we present a new component, *RiverFuzzRL*[1].

Modeling fuzzing as a reinforcement learning problem. In RL terminology, the main objective is to adapt an *agent*'s behavior to interact with a system (*environment*) by acting such that it learns to maximize the rewards received over time. The *reward* is the feedback value coming from the environment after each agent's action. Translating this to fuzzing, we can imagine that the *agent* is the Fuzzer, which performs modifications/actions over the input given to the program under test to achieve the desired testing target, such as more code coverage. The program being tested, or a wrapper above it, is the *environment* itself. The *reward* value can be considered as a metric to how well did the agent performed in modifying the input buffer to achieve the desired testing target (e.g., code coverage, count of low probability paths, execution time taken by the program, etc.). The agent's knowledge is a set of *observations*, which are usually partial in a testing process. The set of these observations can be drawn from complex mechanisms such as LLVM sanitizers or taint analysis. Better observations for the agent usually mean more time or requirements to obtain them, but this can improve the overall reward results. The target of the training process is to find a policy that tells the agent how to act given the observations, i.e., $\pi(a|S)$, the probability of choosing an action a from the possible modifications allowed over the input space, given the set of observations encapsulated in state S. The training process is done using *episodes*, which are a limited sequence of actions performed. This limit is usually represented either by entering a state considered as final (e.g., crash/exception), or when reaching a certain maximum number of steps. One of the main challenges in training RL agents is that the rewards are usually delayed, i.e., it needs more actions to get to an important reward, as it is the case with fuzzing.

For a more formal description of the problem, we refer the reader [4], which defines the problem of fuzzing as a Markov Decision Process, and [5], which contains a collection of proofs and methods of different RL algorithms and techniques.

2. Related work

Our work focuses on fuzz testing combined with online symbolic execution. In this section we evaluate the literature that we found important and which provided ideas on how to combine these two aproaches with a RL solution.

General fuzzing techniques [1], [6], [7] have their own advantages over symbolic/concolic testing when testing large codebase applications. In this field, the techniques are divided in three main categories: blackbox random fuzzing, whitebox random fuzzing, and grammar-based fuzzing. The blackbox fuzzing methods can be augmented with other strategies for better results. An example is the work in [8] (also a component of the River tool) and AFL (American Fuzzy Lop) [9], which use genetic algorithms and various heuristics to find vulnerabilities and achieve an improved code coverage over similar tools.

In the field of online symbolic execution (aka concolic execution), most of the frameworks such as KLEE [10], S2E [11], DART [12], or CUTE [13], work at the source code level. CRETE [14], which is based on KLEE, operates on the LLVM representation. However, our River tool is able to perform directly at binary level, without access to the source code.

Our previous work in [2] uses RL, more specifically a contextual bandits solution, to optimize the input selection for a generational concolic execution type. A similar work to reduce the exploration space in symbolic execution and avoid initial path explosion is documented in [15]. Worst-case complexity solutions using DQN were proposed in PySE [16] and [17]. They apply DQN on online symbolic execution to promote longer execution paths and penalize actions that do not lead to satisfiable constraints. In [18], the authors use RL in continuous software integration testing process. Their solution is to rank a test suite, then the most promising ones are scheduled up to an estimated time limit for testing at each integration cycle. The duration and previous history verdict (i.e., fail or pass) are considered as features in the learning process. Their reward functions try to promote the failing test first by penalizing passed tests over failing ones. Recent advances in this direction can be found in [19].

The most relevant work to ours are [4] and [20]. The former considers an RL environment tailored to PDF files given as inputs using DQN as learning strategy and does not require access to the source code of the parser. The latter applies also DQN on general purpose fuzzing inputs, but uses LLVM sanitizers as input for the policy learning. Thus, it does not work at binary level and needs source code access. Note that none of the two solutions are open-source and that they provide no entry-point to allow user customization regarding the techniques and states used. A detailed comparison of our work to them is presented in Section 4.

3. Tool architecture and implementation

We build a fuzzing solution using RL extensible by users from multiple points of view: state (observation) representation, actions (including dictionary tokens), rewards, and algorithms used for the training of agents.

