BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning

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. Satisfiability Modulo Theories (SMT) solvers are fundamental tools in the broad context of software engineering and security research. If SMT solvers are to continue to have an impact, it is imperative we develop efficient and systematic testing methods for them. To this end, we present a reinforcement learning driven fuzzing system *BanditFuzz* that zeroes in on the grammatical constructs of well-formed solver inputs that are the root cause of performance or correctness issues in solvers-under-test. To the best of our knowledge, BanditFuzz is the first machine-learning based fuzzer for SMT solvers.

BanditFuzz takes as input a grammar G describing the well-formed inputs to a set of distinct solvers (say, P_1 and P_2) that implement the same specification and a fuzzing objective (e.g., maximize the relative performance difference between P_1 and P_2), and outputs a ranked list of grammatical constructs that are likely to maximize performance differences between P_1 and P_2 or are root causes of errors in these solvers. Typically, mutation fuzzing is implemented as a set of random mutations applied to a given input. By contrast, the key innovation behind BanditFuzz is the modeling of a grammar-preserving fuzzing mutator as a reinforcement learning (RL) agent that, via blackbox interactions with programs-under-test, learns which grammatical constructs are most likely the cause of an error or performance issue. Using BanditFuzz, we discovered 1,700 syntactically unique inputs resulting in inconsistent answers across state-of-the-art SMT solvers Z3, CVC4, Colibri, MathSAT, and Z3str3 over the floating-point and string SMT theories. Further, using BanditFuzz, we constructed two benchmark suites (with 400 floating-point and 300 string instances) that expose performance issues in all considered solvers. We also performed a comparison of BanditFuzz against random, mutation, and evolutionary fuzzing methods. We observed up to a 31% improvement in performance fuzzing and up to 81% improvement in the number of bugs found by BanditFuzz relative to these other methods for the same amount of time provided to all methods.

Keywords: SMT Solvers, Fuzzing, Reinforcement Learning

1 Introduction

Over the last two decades, many sophisticated program analysis [22], verification [26], and bug-finding tools [15] have been developed thanks to powerful Satisfiability Modulo Theories (SMT) solvers. Precisely because solvers have become such critical components, any errors in these solvers call into question the result produced by analysis tools that use them. Further, the efficiency of SMT solvers significantly impacts the efficiency of modern program analysis and verification tools. Given the insatiable demand for error-free and efficient SMT solvers, these infrastructural tools must be subjected to extensive testing and analysis. While SMT bit-vector and integer solvers have been subjected to heavy testing over the years [14], the same cannot be said for many other theories, e.g., the theory of quantifier-free floating-point arithmetic (FP).

Fuzzing is a powerful way to test SMT solvers [9]. Fuzzers come in many forms. Blackbox random fuzzers on unstructured inputs frequently invoke random string generation or require a database of input seeds that are mutated via random or bit-wise procedures [35, 11, 24, 27]. By contrast, mutational fuzzing for programs that accept highly-structured inputs requires fuzzers to be augmented with appropriate grammars, in order to enable testing of core functionality beyond the parser [41, 27].

Another class of testing methods that has been widely studied is whitebox fuzzing, where a variety of program analysis methods are combined with fuzzing [21, 27]. However, whitebox testing methods may not scale well for large complex programs such as SMT solvers. There has been some effort towards formally verifying SAT solvers [8]. However, we are not aware of any initiative that scales to verifying large state-of-the-art SMT solvers.

In this paper, we introduce a reinforcement learning (RL) based fuzzer, called BanditFuzz, that improves on the aforementioned fuzzing approaches by up to a 31% improvement in PAR-2 score margins for relative performance fuzzing, and up to a 81% increase in bug-revealing inputs. Using BanditFuzz, we generated a database of 1,600 inputs that expose inconsistencies amongst four bleeding-edge FP solvers, namely, Z3 [20], CVC4 [4], MathSAT [18], and Colibri [29]. We also used BanditFuzz to generate a database of 110 instances where two versions (build and debug) of the Z3 sequence string solver [20] disagree on whether or not these inputs are satisfiable (SAT). Additionally, we used BanditFuzz to generate 400 inputs that expose relative performance issues across the aforementioned FP solvers and 300 inputs exposing relative performance issues in the Z3Seq [20], Z3str3 [6], and CVC4 [4] string solvers.

