BanditFuzz: A Reinforcement-Learning based Performance Fuzzer for SMT Solvers

Joseph Scott[1], Federico Mora[2], and Vijay Ganesh[1]

[1] University of Waterloo, Ontario, Canada
{joseph.scott,vijay.ganesh}@uwaterloo.ca
[2] University of California, Berkeley
fmora@berkeley.edu

Abstract. In this paper, we present a reinforcement-learning based fuzzing system, BanditFuzz, that zeroes in on the grammatical constructs of well-formed inputs that are the root cause of performance slowdown in SMT solvers. BanditFuzz takes the following as input: a grammar G describing well-formed inputs to a set of distinct solvers (say, a target solver T and a reference solver R) that implement the same specification, and a fuzzing objective (e.g., maximize the relative performance difference between T and R). BanditFuzz outputs a list of grammatical constructs that are ranked in descending order by how likely they are to maximize performance differences between solvers T and R. Using BanditFuzz, we constructed two benchmark suites (with 400 floating-point and 300 string instances) that expose performance issues in all considered solvers, namely, Z3, CVC4, Colibri, MathSAT, Z3seq, and Z3str3. We also performed a comparison of BanditFuzz against random, mutation, and evolutionary fuzzing methods and observed up to a 81% improvement based on PAR-2 scores used in SAT competitions.

1 Introduction

Over the last two decades, many sophisticated program analysis [16], verification [18], and bug-finding tools [11] have been developed thanks to powerful Satisfiability Modulo Theories (SMT) solvers. The efficiency of SMT solvers significantly impacts the efficacy of modern program analysis and verification tools. Given the insatiable demand for efficient and robust SMT solvers, it is imperative that these infrastructural tools be subjected to extensive correctness and performance testing, analysis, and verification.

One such approach is (relative) performance fuzzing[3], which can be defined as follows: methods to automatically and efficiently generate inputs for a program-under-test T such that these inputs cause T to slowdown relative to a different program(s) R that implement the same specification.

Reinforcement Learning (RL) based Performance Fuzzing: Researchers have explored many methods for performance fuzzing of programs, including

[3] We use the terms "relative performance fuzzing" and "performance fuzzing" interchangeably in this paper.

blackbox random and mutation fuzzing. While blackbox approaches are cheap to build and deploy, they are unlikely to efficiently find inputs that expose performance issues. The reason is that purely blackbox approaches are oblivious to input/output behavior of programs-under-test, while methods for exposing performance issues often rely on understanding program behavior. A whitebox test generation approach (such as some variation of symbolic analysis) is indeed suitable for such a task but is inefficient for a different reason, namely, the path explosion problem.

By contrast, the paradigm of RL is particularly useful for this task, since RL methods are a powerful way of navigating a space of inputs, guided by the corrective feedback they receive via historical analysis of the input/output behavior of programs-under-test. In this paper, we introduce a reinforcement learning (RL) based fuzzer, called BanditFuzz, that improves on the traditional fuzzing approaches by up to a 81% improvement for *relative performance* fuzzing. The metric we use for comparing various fuzzing algorithms considered in this paper is the PAR-2 score margins used in SAT competitions [22]. Using BanditFuzz, we generated a database of 400 inputs that expose relative performance issues across a set of FP solvers, namely, CVC4 [4], MathSAT [13], Colibri [23], and Z3 [14], as well as 300 inputs exposing relative performance issues in the Z3seq (Z3's official string solver [14]), Z3str3 [5], and CVC4 string solvers [19].

Contributions: First RL-based Performance Fuzzer for Floating-Point and String SMT Solvers: We describe the design and implementation of the first RL-based fuzzer for SMT solvers, called BanditFuzz. BanditFuzz uses RL, specifically MABs, in order to construct fuzzing mutations over highly structured inputs with the aim of maximizing a fuzzing objective, namely, the relative performance difference between a target and a reference solver. To the best of our knowledge, using RL in this way has never been done before. Furthermore, as far as we know, BanditFuzz is the first RL-based fuzzer for SMT Solvers.

