DeepRNG: Towards Deep Reinforcement Learning-Assisted Generative Testing of Software

Chuan-Yung Tsai[†], **Graham W. Taylor**[†‡]
[†]Vector Institute for Artificial Intelligence, Toronto, Canada
[‡]School of Engineering, University of Guelph, Guelph, Canada
`kenyon.tsai@vectorinstitute.ai`, `gwtaylor@uoguelph.ca`

Abstract

Although machine learning (ML) has been successful in automating various software engineering needs, software testing still remains a highly challenging topic. In this paper, we aim to improve the generative testing of software by directly augmenting the random number generator (RNG) with a deep reinforcement learning (RL) agent using an efficient, automatically extractable state representation of the software under test. Using the Cosmos SDK as the testbed, we show that the proposed DeepRNG framework provides a statistically significant improvement to the testing of the highly complex software library with over 350,000 lines of code. The source code of the DeepRNG framework is publicly available online.[1,2]

1 Introduction

Machine learning (ML) has become a popular tool in assisting various software engineering needs [1], with successes like code writing services provided by GitHub Copilot [8] and OpenAI Codex [2]. Software testing, as an equally critical and challenging topic, on the other hand, has yet to receive as much attention in the ML community, even though the number of and the damage caused by software vulnerabilities have been rapidly increasing over the recent years [3, 10].

Fuzzing is a common technique that uses random inputs to test software. Although it has played a crucial role in automating software testing for several decades, traditional fuzzing faces many challenges, such as how to efficiently generate valid and effective test inputs [12]. Generative testing (also known as property-based testing) on the other hand is mostly paired with a test generator carefully designed based on the specification of the software. It is similar to a generation-based fuzzer and can often reach deeper program states more effectively [15]. Cosmos SDK [11], one of the most used crucial frameworks for building blockchains,[3] adopts the generative testing approach. Its built-in test generator generates a random combination of randomly parameterized operations for each block in a simulated blockchain and regularly checks if all invariants (properties) hold true.

In this project, we use the Cosmos SDK as our main testbed for developing a deep reinforcement learning (RL)-assisted generative testing framework. Although the SDK itself is highly complex with more than 1,500 source files and over 350,000 lines of code, and testing any large-scale software is fundamentally challenging, we demonstrate that our framework, which directly augments the random number generator (RNG) with an efficient deep RL agent, yields a statistically significant improvement to the current testing paradigm.

[1]`https://github.com/cytsai/cosmos-sdk/tree/helios-v1/simapp/rand`
[2]`https://github.com/cytsai/cosmos-sdk-gym`
[3]More than 100 billion USD in digital assets are managed by blockchains built with the Cosmos SDK [9].

Table 1: Summary of ML-assisted software testing approaches. Best viewed in color.

	Core[a]	State[b] (History[c])	Action[d]	Reward[e]	Automatic[f]
AFL [20]	GA	None	Test Input[g]	Coverage	Yes
NEUZZ [17]	NN	Coverage Bitmap		Approx. Coverage	Yes
GMetaExp [4]	DRL	Coverage Bitmap (S)		Coverage	Unknown
RLCheck [15]	RL	Call Stack (A)	Generator Input[h]	Trace Diversity	No
DeepRNG	DRL	Call Stack (SAR)		Coverage	Yes

[a]Core learning algorithm. [b]Or observation, algorithm's input extracted from the software. [c]Past information also used by the algorithm. [d]Or mutation/perturbation target, algorithm's output. [e]Or objective, optimization target of the algorithm. [f]If the software can be automatically instrumented (*e.g.* to output its states) for the test. [g]Input used directly to test the software. [h]Input (*e.g.* random numbers) used by the test generator to test the software.

1.1 Related Work

Test coverage is commonly used to measure the progress of testing software, since higher test coverage suggests lower chance of having undetected bugs. As such, coverage-guided fuzzing has become a highly popular technique and AFL [20] and its variations are arguably the most widely used coverage-guided fuzzers. They use genetic algorithms (GAs) to continuously mutate inputs that reached new states of the software under test, which empirically ensures good test coverage.