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 or correctness of T is compared), a fuzzing objective (e.g., find bugs or relative performance issues), and outputs a ranked list of *grammatical constructs* (e.g., syntactic tokens, expressions, or keywords over the input language described by G) in the descending order of ones that are most likely to trigger an error or cause performance issues. BanditFuzz also generates 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 (for brevity, we only describe the performance fuzzing mode of BanditFuzz here): BanditFuzz generates well-formed inputs that adhere to G, mutates them in a grammar-preserving manner, and uses RL methods to learn which grammatical constructs are the most likely to cause errors or performance issues in the programs in P. Traditional mutation fuzzers choose or implement a mutation operator at random and are oblivious to the behavior of the programs-under-test. By contrast, 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 [42, 44]. The crucial insight behind BanditFuzz is to maintain a list of grammatical constructs in G, that is ranked based on how likely they are to be the root cause of an error or performance issue. Initially, all grammatical constructs in G are treated as uniformly likely to cause an error by the RL agent. BanditFuzz randomly generates a well-formed input I to begin with, and runs all the programs in P on the input I. In each of its 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 I. It analyzes the results of these runs to provide feedback (i.e., rewards) to the RL agent in the form which constructs were 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., maximizing the relative performance difference between the solvers in P). The process continues until the RL agent converges to a ranking and 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 it isolates or localizes the root cause (in the form of grammatical constructs of well-formed inputs) of bugs or performance issue. This form of isolation is particularly useful to understand problematic behavior in complex programs such as SMT solvers. Further, the paradigm of reinforcement learning is particularly useful for this task in the setting of blackbox fuzzing, since RL methods are powerful enough to navigate a space of inputs in a fashion guided by the feedback they receive via analysis of the input/output behavior of these complex systems.

Differences between BanditFuzz and Delta Debugging: At first glance, BanditFuzz's root cause analysis and error isolation feature may seem similar to delta debugging [49]. However, there are significant differences between these two methods. In delta debugging, a bug-revealing input E is given, and the task of a delta-debugger is to minimize E to get E', by choosing which part of that input to keep (and which one to omit), while ensuring that E' exposes the same error in the program-under-test as E [32, 31, 2, 49]. In contrast, BanditFuzz has two modes associated with it. In its performance fuzzing mode, BanditFuzz generates many inputs that maximize the performance difference between a target and a reference solver, and further ranks the grammatical constructs (in descending order) that are likely the root cause of the lack of performance in the target solver across all the generated inputs. Additionally, in its bug-localization fuzzing mode, BanditFuzz generates many bug-revealing inputs for the program-under-test T, and uses RL to isolate and rank the grammatical constructs that are most likely the root cause(s) of the errors in T across all the generated inputs. We are not aware of the use of delta debugging for revealing performance issues in programs-under-test, nor are we aware of delta debugging tools that construct a ranking of grammatical constructs that are most likely the root cause of errors in the program-under-test across many inputs.

Contributions

1. **First RL-based Fuzzer for FP and String SMT Solvers:** We describe the design and implementation the first RL-based fuzzer for SMT solvers, called BanditFuzz. BanditFuzz uses reinforcement learning, specifically MABs, in order to construct fuzzing mutations over highly structured inputs with the aim of maximizing a performance or bug-finding fuzzing objective. 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.[3]
2. **Empirical Evaluation of BanditFuzz (Fuzzing for Performance):** 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 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 [28]. BanditFuzz outperforms existing fuzzing algorithms (such as random, mutation, and genetic fuzzing) by up to an 31% increase in PAR-2 score margins, for the same amount of resources provided to all methods. We also contribute two large set of inputs discovered by BanditFuzz that contain a combined total of 400 for the theory of FP and 300 for theory of strings that the SMT community can use to test their solvers. We are not aware of any other such tool for performance fuzzing of SMT solvers.
3. **Empirical Evaluation of BanditFuzz (Fuzzing to Find Bugs):** We provide an extensive empirical evaluation of our fuzzer for detecting bugs in SMT solvers and compare with existing fuzzing approaches. We consider four state-of-the-art FP SMT solvers, namely, Z3, CVC4, MathSat, and Colibri, as well as, two string SMT solvers, Z3str3 and a debug build of Z3str3. Using BanditFuzz we found a total of 1,600 bug revealing inputs across FP solvers and 110 bug revealing inputs between z3str and its debug build. We

[3] All our tools and data can be downloaded from the anonymized BanditFuzz website: https://sites.google.com/view/banditfuzz.

observe that BanditFuzz improves on existing fuzzing algorithms by up to an 81% increase in total bug revealing inputs, for the same amount of resources provided to all methods.

2 Preliminaries

2.1 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 [44, 42, 43]. As discussed in the introduction, the reinforcement learning paradigm is particularly suited for modelling mutation fuzzing based in an online corrective feedback setting. In this paper, we specifically deploy multi-armed bandit algorithms [42] (MAB), a class of reinforcement learning algorithms, to learn mutation operators.