Extensive Empirical Evaluation of BanditFuzz: We provide an extensive empirical evaluation of our fuzzer for detecting relative performance issues in SMT solvers and compare it to existing techniques. That is, we use our fuzzer to find instances that expose the large performance differences in four state-of-the-art floating-point (FP) solvers, namely, Z3, CVC4, MathSat, and Colibri, as well as three string solvers, namely, Z3str3, Z3 sequence (Z3seq), and CVC4 solvers as measured by PAR-2 score [22]. BanditFuzz outperforms existing fuzzing algorithms (such as random, mutation, and genetic fuzzing) by up to an 81% increase in PAR-2 score margins, for the same amount of resources provided to all methods. We also contribute two large sets of inputs discovered by BanditFuzz that contain a combined total of 400 for the theory of FP and 300 for the theory of strings that the SMT community can use to test their solvers. We are not aware of any other such tool for the performance fuzzing of SMT solvers. Finally, we also communicated our results to the developers of Z3str3. They were surprised by the class of instances on which we were able to demonstrate a performance difference between Z3str3 and other solvers.

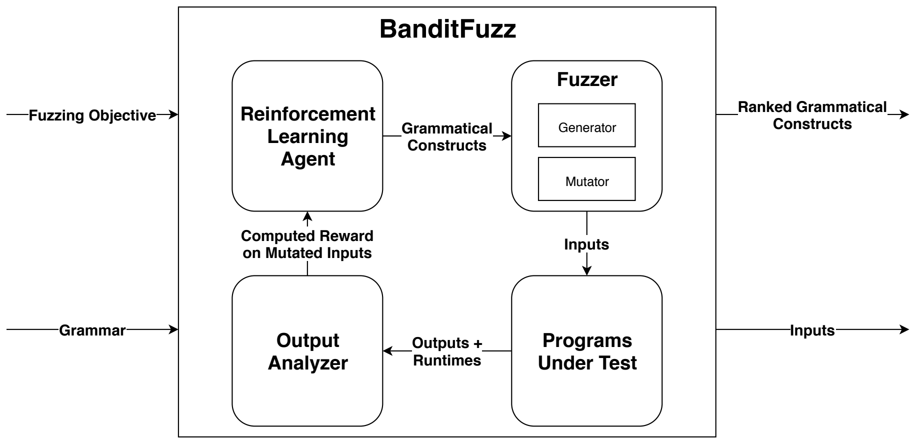

Fig. 1. Architecture of BanditFuzz

2 BanditFuzz: An RL-based Performance Fuzzer

In this section, we describe our technique, BanditFuzz, a grammar-based mutation fuzzer that uses reinforcement learning (RL) to efficiently isolate grammatical constructs of an input that are the cause of a performance issue in a solver-under-test. The ability of BanditFuzz to isolate those grammatical constructs that trigger performance issues, in a blackbox manner, is its most interesting feature. The architecture of BanditFuzz is presented in Figure 1.

2.1 Description of the BanditFuzz Algorithm

BanditFuzz takes as input a grammar G that describes well-formed inputs to a set P of solvers-under-test (for simplicity, assume P contains only two programs, a target program T to be fuzzed, and a reference program R against which the performance or correctness of T is compared), a fuzzing objective (e.g., relative performance issues) and outputs a ranked list of grammatical constructs (e.g., syntactic tokens or keywords over G) in the descending order of ones that are most likely to cause performance issues. We infer this ranked list by extrapolating from the policy of the RL agent. It is assumed that BanditFuzz has blackbox access to the set P of the solvers-under-test, and their input grammar is G.

The BanditFuzz algorithm works as follows: BanditFuzz generates well-formed inputs that adhere to G and mutates them in a grammar-preserving manner (the instance generator and mutator together are referred to as fuzzer in Figure 1) and deploys an RL agent (specifically a MAB agent) within a feedback loop to learn which grammatical constructs of G are the most likely culprits that may cause performance issues in the target program T in P.