However, GA-based fuzzers can be very inefficient given that most GAs can only randomly explore the state space, ignoring any underlying structure that may guide the exploration. As such, NEUZZ [17] proposed to use a neural network (NN) to smoothly approximate the coverage bitmap (binary vector with each bit indicating whether a branch has been executed), whose gradients are then used to perturb the inputs to increase the approximated coverage.

For small-to-medium-sized software with source code available, GMetaExp [4] proposed to represent the program as a graph (*e.g.* abstract syntax tree) with observed coverage information labelled on each vertex. A graph neural network (GNN) and a long short-term memory network (LSTM) are used to encode the current and past states of the program for the deep RL (DRL) agent to predict actions that maximize the expected reward representing coverage.

Closest to our work, RLCheck [15] was the first to use RL for generative testing. It focused on the validity and diversity of the generated test cases using a classical RL algorithm (tabular Q-learning, which cannot handle large state spaces). It also adopted a manual approach to record recently taken branches (essentially truncated call stacks) as states of the software and recent actions of the RL agent, which require significant changes to the source code.

1.2 Our Contribution

We summarize the key features of the related work and our framework in Table 1,[4] and our main contributions as follows:

- We propose to use the (full) call stack of the software (see Fig. 1 and A.1 for examples) as the state representation which is effective and lightweight (compared to *e.g.* a coverage bitmap), and implement and open-source the library for automatic instrumentation.[1]
- We extend the state-history representations proposed by the previous work by using a deep RL architecture which can flexibly encode history of past states, actions and rewards without needing to truncate older history (as in *e.g.* RLCheck [15]).
- We show that the DeepRNG framework can achieve better coverage for the Cosmos SDK in a statistically significant way, as well as identifying previously unknown bugs.[5]

[4]Due to space constraints, we have only listed the three most representative and closely related ML-assisted fuzzers, AFL, NEUZZ and GMetaExp. Please see [16, 18] for more examples.

[5]`https://github.com/cosmos/cosmos-sdk/discussions/9178#discussioncomment-763961`

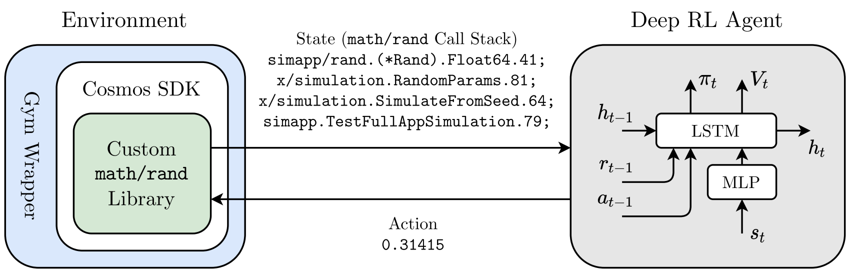

Environment
Gym Wrapper
Cosmos SDK
Custom math/rand Library
State (math/rand Call Stack)
simapp/rand.(*Rand).Float64.41;
x/simulation.RandomParams.81;
x/simulation.SimulateFromSeed.64;
simapp.TestFullAppSimulation.79;
Action
0.31415
Deep RL Agent
π_t
V_t
h_{t-1}
LSTM
h_t
r_{t-1}
a_{t-1}
MLP
s_t

Figure 1: The DeepRNG framework, where the colored modules (the custom RNG library and the OpenAI Gym wrapper) are the main software contributions of this project. The custom RNG library routes function calls to an external RL agent, with the call stack information passed as the states of the software under test, and the coverage information and exit codes (not shown in the figure) passed as the rewards. The RL agent then returns random numbers as actions that are sampled from a distribution learned to maximize the expected reward, conditioned on an encoding of the current states and optionally the past states, actions and rewards. The states, rewards and actions are passed as strings over `stdout`, `stdin` and/or named pipes, which are mainly handled by the OpenAI Gym wrapper to ensure the communication starts and ends correctly. In this example, the Cosmos SDK's `TestFullAppSimulation` routine (the "main function" for testing) entered the `SimulateFromSeed` subroutine, which called `RandomParams` that in turn requested a random `Float64` (between 0.0 and 1.0) from Go's RNG library, `math/rand`. The RL agent then returned `0.31415` as its action. Best viewed in color.