First, we do a separation of concerns that splits the problem of fuzzing with RL in two different parts as seen in Fig. 1:

A. Defining the fuzzing environment. Here the user can frame the fuzzing problem in a typical RL formulation, as shown on the right side of Figure 1. The abstract base class of the fuzzing environment is the OpenAI Gym environment

1. see https://github.com/unibuc-cs/river/tree/master/River3/FuzzRL for the source code and experiments accompanying the paper

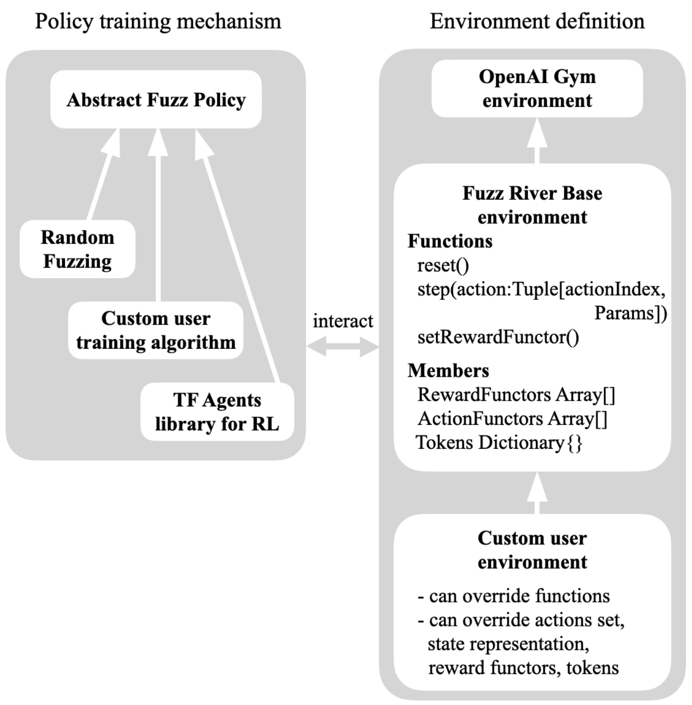

Figure 1: The architecture of the RiverFuzzRL. The left side represents the method used for learning a policy for performing fuzzing. The user is free to implement their own solution or link one of the public libraries of RL algorithms to speed-up experiments. The right side contains the user tested environment framed as an RL problem. At the core of it, we use the standard OpenAI Gym environment specification [21].

[21]. The default class providing out-of-the-box various parameterized implementations provided in our framework is named *Fuzz River Base*. This object can be further inherited by objects with two possible purposes: (a) selecting which of the default functionalities should still hold in the users' experiments, and (b) adding users' customized functionalities for different pieces in the RL formulation. Below is the current full list of components that are provided by default or can be customized by the user:

1) *Environment Reset function.*
   By default, the environment reads a collection of input examples from a folder path, if any is given, and randomly selects one of them as the current input buffer. This is similar to other fuzzing solutions, but the user is able to override this functionality and inject their own initial input semantics.
2) *Observations.*
   First, users must decide how the observation or state should be represented in their application. Typical examples used in the literature include: the current input buffer, a window of this, code coverage information, LLVM sanitizers output, last path of basic blocks explored by the fuzzer etc. With these in mind, we allow the user (through parameter configuration) to use a set from the following default observations:
   a) The last input $\mathbf{I}$ used to evaluate the program.
   b) A list containing the path that the last input used produced as a sequence of basic block addresses: $P = [B_1, \ldots, B_n]$.
   c) The same list as above, but represented as a dictionary mapping from basic block addresses to the number of occurrences along the path P (e.g., if there is a loop, the above representation will contain in a plain sequence all the iterations, while this one aggregates the same basic block into a single entry).
   d) If a symbolic execution action (see next section) is used, for each B_i in P, if B_i is conditional, the tool also returns a list of branch destinations available from B_i and the SMT constraint condition that needs to be solved to get to the given branch destination, as well as the index of the branch taken in the previous execution of the input. Formally, we represent this as a dictionary, as exemplified in the following: $P = [B_1 : None,\ B_2 : \{(LastTaken, B_{2_1)}), (B_{2_0}, Cond_1), (B_{2_1}, Cond_2)\}$, $B_3 : \{(LastTaken, B_{3_0}), (B_{3_0}, Cond_3), (B_{3_1}, Cond_4)\}, \ldots, B_n]$. Here, the first branch along the path is a non-conditional branch, while the second and third branches have each two possible continuations.
   e) A 2D hash-map representation of the basic block addresses of the program (similar to the one in [9]).
   f) An LSTM (long-short term memory network) based embedded space of the path of binary blocks for P. A fixed size representation of the path is useful, e.g., when trying to detect the longest execution runs along a program via an approximation function based on deep neural networks. A similar implementation for the same problem is suggested by [16].
3) *Actions.*
   Second, the actions chosen by the RL agents must be specified. Their role is to change the current input buffer in a way that comes closer to reach the testing target purpose (i.e., increase the rewards obtained over time). By default, we offer all the actions available in *libFuzzer* [20] (e.g., changing, erasing, adding bytes at given locations, adding words from a tokens dictionary etc.), and some other functionalities such as text based-inputs similar to shuffling objects tags in [4]. These actions can be specified by a numeric action index and a fixed parameter size. The dictionary token used to add plain words can also be customized depending on the user application. For instance, in our example of fuzzing efficiently a PNG parser library, the dictionary tokens are overridden to allow for custom tags specific to the PNG input files, such that the words added by the actions have a higher probability of being accepted by the parser program under test.
4) *Step function and rewards*
   The *step* in a fuzzing application using an RL environment is represented by giving the current input buffer as

input to the program under test for execution. A novelty of our framework is that actions can be executed under a taint analysis propagation mechanism along with a symbolic execution gathering of path conditions (as specified above in the observations description). We present the motivation for this in Section 4. Each step parameter contains a value defining if the current step should be done symbolically or not. If the option is used, SMT conditions are gathered along each branch. A common solver inside our framework, such as $Z3$, can be used to allow users to solve conditions and modify the current input to take some specific desired branches.

Reward function customizations are also important, since they allow the user to define what do they want to obtain as a testing target. The implementation of this customization type is done by using a default array of possible reward functors (containing two entries by default, as detailed and motivated in the next section), with the user being able to extend this with their own implementations.

The step function also returns if the current episode is done or not. Usually this is done when a crash/exception occurred along the testing process, or when a certain specified maximum number of steps were executed within the episode.

B. Defining the method to train and use agents on the given environment. For proving that our separation of concerns works as expected, we link to our framework a library of algorithms for reinforcement learning - TF Agent [3]. This is done as an example in our current implementation, but any other similar library of algorithms would work similarly. Thus, the user can easily evaluate many other off-the-shelf methods for training their customized environments. As shown in the left part of Fig. 1, the users can also implement their own custom methods.

A note on stateful vs stateless testing. Previous work considered application of RL on stateless programs, i.e., the memory operating on should be reset after each input sent, in order to have a deterministic reproducibility of the action. However, there are cases when programs start exhibiting issues after a certain sequence of inputs sent. To support stateful programs evaluation, our RiverFuzzRL framework allows an option to not reset the program's memory state between inputs, such that, if an episode ends with a crash, the entire sequence of inputs sent during the episode is considered as the one leading to the issue.

4. Details and comparison

In this section, we go deeper into the details and motivate the need for extensibility for a fuzzing solution using RL methods. This is done by comparing our results to the most relevant paper published up to the moment, which also uses RL for fuzzing.

State representation. In previous works, the user is limited to fixed representations of the observation space. For example, in [20] they consider as state representation the entire array of bits sent as inputs to libFuzzer being limited to 4096 bytes. In [4], the authors consider state representation for PDF objects fuzzing. Since such inputs can be large, their solution is to select only a window of the original input with an offset and window size being part of the RL training process. Also, both papers consider the code coverage as the main metric, but what if the user is trying to find the inputs corresponding to the longest running time of the program? Since they use either LLVM sanitizers output or correlations between the window where the input is applied and action made, there is no way for the algorithm to learn to produce the longest execution times, and try to go as deep as possible into loops.