BanditFuzz reduces the problem of how to mutate an input to an instance of the MAB problem. As discussed earlier, in the MAB setting an agent is designed to maximize its rewards by selecting the arms (actions) that give it the highest expected reward, while maintaining an exploration-exploitation tradeoff. In BanditFuzz, the agent chooses actions (grammatical constructs used by the fuzzer to mutate an input) that maximize the reward over a period of time (e.g., maximizing the runtime difference between the target solver T and a reference solver R). It is important to note that the agent learns an action selection policy by analyzing the results of its actions over time. Within its iterative feedback loop (that enables rewards from the analysis of solver outputs to the RL agent), BanditFuzz observes and analyzes the effects of the actions it takes on the solvers-under-test. BanditFuzz maintains a record of these effects over many iterations, analyzes the historical data collected, and zeroes-in on those grammatical constructs that maximize the expected reward. At the end of its run, BanditFuzz outputs a ranked list of grammatical constructs which are most likely to cause performance issues, in descending order. In the fuzzing for relative performance fuzzing mode, BanditFuzz performs the above-described analysis to produce a ranked list of grammatical constructs that maximize the difference in running time between a target solver T and a reference solver R.

2.2 Fuzzer: Instance Generator and Grammar-preserving Mutator

BanditFuzz's fuzzer (See Architecture of BanditFuzz in Figure 1) consists of two sub-components, namely, an instance[4] generator and a grammar-preserving mutator (or simply, mutator). The instance generator is a program that randomly samples the space of inputs described by the grammar G. The mutator is a program that takes as input a well-formed G-instance and a grammatical construct δ and outputs another well-formed G-instance.

Instance Generator: Here we describe the generator component of BanditFuzz, as described in Figure 1. Initially, BanditFuzz generates a random well-formed instance using the input grammar G (FP or string SMT-LIB grammar) via a random abstract syntax tree (AST) generation procedure built into StringFuzz [6]. We generalize this procedure for the theory of FP.

To further elaborate on our FP input generation procedure, we first populate a list of free 64-bit FP variables and then generate random ASTs that are asserted in the instance. Each AST is rooted by an FP predicate whose children are FP operators chosen at random. We deploy a recursive process to fill out the tree until a predetermined depth limit is reached. Leaf nodes of the AST are filled in by randomly selecting a free variable or special constant. Rounding modes are filled in when required by an operator's signature. The number of variables and assertions are parameters to the generator and are specified for each experiment.

Similar to the generator in StringFuzz, BanditFuzz's generation process is highly configurable. The user can choose the number of free variables, the number of assertions, the maximum depth of the AST, the set of operators, and rounding terms. The user can also set weights for specific constructs as a substitute for the default uniform random selection.

[4] We use the terms "instance" and "input" interchangeably through this paper.

Grammar-preserving Mutator: The second component of the BanditFuzz fuzzer is the mutator. In the context of fuzzing SMT solvers, a mutator takes a well-formed SMT formula I and a grammatical construct δ as input, and outputs a *mutated* well-formed SMT formula I' that is like I, but with a suitable construct (say, γ) replaced by δ. The construct γ in I could be selected using some user-defined policy or chosen uniform-at-random over all possible grammatical constructs in I. In order to be grammar-preserving, the mutator has to choose γ such that no typing and arity constraints are violated in the resultant formula I'. The grammatical construct δ, one of the inputs to the mutator, may be chosen at random or selected using an RL agent. We describe this process in greater detail in the next subsection.

On the selection of a grammatical construct, a grammatical construct of the same type (predicate, operator, or rounding mode, etc.) is selected uniformly at random. If the replacement involves an arity change, the rightmost subtrees will be dropped on a decrease in arity, or new subtrees will be generated on the increase in arity.