2 DeepRNG Framework

The proposed DeepRNG framework is illustrated in Fig. 1. The two new modules designed for this project, the custom RNG library implemented in Go[1] and its corresponding OpenAI Gym wrapper,[2] are both open-source and publicly available online. The custom RNG library can also reproduce simulations from log files without launching an RL agent, which is essential for reporting bugs. To achieve the goal of automatically extracting the call stack as the state of the software, we use the `runtime.CallersFrames` function in Go. Similar functions can be found in most programming languages [19], meaning that the DeepRNG framework is in theory extensible to most languages.

For the deep RL agent, we adopt the IMPALA algorithm [7] as well as its neural network architecture which flexibly supports encoding of the current state (s_t) and optionally the history of the past states, actions and rewards (h_{t-1}, a_{t-1}, r_{t-1}) of the software and the agent, using a multilayer perceptron (MLP) and a long short-term memory network (LSTM). In addition, the IMPALA agent is highly computationally efficient by design and provides the fastest training speed among all the deep RL agents we have tested.

3 Experiments

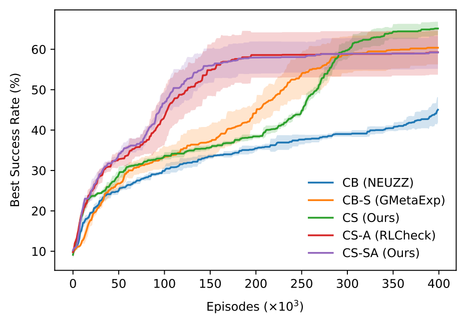
We use the IMPALA agent implemented in RLlib [13] for all of our experiments on a workstation with an Intel Core i7-10700 CPU, 64 GB of main memory and an Nvidia GeForce RTX 3090 GPU.[6] We focus on comparing different state-history representations to validate our approach and to showcase our framework's ability to better test real-world software (in this paper, the Cosmos SDK).

3.1 Binary Search Tree Generation

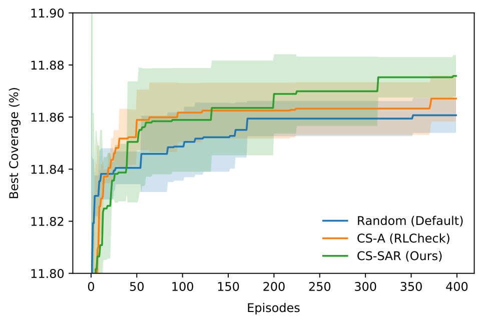
Inspired by the binary search tree (BST) test case generation problem in [15], we create a similar toy problem focusing on randomly generating valid BSTs,[7] where randomly numbered nodes are recursively generated until `TREE_DEPTH` and each node's left and right subtrees are independently pruned with a probability of `TREE_PRUNE`. We set `TREE_DEPTH` $= 4$, `TREE_PRUNE` $= 0.5$ and the

[6]We use 8 parallel environments and 8 CPU-based actors to collect trajectories for one GPU-based learner.

[7]A valid BST is a binary tree whose internal nodes each store a number greater than all the numbers in the node's left subtree and less than those in its right subtree.



(a) Success rate (*i.e.* average reward) of generating valid binary search trees (BSTs).



(b) Test coverage (*i.e.* reward) of the Cosmos SDK.