Our framework gives the user a hook function to call and process the current raw input. We support for experimentation purposes both custom implementations in the previous works. However, using our tools, the user has full control over the state representation format, which could lead to more research opportunities based on the state representation that leads to improved reward values. For example, for text files or image parser types of inputs, users could select a set of convolutions to project the original input space to smaller representations. These compressed inputs can still hold the latent characteristics of the original input, without needing to cut it and thus potentially loose features from it. A positive aspect of this approach is also presented in our evaluation section, where we test an XML parser.

Action selection. The work in [20] uses all the actions available by default in libFuzzer set, while [4] uses only a small subset of these, oriented towards their testing target (PDF objects). As an example, the shuffle operation in their case is able to shuffle PDF sub-objects given input frame state. This is an example of custom action that intuitively keeps a correct structure of a PDF object (input), which could improve the code coverage in the end since the input produced after applying this action has more chances of being accepted by the program this way rather than a default random shuffling of bytes.

In this perspective, we provide by default the set of actions available in both works for result reproducibility, but also let the user add their own actions by registering custom hand-written actions functions. Fine tuning the actions set is an important factor in potential results of the fuzzing process. For example, by using our framework the user can add to the fuzzing process only a set of bytes known as useful, rather than letting the fuzzing process going through all the possible 256 values of a byte. Using techniques such as taint analysis, they can select an action to direct only on the areas of the input that affect the branch results [22]. Also, many of the parameters in the action space in previous work, based on libFuzzer for example, do not consider the sub-parameters space of the action. As an example, in [20], the action *ChangeByte* randomly changes a byte in the input string. But maybe it has an importance with which value range is the input changed and the location of the change. This is similar to the rest of their action space: there is no user parameters regarding this fine-tune selection of actions parameters. In our solution, we let the user customize this through registering actor functors and sending parametrized spaces to restrict the value used in the fuzzing process. This is technically implemented using nested types in the OpenAI Gym environment definition.

Reward function customization. By default we offer implementations for two types of reward functions: (a) one targeted for code coverage (1), which simply rewards the agent when choosing an action with better code coverage in a state S at time t, and (b) another one rewarding longer execution paths (2):

$$R_t(S_t, a_t) = cov_t - cov_{t-1}, \tag{1}$$

$$R_t(S_t, a_t) = execTime_t - execTime_{t-1}. \tag{2}$$

More detailed reward function examples are provided in [2], which also takes into account the potential parallelism opportunities to divide the testing process and to reward continuous integration testing.

Agents customization. Different types of RL methods could have various impacts depending on the problem. It is the case in software testing too, where, depending on the observation representation and the set of available actions, an agent could either be blocked in local optima or be too slow to train. Next section shows an experiment that points towards allowing quick evaluation of different RL methods other than the DQN models used in the existing work.

5. Evaluation

Having the framework in place, we succeeded to derive custom components and reproduce easily the methods in [20], [4], [16] and their results. This was indeed one goal when implementing the tool architecture and support - to minimize the time effort and coding challenge that need to be addressed for experimenting with such methods. It is also important to note that policies learned for one particular application can still provide good results in other type of applications, [20], [2]. However, further investigations are needed to evaluate properly the success of policy transfer between different clusters of applications such as text-based to image-based parsers and others.

Challenges. During the development of this tool, we had to address a few challenges:

- What could be a solution for getting out of local optima, i.e., states where the agent is having difficulties in obtaining new, rare paths?
- What is the right architecture to provide as much extensibility as possible for users' experiments?
- What should be the default set of implemented actions, observations, and rewards to allow fast experiments for community?
- What are the best default parameters for training vs inference speed, and how can we provide an interface for users to control these?