2.3 RL Agent and Reward-driven Feedback Loop in BanditFuzz

As shown in Figure 1, another important component of BanditFuzz is an RL agent (based on Thompson sampling) that receives rewards and outputs a ranked list of grammatical constructs (actions). The fuzzer maintains a policy and selects actions from it ("pulling an arm" in the MAB context), and appropriately modifies the current input I to generate a novel input I'. The rewards are computed by the Output Analyzer, which takes as input the outputs and runtimes produced by the solver-under-test S and computes scores and rewards appropriately. These are fed to the RL agent; the RL agent tracks the history of rewards it obtained for every grammatical construct and refines its ranking over several iterations of BanditFuzz's feedback loop. In the following subsections, we discuss it in detail.

Computing Rewards for Performance Fuzzing: We next describe BanditFuzz's reward computation for performance fuzzing in detail. BanditFuzz works as follows in the performance fuzzing mode. Initially, the fuzzer generates a well-formed input I (sampled uniformly-at-random). BanditFuzz then executes both the target solver T and reference solver R on I and records their respective runtimes (it is assumed that both solvers may produce the correct answer with respect to input I or timeout). BanditFuzz's OutputAnalyzer module then computes a score, PerfScore, defined as

$$\text{PerfScore}(I) := \text{runtime}(I, T) - \text{runtime}(I, R)$$

where the quantity $\text{runtime}(I, T)$ refers to the wall clock runtime of the target solver T on I, and $\text{runtime}(I, R)$ the runtime of the reference solver R on I. If the target solver reaches the wallclock timeout, we set $\text{runtime}(I, T)$ to be $2 \cdot \textit{timeout}$ — PAR-2 scoring in the SAT competition. In the same iteration, BanditFuzz mutates the input I to a well-formed input I' and computes the

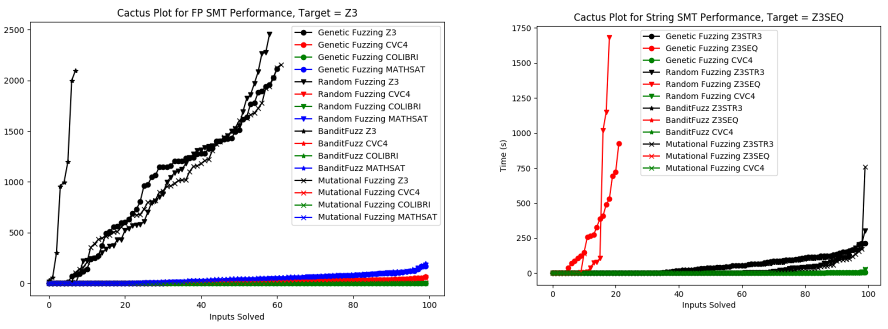

Fig. 2. Cactus Plot for targeting the Z3 FP solver (left) and Z3seq String solver (right) against reference solvers. As seen above, BanditFuzz maximizes the PAR-2 score of the target solver, compared to the other fuzzing algorithm within a given time budget.

quantity PerfScore(I'). Recall that we refer to the mutation inserted into I to obtain I' as γ.

The OutputAnalyzer then computes the rewards as follows. It takes as input I, I', quantities PerfScore(I), and PerfScore(I'), and if the quantity PerfScore(I') is better than PerfScore(I) (i.e., the target solver is slower than the reference solver on I' relative to their performance on I), the mutations γ gets a positive reward, else it gets a negative reward. Recall that we want to reward those constructs which make the target solver slower than the reference one.

The rewards thus computed are fed into the RL agent. The bandit then updates the rank of the grammatical constructs. The Thompson sampling bandit analyzes historically, the positive and negative rewards for each grammatical construct and computes the α and β parameters. The highest-ranked construct γ is fed into the fuzzer for the subsequent iteration. This process continues until the fuzzing resource limit has been reached.

3 Results: BanditFuzz vs. Standard Fuzzing Approaches

In this section, we present an evaluation of BanditFuzz vs. standard performance fuzzing algorithms, such as random, mutational, and evolutionary.

3.1 Experimental Setup

All experiments were performed on the SHARCNET computing service [3]: a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz. We limited each solver to 8GB of memory without parallelization. Otherwise, each solver is run under its default settings.