Figure 2: Experimental results. Both subfigures plot the best value achieved at a given episode since the beginning (also known as the maximum reward). For each curve (one state-history representation), state can be coverage bitmap (CB) or call stack (CS), and history can be any combination of past states (S), actions (A) and rewards (R). Best viewed in color.

Table 2: Cosmos SDK test coverage results.

	Best Coverage (%)	p-value
Random (Default)	11.861 ± 0.007	Baseline
CS	11.864 ± 0.009	0.584
CS-S	11.870 ± 0.011	0.139
CS-SR	11.883 ± 0.018	0.032
CS-A (RLCheck)	11.867 ± 0.009	0.234
CS-AR	11.869 ± 0.007	0.088
CS-SA	11.876 ± 0.010	0.024
CS-SAR	11.876 ± 0.008	0.012

agent's action space to integers between 1 and 31 such that the optimal solution (agent always able to generate valid BSTs) is achievable. Worth to note, even though the optimal solution is obvious to experienced programmers, the RL agent can only discover the solution using the very sparse binary rewards (0.0 for invalid BSTs, 1.0 for valid BSTs) that it receives at the end of each run (episode). More details about this problem and its optimal solution can be found in the appendix.

We train the IMPALA agent using different state-history representations for 0.4 million episodes (around 2 million environment steps) and 5 repeated runs per representation. The results are shown in Fig. 2(a). Here, the coverage bitmap (CB) representation consists of a 31-bit binary vector indicating the coverage status of all 31 nodes within the tree, and the call stack (CS) representation consists of a 31-bit one-hot vector indicating which one of the 31 nodes is currently being visited. Interestingly, although the CB representation carries more information (than the CS), it actually performs worse (or no better).[8] Moreover, among all CS-based representations, not using any history actually performs the best. This suggests that more information is not necessarily better (perhaps particularly so for RL problems where credit assignment is fundamentally challenging) and having a flexible framework based on efficient representations to facilitate the exploration of configurations is more critical.

3.2 Cosmos SDK

For the Cosmos SDK experiments, we train the IMPALA agent using different state-history representations for 400 episodes (around 10 million environment steps) and 5 repeated runs per representation. Because the coverage bitmap of the Cosmos SDK (like any other large-scale software) is too large (around 78,000-bits long) to be processed efficiently, we only use the call stack (CS) as the state representation, which is simply encoded as a 158-bit one-hot vector indicating which one of the 158 unique `math/rand` call stacks is currently being observed. The agent is tasked to generate actions

[8]We have also tested combining both representations, which yields results similar to using the CS alone.

(random numbers) within $[0, 1)$, which can be post-processed to meet different types of RNG requests (*e.g.* scaled and rounded for random integers within different ranges). During each episode, the agent is continuously rewarded with the increases in coverage[9] (which sum to the final coverage between 0.0 and 1.0) and a final reward of 1.0 if the simulation fails (*i.e.* a bug was encountered).

The results are shown in Fig. 2(b) and summarized in Table 2. Compared to the default random testing of the Cosmos SDK, using the call stack (CS) representation combined with the state, action and reward (SAR) history achieves the most statistically significant improvement based on the *t*-test (along with two other also statistically significant configurations as highlighted in Table 2). Incorporating the reward history could allow the agent to "know" how much of the software has (not) been covered and to adapt its strategy, which does seem to improve or robustify the coverage results. In addition to improving the coverage, the DeepRNG framework also has identified at least two types of bugs (invalid initialization and unexpected timeout) in the Cosmos SDK.[5]

4 Discussion

In this paper, we propose the DeepRNG framework and experimentally validate its effectiveness by showing promising results for testing the Cosmos SDK. Below are some of our future directions.

