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@cs.berkeley.edu

Abstract. Satisfiability Modulo Theories (SMT) solvers are fundamental tools that are used widely in software engineering, verification, and security research. Precisely because of their widespread use, it is imperative we develop efficient and systematic methods to test them. To this end, we present a reinforcement-learning based fuzzing system, BanditFuzz, that learns grammatical constructs of well-formed inputs that may cause performance slowdown in SMT solvers. To the best of our knowledge, BanditFuzz is the first machine-learning based performance fuzzer for 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., aim to 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 increase the performance difference 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. That is, relative to other fuzzing methods considered, BanditFuzz was found to be more efficient at constructing inputs with wider performance margin between a target and a set of reference solvers.

1 Introduction

Over the last two decades, many sophisticated program analysis [20], verification [24], and bug-finding tools [13] have been developed thanks to powerful Satisfiability Modulo Theories (SMT) solvers. The efficiency of SMT solvers significantly impacts the efficacy of modern program analysis, testing, 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 and verification.

While there is considerable work on test generation and verification techniques aimed at correctness of SMT solvers [11, 7], we are not aware of previous work aimed at automatically generating inputs that expose performance issues in these complex and sophisticated tools.

One such approach is (relative) performance fuzzing[3], which can be defined as follows: methods aimed at automatically and efficiently generating inputs for a program-under-test T such that the performance margin between the program T and a set of other program(s) R that implement the same specification is maximized.

Reinforcement Learning (RL) based Performance Fuzzing: Researchers have explored many methods for performance fuzzing of programs, including blackbox random and mutation fuzzing [25]. 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. A whitebox test generation approach (such as some variation of symbolic analysis) is indeed suitable for such a task, but they tend to be inefficient for a different reason, namely, the path explosion problem. In particular, for complex systems like SMT solvers, purely whitebox performance fuzzing approaches are unlikely to scale.

By contrast, the paradigm of RL is well suited for this task of performance fuzzing, since RL methods are an efficient way of navigating a search space (e.g., a space of inputs to programs), guided by corrective feedback they receive via historical analysis of input/output (I/O) behavior, of programs-under-test. Further, they can be low-cost since they interact with programs-under-test in a blackbox fashion.

In this paper, we introduce an RL-based fuzzer, called BanditFuzz, that improves on traditional fuzzing approaches by up to a 81% improvement for *relative performance* fuzzing. That is, relative to other fuzzing methods considered in this paper, BanditFuzz is more efficient at constructing inputs with wider performance margins between a target and a set of reference solvers.

The metric we use for comparing various fuzzing algorithms considered in this paper is the PAR-2 score margins used in SAT competitions [29]. Using BanditFuzz, we generated a database of 400 inputs that expose relative performance issues across a set of FP solvers, namely, CVC4 [4], MathSAT [16], Colibri [30], and Z3 [18], as well as 300 inputs exposing relative performance issues in the Z3seq (Z3's official string solver [18]), Z3str3 [6], and CVC4 string solvers [26].

Description of BanditFuzz: BanditFuzz takes as input a grammar G that describes well-formed inputs to a set P of programs-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 of T is compared), a fuzzing objective (e.g., aim to maximize the relative performance margin between a target and a set of reference solvers). BanditFuzz outputs a ranked list of *grammatical constructs* (e.g., syntactic tokens, expressions, keywords, or combinations

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

thereof, over the input language described by G) in the descending order of ones that are most likely to trigger a performance issue, as well as actual instances that expose these issues in the programs-under-test. (It is assumed that BanditFuzz has blackbox access to programs in the set P and that all programs in the set P have the same input grammar G.)

Briefly, BanditFuzz works as follows: BanditFuzz generates well-formed inputs that adhere to the input grammar G, mutates them in a grammar-preserving manner, and uses RL methods to perform a historical analysis of the I/O behavior of the programs in P, in order to learn which grammatical constructs are most likely to cause performance issues in the programs in P. By contrast, traditional mutation fuzzers choose or implement a *mutation operator* at random and are oblivious to the behavior of the programs-under-test.