Baselines: We compare BanditFuzz with three different widely-deployed fuzzing loops that are built on top of StringFuzz [6]: random, mutation, and genetic fuzzing. We describe the three approaches below. We extend StringFuzz to

Target Solver	BanditFuzz	Random	Mutational	Genetic	% Improvement
Colibri	499061.5	499544.2	499442.2	499295.1	-0.10 %
CVC4	144568.9	68714.2	125273.0	38972.7	15.40 %
MathSAT5	36654.5	12024.9	31615.4	8208.0	15.94 %
Z3	467590.0	239774.3	256973.1	251108.2	81.96 %

Table 1. PAR-2 Score Margins of the returned inputs for considered fuzzing algorithms for FP SMT performance fuzzing. As seen in the table above, BanditFuzz maximizes the PAR-2 score of the target solver, compared to the other fuzzing algorithm within a given time budget.

floating-point, as described in Section 2.2. All baselines generate and modify inputs via StringFuzz's generator and transformer interface.
Random Fuzzing – Random fuzzers are programs that sample inputs from the grammar of the program-under-test (we only consider model-based random fuzzers here). Random fuzzing is a simple yet powerful approach to software fuzzing. We use StringFuzz as our random fuzzer for strings and extend a version of it to FP as described in Section 2.2.
Mutational Fuzzing – A mutation fuzzer typically mutates or modifies a database of input seeds in order to generate new inputs to test a program. Mutation fuzzing has had a tremendous impact, most notably in the context of model-less program domains [34, 12, 24, 20]. We use StringFuzz transformers as our mutational fuzzer with grammatical constructs selected uniformly at random. We lift StringFuzz transformer's to FP as described in Section 2.2.
Genetic/Evolutionary Fuzzing – Evolutionary fuzzing algorithms maintain a population of inputs. In every generation, only the *fittest members of the population* survive, and new members are created through random generation and mutation [28, 30].

We configure StringFuzz to generate random ASTs at random with five assertions. Each formula has one check-sat call. Each AST has depth three with five string/FP constants [5].

3.2 Quantitative Method for Comparing Fuzzing Algorithms

We run each of the baseline fuzzing algorithms and BanditFuzz on a target solver (e.g., Z3's FP procedure) and a set of reference solvers (e.g., CVC4, Colibri, MathSAT) for 12 hours with the aim of constructing an input that maximizes the difference between the runtime of the target solver and the reference solvers. We repeat this process for each fuzzing algorithm 100 times. We then take and compare the highest-scoring instance for each solver for each fuzzing algorithm.

The fuzzing algorithm that maximizes the runtime difference between the target solver and the reference solvers, in the given amount of time, is declared the

[5] Integer/Boolean constants are added for the theory of strings when appropriate (default behaviour of StringFuzz)

Target Solver	BanditFuzz	Random	Mutational	Genetic	Improvement
CVC4	45629.8	30815.4	30815.4	31619.4	44.15%
Z3str3	499988.6	499986.7	499987.2	499986.8	0.00%
Z3seq	499883.4	409111.0	433416.5	445097.427	12.31%

Table 2. PAR-2 Score Margins of the returned inputs for considered fuzzing algorithms for string SMT performance fuzzing. As seen in the table above, BanditFuzz maximizes the PAR-2 score of the target solver, compared to the other fuzzing algorithm within a given time budget.

best fuzzing algorithm among all the algorithms we compare. We show that BanditFuzz consistently outperforms random, mutation, and evolutionary fuzzing algorithms according to these criteria.