- For the Cosmos SDK, although the improvement in test coverage is statistically significant, the numerical increase is actually quite small. This is mainly due to the fact that the SDK has many modules not reachable by its current test generator. Filtering out unreachable (or unimportant) modules may simplify the problem and lead to better results.
- To further optimize the test coverage, it's also possible to completely disregard the "exploitation phase" of the RL algorithm, since exploring new states is the only source of rewards. The exploration phase of [5, 6] may be used to this end.

Acknowledgments

We thank the Interchain Foundation for funding this project, and Ethan Buchman and members of Informal Systems for valuable suggestions. Resources used in preparing this research were provided, in part, by the Province of Ontario, the Government of Canada through CIFAR, and companies sponsoring the Vector Institute.

References

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[2] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde, Jared Kaplan, Harri Edwards, Yura Burda, Nicholas Joseph, Greg Brockman, et al. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[3] MITRE Corporation. CVE (common vulnerabilities and exposures) details. `https://www.cvedetails.com/browse-by-date.php`, 2021.

[4] Hanjun Dai, Yujia Li, Chenglong Wang, Rishabh Singh, Po-Sen Huang, and Pushmeet Kohli. Learning transferable graph exploration. *Advances in Neural Information Processing Systems*, 32:2518–2529, 2019.

[5] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. Go-explore: a new approach for hard-exploration problems. *arXiv preprint arXiv:1901.10995*, 2019.

[6] Adrien Ecoffet, Joost Huizinga, Joel Lehman, Kenneth O Stanley, and Jeff Clune. First return, then explore. *Nature*, 590(7847):580–586, 2021.

[7] Lasse Espeholt, Hubert Soyer, Remi Munos, Karen Simonyan, Vlad Mnih, Tom Ward, Yotam Doron, Vlad Firoiu, Tim Harley, Iain Dunning, et al. IMPALA: Scalable distributed deep-RL with importance weighted actor-learner architectures. In *International Conference on Machine Learning*, pages 1407–1416. PMLR, 2018.

[8] GitHub. GitHub Copilot. `https://copilot.github.com`, 2021.

[9] Per-step reward is the difference between previous-step and current-step coverages. Coverage is measured using the `testing.Coverage` function in Go, which estimates the branch coverage [14].

[9] Tendermint Inc. Tendermint's proposal for a new versioning of Cosmos SDK. `https://medium.com/tendermint/tendermints-proposal-for-a-new-versioning-of-cosmos-sdk-d52a01976852`, 2021.

[10] Herb Krasner. The cost of poor software quality in the US: A 2020 report. In *Proc. Consortium Inf. Softw. QualityTM (CISQTM)*, 2021.

[11] Jae Kwon and Ethan Buchman. Cosmos whitepaper. `https://cosmos.network/resources/whitepaper`, 2019.

[12] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):1–13, 2018.

[13] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In *International Conference on Machine Learning*, pages 3053–3062. PMLR, 2018.

[14] Rob Pike. The Go blog: The cover story. `https://go.dev/blog/cover`, 2013.

[15] Sameer Reddy, Caroline Lemieux, Rohan Padhye, and Koushik Sen. Quickly generating diverse valid test inputs with reinforcement learning. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pages 1410–1421. IEEE, 2020.

[16] Gary J Saavedra, Kathryn N Rodhouse, Daniel M Dunlavy, and Philip W Kegelmeyer. A review of machine learning applications in fuzzing. *arXiv preprint arXiv:1906.11133*, 2019.