BanditFuzz reduces the problem of how to optimally mutate an input to an instance of the multi-arm bandit (MAB) problem, well-known in the RL literature [44, 46]. The crucial insight behind BanditFuzz is the idea of automatically analyzing the history of a target solver's performance, and using this analysis to create a list of grammatical constructs in G, and ranking them based on how likely they are to be a cause of a performance issue in the solver-under-test. Initially, all grammatical constructs in G are treated as uniformly likely to cause a performance issue by BanditFuzz's RL agent. BanditFuzz then randomly generates a well-formed input I, to begin with, and runs all the programs in P on the input I. In each of the subsequent iterations of its feedback loop, BanditFuzz mutates the input I from its previous iteration using the ranked list of grammatical constructs (i.e., the agent performs an action) and runs all solvers in P on the mutated version of the input I. It analyzes the results of these runs to provide feedback (i.e., rewards) to the RL agent in the form of those constructs that are most likely to cause relative performance difference between the target program T with respect to the reference program R. It then updates and re-ranks its list of grammatical constructs with the goal of maximizing its reward (i.e., increasing the relative performance difference between the target and reference solvers in P). The process continues until the RL agent converges to a ranking or runs out of resources.

Key Features of BanditFuzz: A key feature of BanditFuzz that sets it apart from other fuzzing and systematic testing approaches is that, in addition to generating inputs that reveal performance issues, it isolates or localizes a cause of performance issue, in the form of a ranked list of grammatical tokens that are the most likely cause of a performance issue in the target solver-under-test. This form of localization is particularly useful in understanding problematic behaviours in complex programs such as SMT solvers.

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 per-

formance 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 performance 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 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 [29]). 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 benchmark suites 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.

2 Preliminaries

Reinforcement Learning: There is a vast literature on reinforcement learning, and we refer the reader to the following excellent surveys and books on the topic [46, 44, 45]. As discussed in the introduction, the reinforcement learning paradigm is particularly suited for modelling mutation fuzzing, whenever an online corrective feedback loop makes sense in the fuzzing context. In this paper, we specifically deploy multi-armed bandit (MAB) algorithms [44], a class of reinforcement learning algorithms, to learn mutation operators (functions that perform a syntactic modification on well-formed inputs in a grammar-preserving fashion).

Reinforcement learning algorithms are commonly formulated using *Markov Decision Processes* (MDPs) [39, 46, 42], a 4-tuple of states S, actions A, rewards R, and transitions T. The multi-armed bandit (MAB) problem is a common reinforcement learning problem based on a stateless MDP (or more precisely, a single state $S = \{s_0\}$) and a finite set of actions A. Due to the nature of the problem, there is no learned modelling of transitions T. What remains to be learned, is the unknown probability distribution of rewards R over the space of actions A. In the context of MAB, actions are often referred to as arms (or bandits [4]).

In this paper, we exclusively consider the case where rewards are sampled from an unknown Bernoulli distribution (rewards are $\{0, 1\}$). The MAB agent attempts to approximate the expected value of the Bernoulli distribution of reward for each action in A. The MAB learns a *policy* – a stochastic process of how

[4] The term bandit comes from gambling: the arm of a slot machine is referred to as a one-armed bandit, and multi-arm bandits referred to several slot machines. The goal of the MAB agent is to maximize its reward by playing a sequence of actions (e.g., slot machines).

to select actions from A. The learned policy balances the exploration/exploitation trade-off, i.e., a MAB algorithm selects every action an infinite number of times in the limit. Still, it selects the action(s) with the highest expected reward more frequently.

While we implemented three solutions to the MAB problem into BanditFuzz, we focus on only one in this paper, namely, *Thompson Sampling*. Thompson Sampling builds a Beta distribution for each action in the action space. Beta distributions are a variant of Gamma distributions and have a long history. We refer the reader to Gupta et al. on Beta and Gamma distributions [21]. Intuitively, a Beta distribution is a continuous approximation of an underlying Bernoulli distribution approaching the same mean (p parameter) in the limit. It is maintained by updating the parameters $\alpha - 1$ (the samples of 1) and $\beta - 1$ (the samples of 0) from the underlying Bernoulli distribution.

In Thompson sampling, the agent maintains a Beta distribution for each action. The agent samples each action's distribution, and greedily picks its arm based on the maximum sampled value. Upon completing the action, α is incremented on a reward. Otherwise, β is incremented. For more on Thompson sampling, we refer to Russo et al. [40].