Quantitative Evaluation via PAR-2 Margins: For each solver/input pair, we record the wallclock time. To evaluate a solver over a set of inputs, we use *PAR-2* scores. PAR-2 is defined as the sum of all successful runtimes, with unsolved inputs labelled as twice the timeout. As we are fuzzing for performance with respect to a target solver, we evaluate the returned test suite of a fuzzing algorithm based on the *PAR-2 margin* between the PAR-2 of the target solver and the input wise maximum across all of the reference solvers. More precisely,

$$\text{PAR-2Margin}(S, s_t, D) := \sum_{I \in D} \text{PAR-2}(I, s_t) - \underset{s \in S, s \neq s_t}{\text{argmax}}(\text{PAR-2}(I, s))$$

for a set of solvers S and target solver $s_t \in S$, and generated input dataset D.

3.3 Performance Fuzzing Results for FP SMT Solvers

In our performance fuzzing evaluation of BanditFuzz, we consider the following state-of-the-art FP SMT solvers: **Z3** v4.8.0 - a multi-theory open source SMT solver [14], **MathSAT5** v5.5.3. a multi theory SMT solver [13], **CVC4** 1.7-prerelease [git master 61095232] - a multi theory open source SMT Solver [4], and **Colibri** v2070 - A proprietary SMT Solver with specialty in FP [7, 23].

Table 1 presents the margins of the PAR-2 scores between the target solver and the maximum of the reference solvers across the returned inputs for each fuzzing algorithm. BanditFuzz shows a notable improvement on fuzzing baselines except for when Colibri is selected as the target solver. In the case of Colibri being the target solver, all baselines observe PAR-2 margins near the maximum value of 500,000, leaving no room for BanditFuzz to improve. Having such a high margin indicates each run of a fuzzer resulted in an input where Colibri timed out, yet all other considered solvers solved it almost immediately.

Figure 2 (left) presented the cactus plot for the experiments when Z3 was the target solver. Also, we can obtain a ranking of grammatical constructs by extrapolating the α, β values from the learned model and sampling its beta distribution to approximate the expected value of reward for the grammatical construct's corresponding action. The top three for each target solver are: Colibri – fp.neg, fp.abs, fp.isNegative, CVC4 – fp.sqrt, fp.gt, fp.geq, MathSAT5 –

fp.isNaN, RNE, fp.mul, Z3 – fp.roundToIntegral, fp.div, fp.isNormal. This indicates that, e.g., CVC4's reasoning on fp.sqrt could be improved by studying Z3's implementation.

3.4 Performance Fuzzing for String SMT Solvers

In our performance fuzzing evaluation of BanditFuzz, we consider the following state-of-the-art string SMT solvers: **Z3str3** v4.8.0 [5], **Z3seq** v4.8.0 [14], and **CVC4 v1.6** [4]. We fuzz the string solvers for relative performance issues, with each considered as a target solver. Identically to the above FP experiments, each run of a fuzzer is repeated 100 times to generate 100 different inputs.

Table 2 presents the margins of the PAR-2 scores between the target solver and the maximum of the remaining solvers across the returned inputs for each fuzzing algorithm. BanditFuzz shows a substantial improvement on fuzzing baselines except for when Z3str3 is selected as the target solver. However, in this scenario, the PAR-2 margins are near the maximum value of 500000, across all fuzzing algorithms. This implies a nearly maximum input resulting in Z3str3 timing out while CVC4 and Z3seq solve the input nearly instantly.

As in the previous Section 3.3, we can extrapolate the grammatical constructs that were most likely to cause a performance slowdown. The top three for each target solver are as follows: CVC4 – re.range, str.contains, str.to_int, Z3seq – re.in_regex, str.prefixOf, str.length, Z3str3 – str.contains, str.suffixOf, str.concat. Further, Figure 2 (right) presents the cactus plot for the experiments when Z3seq was the target solver. The cactus plot provides a visualization of the fuzzing objective, maximizing the performance margins between Z3seq and the other solvers collectively. The line for BanditFuzz for the Z3seq solver is not rendered on the plot as the inputs returned by BanditFuzz were too hard for Z3seq and were not solved in the given timeout.

4 Related Work

Fuzzers for SMT Solvers: We refer to Takanen et al. [32] and Sutton et al. [31] for a detailed overview of fuzzing. While there are many fuzzers for finding bugs in specific SMT theories [6, 10, 9, 21], BanditFuzz is the first performance fuzzer for SMT solvers that we are aware of.