Lessons learned and experiments. The following experiments were done on a cluster of 2 CPUs Intel Xeon

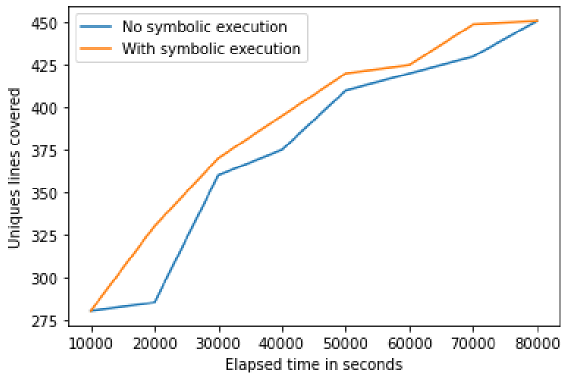

Figure 2: Effect of using symbolic execution to agents steps when the reward was not improving after $T = 50$ episodes.

.

3.7 Ghz, and 4 Nvidia Titan V video cards. The models definition and training was done using TensorFlow 2.

The work in [20] presented the difficulties of different fuzzing methods to produce inputs that obtain branches soon enough to exceed the barrier of some code coverage thresholds. This is mainly a problem because fuzzers, either using heuristics or RL, have difficulties in modifying branches with the exact values needed to take certain paths along the program. Online symbolic execution can be used to have better information about the conditions of each branch point. Considering this, we investigated (see results in this section) what is the impact of implementing a custom solution to use both methods at the same time. The results as shown in Fig. 2 suggest that RL agents can get out of local optima faster using a combination of these, without affecting performance, when tuned properly. In this experiment, we added symbolic execution steps when the reward signal was not improving after a certain number of episodes ($T = 50$ in our simulation). This was followed by a next custom action that changed the input to get one (random) branch that was not evaluated before.

As mentioned before, the literature for combining fuzz testing with RL considers only DQN methods. To verify if there are other RL techniques that work better, we customized a solution to use a policy-gradient (PPO [23]) from the TF Agents library for the problem of obtaining the longest execution path from two binary programs that were also evaluated in [16]: binary tree search and insertion sort (note that the evaluation was done on binaries, without access to the source code). Table 1 shows the training time needed to get the corresponding results with both methods, proving as expected in theory that a policy-gradient method works better for this type of scenarios. Intuitively, the probability of decisions that lead to loops' evaluations is increased faster in the training process when the agent observes what leads to better rewards.

Custom actions can also speed-up the agent convergence. As example, we evaluated an XML parser library [2], where

2. https://hub.docker.com/r/ouspg/libfuzzer-libxml2

TABLE 1: Comparative training time (in minutes) to obtain an agent that achieves longest execution path on a given problem using DQN vs PPO methods.

Problem	*DQN*	*PPO*
insertion sort	997	812
binary search tree	1127	935

an agent trained using DQN using our framework succeeded to converge to the maximum numbers of line code coverage with a speed-up of 29% over [20], when choosing a subset of actions from [20] (changing, adding bytes and dictionary values only), together with the shuffle techniques from [4], which just switched tag nodes inside the XML input buffer.

An important optimization aspect when using deep networks that map an observation to the action-value pair, for example in DQN models (and not only), is the frame-skipping technique [24]. Initially, the technique was used for training Atari games and it involves sticking with a select actions for more frames instead of taking decisions on each frame of the game. This optimization is transferable to fuzzing too, and was addressed in [20]. Our framework allows a parameter to control the rate for action selection using the estimation model.As an example, if $FrameSkipRate = 3$, then a decision is made using the current learned policy at each 3 steps, otherwise a random action is selected. The role of this method is to let the fuzzing process go further since it is faster and to intervene from time to time with the smart agent. This could be an important aspect in complex networks, where it takes too much time to evaluate the optimal policy at each step.

6. Conclusion

Reinforcement learning is a promising direction in the field of software testing. This paper presented the ideas behind our tool that makes experiments easier for users and developers of fuzzers, although expertise in both domains is required. We hope that our work will trigger many new ideas and better experiments in the area. As future work, we plan to extend our framework for testing not only standalone applications, but also the interaction between many such applications as well as their communication protocols, e.g., IoT-deployed software. Our approach can be easily adapted to other types of test objectives such as input test case prioritization [19], white-box fuzz testing [1], or search-based testing [25].