Satisfiability Modulo Theories and the SMT-LIB Standard: Satisfiability Modulo Theories (SMT) solvers are decision procedures for first-order theories such as integers, bit-vectors, floating-point, and strings that are particularly suitable for verification, program analysis, and testing [5]. The SMT-LIB is an initiative to standardize the language and specification of several theories of interest. In this paper, we exclusively consider solvers, whose quantifier-free FP and string decision procedures are being actively developed at the time of writing of this paper.

Quantifier-free Theory of Floating Point Arithmetic (FP): The SMT theory of FP was first proposed by Rümmer et al. [38] with several recent revisions. In this paper, we consider the latest version, by Brain et al. [10]. The SMT-LIB FP theory supports standard FP sorts of 32, 64, and 128 bit lengths with their usual mantissa and exponent bit vector lengths, and also allows for arbitrary width sorts with appropriate mantissa and exponent lengths. The theory includes common predicates, operators, and terms over FP. We refer the reader to the SMT-LIB standard for details on the syntax and semantics of FP theory. In this paper, we consider the following set of operators: { fp.abs, fp.neg, fp.add, fp.mul, fp.sub, fp.div, fp.fma, fp.rem, fp.sqrt, fp.roundToIntegral }, set of predicates: { fp.eq, fp.lt, fp.gt, fp.leq, fp.geq, fp.isNormal, fp.isSubnormal, fp.isZero, fp.isInfinite, fp.isNaN, fp.isPositive, fp.isNegative }, and rounding terms { RNE, RNA, RTP, RTN, RTZ }. Semantics of all operands follow the IEEE754 08 standard [17].

Quantifier-free Theory of Strings: The SMT-LIB standard for the theory of strings is currently in development [14]. The draft has a finite alphabet Σ of characters, string constants and variables that range over Σ^*, integer constants and variables, as well as the functions { str.++, str.contains, str.at, str.len, str.indexof, str.replace, re.inter, re.range, re.+, re.*, re.++, str.to_re }, and pred-

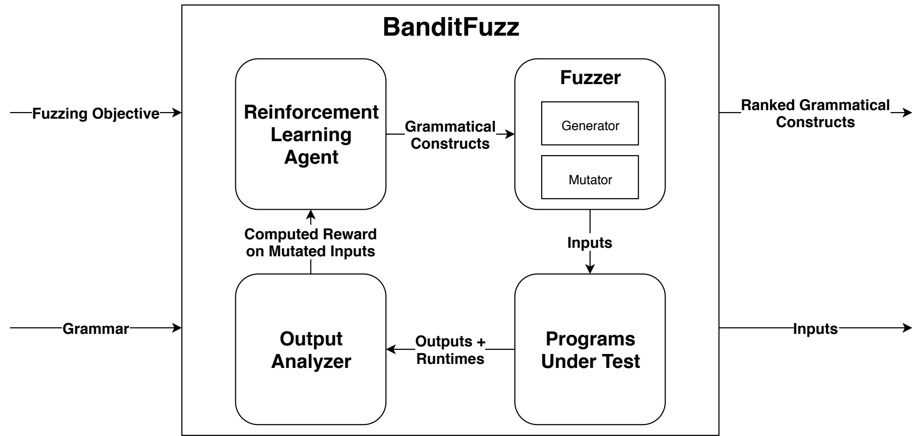

Fig. 1. Architecture of BanditFuzz

icates { str.prefixof, str.suffixof, str.in_re}. We further clarify that this list was carefully selected to include only those that are supported amongst all solvers considered in this paper.

Software Fuzzing: A Fuzzer is a program that automatically generates inputs for a target program-under-test. Fuzzers may treat the program-under-test as a whitebox or blackbox, depending on whether they have access to the source code. Unlike random fuzzers, a mutation fuzzer takes as input a database of inputs of interest and produces new inputs by mutating the elements of the database using a mutation operator (a function defining a syntactic change). These mutation operators are frequently stochastic bit-wise manipulations in the case of model-less programs or grammar-preserving changes for model-based programs [49, 15, 31, 27]. Other common fuzzing approaches include genetic and evolutionary fuzzing solutions. These approaches maintain a population of input seeds that are mutated or combined/crossed-over using a genetic or evolutionary algorithm [36, 41, 23].

3 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.

3.1 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., aim to maximize the relative performance difference between target and reference solvers) 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.