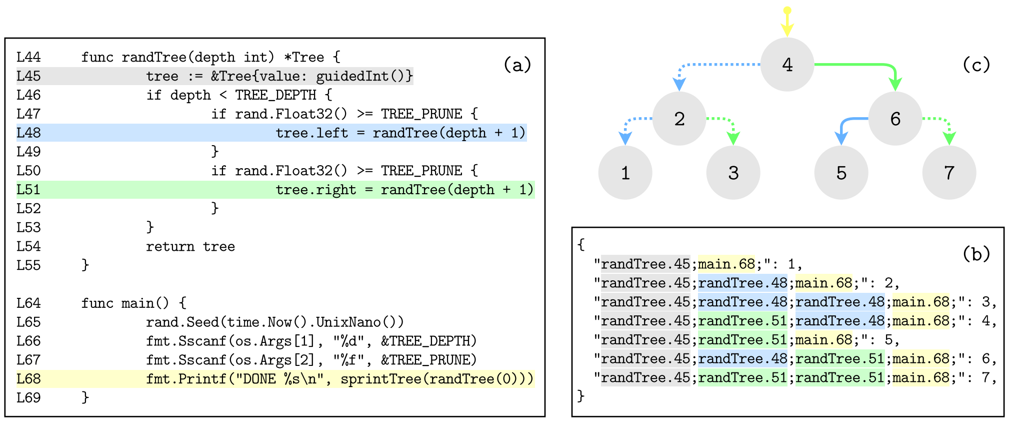
[17] Dongdong She, Kexin Pei, Dave Epstein, Junfeng Yang, Baishakhi Ray, and Suman Jana. NEUZZ: Efficient fuzzing with neural program smoothing. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 803–817. IEEE, 2019.

[18] Yan Wang, Peng Jia, Luping Liu, Cheng Huang, and Zhonglin Liu. A systematic review of fuzzing based on machine learning techniques. *PloS one*, 15(8):e0237749, 2020.

[19] Wikipedia. Stack trace. `https://en.wikipedia.org/wiki/Stack_trace`, 2021.

[20] Michał Zalewski. american fuzzy lop. `https://lcamtuf.coredump.cx/afl`, 2013.

A Appendix



```
L44     func randTree(depth int) *Tree {
L45             tree := &Tree{value: guidedInt()}
L46             if depth < TREE_DEPTH {
L47                     if rand.Float32() >= TREE_PRUNE {
L48                             tree.left = randTree(depth + 1)
L49                     }
L50                     if rand.Float32() >= TREE_PRUNE {
L51                             tree.right = randTree(depth + 1)
L52                     }
L53             }
L54             return tree
L55     }

L64     func main() {
L65             rand.Seed(time.Now().UnixNano())
L66             fmt.Sscanf(os.Args[1], "%d", &TREE_DEPTH)
L67             fmt.Sscanf(os.Args[2], "%f", &TREE_PRUNE)
L68             fmt.Printf("DONE %s\n", sprintTree(randTree(0)))
L69     }
```

```
{
  "randTree.45;main.68;": 1,
  "randTree.45;randTree.48;main.68;": 2,
  "randTree.45;randTree.48;randTree.48;main.68;": 3,
  "randTree.45;randTree.51;randTree.48;main.68;": 4,
  "randTree.45;randTree.51;main.68;": 5,
  "randTree.45;randTree.48;randTree.51;main.68;": 6,
  "randTree.45;randTree.51;randTree.51;main.68;": 7,
}
```

Figure A.1: Binary search tree (BST) generation. (a) Go program of a random BST generator where `randTree` is called recursively to generate randomly numbered nodes until `TREE_DEPTH` and each node's left and right subtrees are pruned with a probability of `TREE_PRUNE` independently. Following the DeepRNG framework, the random number for each node is generated using a custom `guidedInt` function, which passes its state (call stack) to an external RL agent each time being called. The agent is then trained to correspond with a random number that maximizes its final reward, the validity of the BST. (b) All 7 possible states (call stacks) for $\texttt{TREE\_DEPTH} = 2$. (c) The optimal solution of an agent with an action space of integers between 1 and 7. *I.e.* an agent that learns the state-action map $f : s \to a \ \forall (s, a) \in \{(1, 4), (2, 2), (3, 1), (4, 3), (5, 6), (6, 5), (7, 7)\}$ always gets a reward (a valid BST, like $4 \to (5 \leftarrow 6)$ in the figure) regardless of the tree shape and order of new nodes (due to random pruning it cannot control).