Acknowledgements

This work was supported by a grant of Romanian Ministry of Research and Innovation, project no. 401PED/2020.

References

[1] P. Godefroid, "Fuzzing: hack, art, and science," *Commun. ACM*, vol. 63, no. 2, pp. 70–76, 2020.

[2] C. Paduraru, M. Paduraru, and A. Stefanescu, "Optimizing decision making in concolic execution using reinforcement learning," in *ICST'20 Workshops*. IEEE, 2020, pp. 52–61.

[3] S. Guadarrama *et al.*, "TF-Agents: A library for reinforcement learning in TensorFlow," https://github.com/tensorflow/agents, 2018.

[4] K. Böttinger, P. Godefroid, and R. Singh, "Deep reinforcement fuzzing," in *SP'18 Workshops*. IEEE, 2018, pp. 116–122.

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Bradford Books, 2018.

[6] A. Zeller, R. Gopinath, M. Böhme, G. Fraser, and C. Holler, "The fuzzing book," 2019, https://www.fuzzingbook.org.

[7] M. Sutton, A. Greene, and P. Amini, *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley, 2007.

[8] C. Paduraru, M. Melemciuc, and A. Stefanescu, "A distributed implementation using Apache Spark of a genetic algorithm applied to test data generation," in *GECCO'17 Workshops*. ACM, 2017, pp. 1857–1863.

[9] AFL, http://lcamtuf.coredump.cx/afl.

[10] C. Cadar, D. Dunbar, and D. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI'08*. USENIX Association, 2008, p. 209–224.

[11] V. Chipounov, V. Kuznetsov, and G. Candea, "The S2E platform: Design, implementation, and applications," *ACM Trans. Comput. Syst.*, vol. 30, no. 1, pp. 2:1–2:49, 2012.

[12] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed automated random testing," in *PLDI'05*. ACM, 2005, pp. 213–223.

[13] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," in *ESEC/FSE'05*. ACM, 2005, pp. 263–272.

[14] B. Chen, C. Havlicek, Z. Yang, K. Cong, R. Kannavara, and F. Xie, "CRETE: A versatile binary-level concolic testing framework," in *FASE'18*, ser. LNCS, vol. 10802. Springer, 2018, pp. 281–298.

[15] J. Wu, C. Zhang, and G. Pu, "Reinforcement learning guided symbolic execution," in *SANER'20*. IEEE, 2020, pp. 662–663.

[16] J. Koo, C. Saumya, M. Kulkarni, and S. Bagchi, "PySE: Automatic worst-case test generation by reinforcement learning," in *ICST'19*. IEEE, 2019, pp. 136–147.

[17] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double Q-learning," in *AAAI'16*, 2016, pp. 2094–2100.

[18] H. Spieker, A. Gotlieb, D. Marijan, and M. Mossige, "Reinforcement learning for automatic test case prioritization and selection in continuous integration," in *ISSTA'17*. ACM, 2017, p. 12–22.

[19] M. Bagherzadeh, N. Kahani, and L. C. Briand, "Reinforcement learning for test case prioritization," *CoRR*, vol. arXiv abs/2011.01834, 2020.

[20] W. Drozd and M. D. Wagner, "FuzzerGym: A competitive framework for fuzzing and learning," *CoRR*, vol. arXiv abs/1807.07490, 2018.

[21] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *CoRR*, vol. arXiv abs/1606.01540, 2016.

[22] C. Paduraru, M. Melemciuc, and B. Ghimis, "Fuzz testing with dynamic taint analysis based tools for faster code coverage," in *ICSOFT'19*. SciTePress, 2019, pp. 82–93.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. arXiv abs/1707.06347, 2017.

[24] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *ICLR'16 Posters*, 2016.

[25] J. Kim, M. Kwon, and S. Yoo, "Generating test input with deep reinforcement learning," in *SBST'18*. ACM, 2018, p. 51–58.