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 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 cumulative 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., increasing 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 via a historical analysis of 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 thus collected, and zeroes-in on those grammatical constructs that have the highest likelihood of 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 increase the difference in running time between a target solver T and a reference solver R.

3.2 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[5] 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.

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

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 [7]. We generalize this procedure for the theory of FP.

The FP input generation procedure works as follows: 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.

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, an arbitrary 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 are dropped on a decrease in arity, or new subtrees are generated on the increase in arity.

For illustrative purposes, we provide an example mutation here. Consider a maximum depth of two, fixed set of free FP variables (x_0, x_1), limited rounding mode set of $\{RNE\}$, and an asserted equation:

$$(fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.sub\ RNE\ x_0\ x_1)).$$

If the agent elects to insert $fp.abs$ there are two possible results:

$$(fp.eq\ (fp.abs\ x_0)(fp.sub\ RNE\ x_0\ x_1)), \quad (fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.abs\ x_0)).$$

For further analysis, consider the additional asserted equation:

$$(fp.eq\ (fp.abs\ x_0)(fp.abs\ x_1)),$$

Algorithm 1 BanditFuzz's Performance Fuzzing Feedback Loop. Also refer to BanditFuzz architecture in Figure 1.

```
 1: procedure BanditFuzz(G)
 2:     Instance I ← a randomly-generated instance over G                ▷ Fuzzer
 3:     Run target solver T and reference solver(s) R on I
 4:     Compute PerfScore(I)                                     ▷ OutputAnalyzer
 5:     θ = 2· Solver timeout
 6:     while fuzzing time limit not reached and PerfScore(I) < θ do
 7:         construct ← RL AGENT picks a grammatical construct        ▷ RL Agent
 8:         I′ ← Mutate I with construct                               ▷ Fuzzer
 9:         Run target solver T and reference solver(s) R on I′
10:         if PerfScore(I′, P) > PerfScore(I, P) then            ▷ OutputAnalyzer
11:             Provide reward to RL AGENT for construct
12:             I ← I′
13:         else
14:             Provide no reward to AGENT for construct
15:         end if
16:     end while
17:     return I and the ranking of constructs from RL AGENT
18: end procedure
```

if the agent elects to insert $fp.add$, then there are four[6] possible outputs:

$$(fp.eq\ (fp.add\ RNE\ x_0\ x_0)(fp.abs\ x_1))$$
$$(fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.abs\ x_1))$$
$$(fp.eq\ (fp.abs\ x_0)(fp.add\ RNE\ x_1\ x_0))$$
$$(fp.eq\ (fp.abs\ x_0)(fp.add\ RNE\ x_1\ x_1))$$

In these examples, the reason why the possible outputs may seem limited is due to type and arity preservation rules described above. As described below, the fuzzer would select one of the mutations in the above example in a manner that maximizes expected reward (e.g., the fuzzing objective such that the performance difference between a solver-under-test and a reference solver is increases).

3.3 RL Agent and Reward-driven Feedback Loop in BanditFuzz

As shown in Figure 1, the key 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

[6] This is assuming only the RNE rounding mode is allowed, otherwise each of the below expressions could have any valid rounding mode resulting in 20 possible outputs.

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 (see Algorithm 1). In the following subsections, we discuss it in detail.

Computing Rewards for Performance Fuzzing: We describe BanditFuzz's reward computation for performance fuzzing in detail here, and display the pseudo-code for it in Algorithm 1 (see also the architecture in Figure 1 to get a higher-level view of the algorithm). 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 quantity $\text{PerfScore}(I')$. 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 $\text{PerfScore}(I)$, and $\text{PerfScore}(I')$, and if the quantity $\text{PerfScore}(I')$ is better than $\text{PerfScore}(I)$ (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 reward for all other grammatical constructs remains unchanged.

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.

4 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.