Machine Learning for Fuzzing: Bottinger et al. [8] introduce a deep Q learning algorithm for fuzzing model-free inputs, further PerfFuzz by Lemieux et al., uses bitwise mutation for performance fuzzing. These approaches would not scale to either FP SMT nor string SMT theories, given the complexity of their grammars. Such a tool would need to first learn the grammar to penetrate the parsers to begin to discover performance issues. To this end, Godefroid et al. [15] use neural networks to learn an input grammar over complicated domains such as PDF and then use the learned grammar for model-guided fuzzing. To the best of our knowledge, BanditFuzz is the first fuzzer to use RL to implement model-based mutation operators that can be used to isolate the root causes of performance issues in the programs-under-test.

While bandit algorithms (RL algorithms to solve the multi-arm bandit problem) have been used in various aspects as fuzzing, but never to implement a mutation. Karamcheti et al. [17] trained bandit algorithms to select model-less bitwise mutation operators from an array of fixed operators for greybox fuzzing. Woo et al. [33] and Patil et al. [27] used bandit algorithms to select configurations of global hyper-parameters of fuzzing software. Rebert et al. [29] used bandit algorithms to select from a list of valid inputs seeds to apply a model-less mutation procedure on. Our work differs from these methods, as we learn a model-based mutation operator implemented by an RL agent. Appelt et al. [1] combine blackbox testing with machine learning to direct fuzzing. To the best of our knowledge, our work is the first to use reinforcement learning or bandit algorithms to learn and implement a mutation operator within a grammar-based mutational fuzzing algorithm.

Delta Debugging: BanditFuzz differs significantly from delta debugging, where a bug-revealing input E is given, and the task of a delta-debugger is to minimize E to get E' while ensuring that E' exposes the same error in the program-under-test as E [26, 25, 2, 35]. BanditFuzz, on the other hand, generates and examines a set of inputs that expose performance issues in a target program by leveraging reinforcement learning. The goal of BanditFuzz is to discover patterns over the entire generated set of inputs via a historical analysis of the behavior of the program via RL. Specifically, BanditFuzz finds and ranks the language features that are the root cause of performance issues in the program-under-test.

5 Conclusions and Future Work

In this paper, we presented BanditFuzz, a performance fuzzer for FP and string SMT solvers that automatically isolates and ranks those grammatical constructs in an input that are the most likely cause of an error or relative performance issue. BanditFuzz is the first fuzzer for FP SMT solvers that we are aware of, and the first fuzzer to use reinforcement learning, specifically MAB, to fuzz SMT solvers. We compare BanditFuzz against a portfolio of practical baselines, including random, mutational, and evolutionary fuzzing techniques, and found that it consistently outperforms existing approaches. Future work of BanditFuzz is the extension to other theories and fuzzing objectives, such as bug fuzzing. Furthermore, RL performance fuzzers can potentially have a high impact in other domains, such as constructing inputs that cause a denial of service attacks on smart contracts.

References

1. Appelt, D., Nguyen, C.D., Panichella, A., Briand, L.C.: A machine-learning-driven evolutionary approach for testing web application firewalls. IEEE Transactions on Reliability **67**(3), 733–757 (2018)