4.1 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

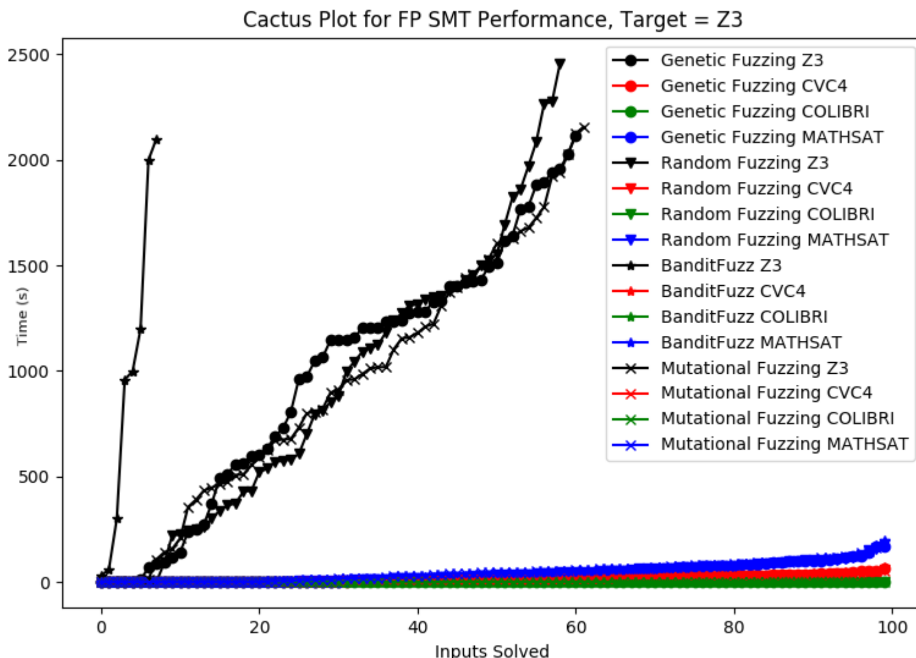

Fig. 2. Cactus Plot for targeting the Z3 FP Solver against reference solvers CVC4, Colibri, and MathSAT. As seen above, BanditFuzz has larger performance margins against the target solver (Z3), compared to the other fuzzing algorithm within a given time budget.

limited each solver to 8GB of memory without parallelization. Otherwise, each solver is run under its default settings. Each solver/input query is ran with a wallclock timeout of 2500 seconds.

Baselines: We compare BanditFuzz with three different widely-deployed fuzzing loops that are built on top of StringFuzz [7]: random, mutation, and genetic fuzzing. We describe the three approaches below. We extend StringFuzz to floating-point, as described in Section 3.2. 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 3.2.

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 [49, 15, 31, 27]. We use StringFuzz transformers as

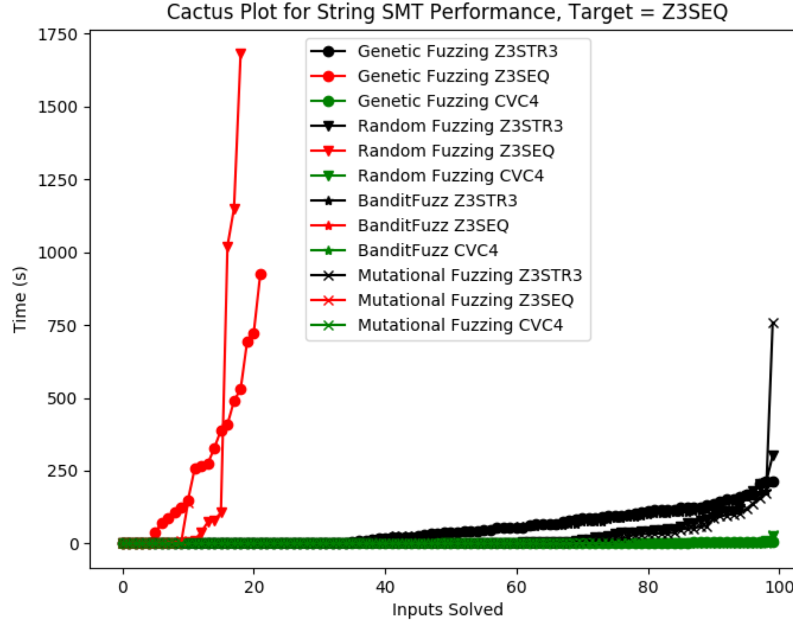

Fig. 3. Cactus Plot for targeting the Z3seq string solver against reference solvers CVC4 and Z3str3. As seen above, BanditFuzz has larger performance margins against the target solver (Z3), compared to the other fuzzing algorithm within a given time budget.

our mutational fuzzer with grammatical constructs selected uniformly at random. We lift StringFuzz transformer's to FP as described in Section 3.2.
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 [36, 41].

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 [7].

4.2 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 to construct a single input with maximal difference

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

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.

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 has the largest runtime separation between the target solver and the reference solvers, in the given amount of time, is declared the 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,

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

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

For example, consider a target solver S_1 against a set of reference solvers S_2, S_3, over a benchmark suite of three inputs. Let the runtimes for the solver S_1 on the three inputs be $1000.0, \textit{timeout}, 100.0$, that of solver S_2 be $50.0, 30.0, 10.0$, and that of solver S_3 be $100.0, 1000.0, 1.0$, respectively. With our timeout of 2500 seconds, S_1 would have a PAR-2 of 6100, S_2 a score of 90, and S_3 a score of 1101. We define the PAR-2 margin by summing the difference between the maximum of S_2, S_3 from that of solver S_1 on each of the inputs, which in this example results in a $(1000 - 100) + (5000 - 1000) + (100 - 10) = 4990$ PAR-2 margin.

We want to remark that a perfect PAR-2 margin (i.e., the target solver fails to solve all instances and each competing solver solves all instances instantly) over a set of n inputs to be $2 \cdot n \cdot \textit{timeout}$, which in the above example with three inputs and a timeout of 2500 is 15,000 ($3 \cdot 2 \cdot 2500$). In our experiments, we generate 100 inputs, resulting in an optimal score of 500000. Note that the fuzzing algorithm with the largest PAR-2 margin over all fuzzed inputs for a given target solver is deemed the best fuzzer for that target solver. The fuzzer

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	445097.4	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 aims to maximize the PAR-2 score of the target solver, compared to the other fuzzing algorithm within a given time budget.

that is best, as measured by PAR-2 margin, among all fuzzers across all target solvers, is considered the best fuzzer overall.

Visualization: As discussed below, the performance results of the solvers on the fuzzed inputs generated by the baseline fuzzers and BanditFuzz are visualized using *cactus plots*. A cactus plot demonstrates a solvers performance over a set of benchmarks, with the X-axis denoting the total number of solved inputs and the Y-axis denoting the solver timeout in seconds. A point (X, Y) on a cactus plot can be interpreted as the solver can solve X of the inputs from the benchmark set with each input solved within Y seconds. In our setting, cactus plots can be used to visualize the performance separation from the target solver and reference solvers.

4.3 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 [18], **MathSAT5** v5.5.3. a multi theory SMT solver [16], **CVC4** CVC4 1.7-prerelease [git master 61095232] - a multi theory open source SMT Solver [4], and **Colibri** v2070 - A proprietary CP Solver with specialty in FP SMT [8, 30].

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 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.

4.4 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 [6], **Z3seq** v4.8.0 [18], 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 perfect input suite with Z3str3 timing out while CVC4 and Z3seq solve the input nearly instantly.

As in the previous Section 4.3, 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 3 presents the cactus plot for the experiments when Z3seq was the target solver. The cactus plot provides a visualization of the fuzzing objective, aiming to maximize the performance margins between Z3seq and the other solvers collectively[8]. 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.

Discussion of Results with Developers: We shared our tool and benchmarks with the Z3str3 string solver team. The Z3str3 team found the tool to be “invaluable” in localizing performance issues, as well as identifying classes of inputs on which Z3str3 outperforms competing string solvers such as CVC4. For example, we managed to generate a class of instances that had roughly an equal number of string constraints and integer (arithmetic over the length of strings) constraints over which Z3str3 outperforms CVC4. By contrast, CVC4 outperforms Z3str3 when inputs have many str.contains and str.concat constructs. The Z3str3 team is currently working on improving their solver based on the feedback from BanditFuzz.

5 Related Work

Fuzzers for SMT Solvers: We refer to Takanen et al. [47] and Sutton et al. [43] for a detailed overview of fuzzing. While there are many tools and fuzzers for finding bugs in specific SMT theories [34, 7, 12, 11, 28, 28], BanditFuzz is the first performance fuzzer for SMT solvers that we are aware of.

Machine Learning for Fuzzing: Bottinger et al. [9] 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

[8] Cactus plots for Z3str3 and CVC4 solvers can be found on the BanditFuzz webpage.