2. Artho, C.: Iterative delta debugging. International Journal on Software Tools for Technology Transfer **13**(3), 223–246 (2011)
3. Baldwin, S.: Compute canada: advancing computational research. In: Journal of Physics: Conference Series. vol. 341, p. 012001. IOP Publishing (2012)
4. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanovi'c, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11). Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (Jul 2011), http://www.cs.stanford.edu/ barrett/pubs/BCD+11.pdf, snowbird, Utah
5. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59. IEEE (2017)
6. Blotsky, D., Mora, F., Berzish, M., Zheng, Y., Kabir, I., Ganesh, V.: Stringfuzz: A fuzzer for string solvers. In: International Conference on Computer Aided Verification. pp. 45–51. Springer (2018)
7. Bobot-CEA, F., Chihani-CEA, Z., Iguernlala-OCamlPro, M., Marre-CEA, B.: Fpa solver
8. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. arXiv preprint arXiv:1801.04589 (2018)
9. Brummayer, R., Biere, A.: Fuzzing and delta-debugging smt solvers. In: Proceedings of the 7th International Workshop on Satisfiability Modulo Theories. pp. 1–5. ACM (2009)
10. Bugariu, A., Müller, P.: Automatically testing string solvers. In: International Conference on Software Engineering (ICSE), 2020. ETH Zurich (2020)
11. Cadar, C., Ganesh, V., Pawlowski, P.M., Dill, D.L., Engler, D.R.: Exe: automatically generating inputs of death. ACM Transactions on Information and System Security (TISSEC) **12**(2), 10 (2008)
12. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. pp. 725–741. IEEE (2015)
13. Cimatti, A., Griggio, A., Schaafsma, B.J., Sebastiani, R.: The mathsat5 smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 93–107. Springer (2013)
14. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008)
15. 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. pp. 50–59. IEEE Press (2017)
16. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. ACM SIGPLAN Notices **43**(6), 281–292 (2008)
17. Karamcheti, S., Mann, G., Rosenberg, D.: Adaptive grey-box fuzz-testing with thompson sampling. In: Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security. pp. 37–47. ACM (2018)
18. Le Goues, C., Leino, K.R.M., Moskal, M.: The boogie verification debugger (tool paper). In: International Conference on Software Engineering and Formal Methods. pp. 407–414. Springer (2011)
19. Liang, T., Reynolds, A., Tsiskaridze, N., Tinelli, C., Barrett, C., Deters, M.: An efficient smt solver for string constraints. Formal Methods in System Design **48**(3), 206–234 (2016)
20. Manes, V.J., Han, H., Han, C., Cha, S.K., Egele, M., Schwartz, E.J., Woo, M.: Fuzzing: Art, science, and engineering. arXiv preprint arXiv:1812.00140 (2018)

21. Mansur, M.N., Christakis, M., Wüstholz, V., Zhang, F.: Detecting critical bugs in smt solvers using blackbox mutational fuzzing. arXiv preprint arXiv:2004.05934 (2020)
22. Marijn Heule, Matti Järvisalo, M.S.: Sat race 2019 (2019), http://sat-race-2019.ciirc.cvut.cz/
23. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. In: 15th International Workshop on Satisfiability Modulo Theories (2017)
24. Miller, C., Peterson, Z.N., et al.: Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep (2007)
25. Misherghi, G., Su, Z.: Hdd: hierarchical delta debugging. In: Proceedings of the 28th international conference on Software engineering. pp. 142–151. ACM (2006)
26. Niemetz, A., Biere, A.: ddSMT: A Delta Debugger for the SMT-LIB v2 Format. In: Proceedings of the 11th International Workshop on Satisfiability Modulo Theories, SMT 2013), affiliated with the 16th International Conference on Theory and Applications of Satisfiability Testing, SAT 2013, Helsinki, Finland, July 8-9, 2013. pp. 36–45 (2013)
27. Patil, K., Kanade, A.: Greybox fuzzing as a contextual bandits problem. arXiv preprint arXiv:1806.03806 (2018)
28. 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)
29. Rebert, A., Cha, S.K., Avgerinos, T., Foote, J., Warren, D., Grieco, G., Brumley, D.: Optimizing seed selection for fuzzing. In: USENIX Security Symposium. pp. 861–875 (2014)
30. Seagle Jr, R.L.: A framework for file format fuzzing with genetic algorithms (2012)
31. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
32. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
33. Woo, M., Cha, S.K., Gottlieb, S., Brumley, D.: Scheduling black-box mutational fuzzing. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 511–522. ACM (2013)
34. Zalewski, M.: American fuzzy lop (2015)
35. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (Feb 2002)