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. [19] 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 MAB algorithms have been used in various aspects as fuzzing, it has not been used to implement a mutation. Karamcheti et al. [22] trained bandit algorithms to select model-less bitwise mutation operators from an array of fixed operators for greybox fuzzing. Woo et al. [48] and Patil et al. [35] used bandit algorithms to select configurations of global hyper-parameters of fuzzing software. Rebert et al. [37] 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 [33, 32, 2, 50]. 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.

6 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 a relative performance slowdown in a target program relative to a (set of) reference programs. 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 baselines, including random, mutational, and evolutionary fuzzing techniques, and found that it consistently outperforms existing fuzzing approaches. In the future, we plan to extend BanditFuzz to all of SMT-LIB.

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. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org` (2016)
6. 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)
7. 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)
8. Bobot-CEA, F., Chihani-CEA, Z., Iguernlala-OCamlPro, M., Marre-CEA, B.: Fpa solver
9. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. arXiv preprint arXiv:1801.04589 (2018)
10. Brain, M., Tinelli, C., Rümmer, P., Wahl, T.: An automatable formal semantics for ieee-754 floating-point arithmetic. In: Computer Arithmetic (ARITH), 2015 IEEE 22nd Symposium on. pp. 160–167. IEEE (2015)
11. 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)
12. Bugariu, A., Müller, P.: Automatically testing string solvers. In: International Conference on Software Engineering (ICSE), 2020. ETH Zurich (2020)
13. 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)
14. Cesare Tinelli, Clark Barret, P.F.: Theory of unicode strings (draft) (2019), http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml
15. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. pp. 725–741. IEEE (2015)
16. 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)
17. Committee, I.S., et al.: 754-2008 ieee standard for floating-point arithmetic. IEEE Computer Society Std **2008**, 517 (2008)
18. 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)

19. 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)
20. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. ACM SIGPLAN Notices **43**(6), 281–292 (2008)
21. Gupta, A.K., Nadarajah, S.: Handbook of beta distribution and its applications. CRC press (2004)
22. 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)
23. Koza, J.R.: Genetic programming (1997)
24. 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)
25. Lemieux, C., Padhye, R., Sen, K., Song, D.: Perffuzz: Automatically generating pathological inputs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 254–265 (2018)
26. 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)
27. 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)
28. 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)
29. Marijn Heule, Matti Järvisalo, M.S.: Sat race 2019 (2019), http://sat-race-2019.ciirc.cvut.cz/
30. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. In: 15th International Workshop on Satisfiability Modulo Theories (2017)
31. Miller, C., Peterson, Z.N., et al.: Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep (2007)
32. Misherghi, G., Su, Z.: Hdd: hierarchical delta debugging. In: Proceedings of the 28th international conference on Software engineering. pp. 142–151. ACM (2006)
33. 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)
34. Niemetz, A., Preiner, M., Biere, A.: Model-Based API Testing for SMT Solvers. In: Brain, M., Hadarean, L. (eds.) Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT 2017), affiliated with the 29th International Conference on Computer Aided Verification, CAV 2017, Heidelberg, Germany, July 24-28, 2017. p. 10 pages (2017)
35. Patil, K., Kanade, A.: Greybox fuzzing as a contextual bandits problem. arXiv preprint arXiv:1806.03806 (2018)
36. 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)
37. 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)

38. Rümmer, P., Wahl, T.: An smt-lib theory of binary floating-point arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT). p. 151 (2010)
39. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited, (2016)
40. Russo, D.J., Van Roy, B., Kazerouni, A., Osband, I., Wen, Z., et al.: A tutorial on thompson sampling. Foundations and Trends® in Machine Learning **11**(1), 1–96 (2018)
41. Seagle Jr, R.L.: A framework for file format fuzzing with genetic algorithms (2012)
42. Sigaud, O., Buffet, O.: Markov decision processes in artificial intelligence. John Wiley & Sons (2013)
43. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
44. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
45. Sutton, R.S., Barto, A.G., et al.: Reinforcement learning: An introduction. MIT press (1998)
46. Szepesvári, C.: Algorithms for reinforcement learning. Synthesis lectures on artificial intelligence and machine learning **4**(1), 1–103 (2010)
47. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
48. 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)
49. Zalewski, M.: American fuzzy lop (2015)
50. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (Feb 2002)