Reinforcement learning algorithms are commonly formulated using *Markov Decision Processes* (MDPs) [37, 44, 40]. An MDP is typically specified by a 5-tuple (S, A, T, R, γ), where S is a set of states, A is a set of actions, T is a matrix of state action transition probabilities where $T[s][a][s']$ denotes the probability of entering state s' from state s upon taking action a, R is a matrix of rewards where $R[s][a][s']$ denotes the reward received for transitioning to s' from s when executing action a, and $0 \leq \gamma \leq 1$ is a discount factor indicating how rewards should be scaled down at each time step.

The multi-armed bandit (MAB) problem is a common reinforcement learning problem based on an MDP with a single state $S = \{s_0\}$ and a finite set of actions A. Because there is only a single state, T is empty. What remains, is the unknown probability distribution of rewards R over A. In the context of MAB, actions are often referred to as arms (or bandits). 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).

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 an action in A. The MAB learns a *policy* – a stochastic process of how to select actions from A. The learned policy balances the exploration/exploitation trade-off, i.e., a MAB algorithm will select every action an infinite number of times in the limit, but will select the action(s) with the highest expected reward more frequently.

We include three solutions to the MAB problem into BanditFuzz. However, for brevity we will focus on 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 [23]. Intuitively, a Beta distribution is a model random distribution for the expected value of a Bernoulli distribution. It is maintained by parameters $\alpha - 1$ the samples of 1, and $\beta - 1$ the samples of 0, from the underlying

Bernoulli distribution. Online, the bandit samples each arm's distribution and greedily picks its arm based on the maximum sampled value. Upon completing the action, α is incremented on reward, otherwise β is incremented. For more on Thompson sampling we refer to Russo et al. [38].

2.2 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, arrays, floating-point, and strings that are particularly suitable for verification, program analysis, and testing. We refer the reader to the Handbook of Satisfiability for more details on SMT solvers [7]. The SMT-LIB is an initiative to standardize the language and specification of several theories of interest [5]. In this paper, we will 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. [36] with several recent revisions. In this paper, we consider the latest version, by Brain et al. [12]. 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 [19].

Quantifier-free Theory of Strings The SMT-LIB standard for the theory of strings is in development. We use the current draft version. 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 predicates { str.prefixof, str.suffixof, str.in_re}. We refer the reader to the publicly available draft of the SMT-LIB standard for further details on the syntax and semantics of these functions and predicates [16]. We further clarify that the above lists of predicates, operators, and terms are intentionally incomplete. The above list was carefully selected to include only those that are supported amongst all solvers considered in this paper.

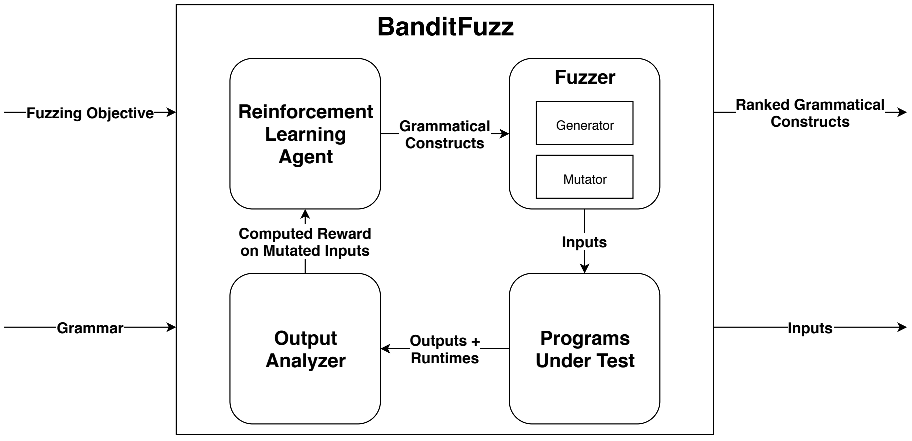

Fig. 1. Architecture of BanditFuzz

2.3 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. Fuzzers frequently implement input generators, which can deploy a *model-based* or a grammar-based or generative strategy where all inputs produced conform to the grammar of the program. Alternatively, fuzzers can implement *model-less* generators, which produce random inputs. Mutation fuzzers are a popular alternative to random fuzzers. 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 [48, 17, 30, 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 [34, 39, 25].

3 BanditFuzz

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 likely to be the cause of a bug or performance issue in the solvers-under-test. The ability of BanditFuzz to isolate those grammatical constructs that trigger erroneous behavior or performance issues, in a blackbox fuzzing 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., find bugs or 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 trigger an error or 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 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 an MAB agent) within a feedback loop to learn which grammatical constructs of G are the most likely culprits that may cause errors or 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 trade-off. 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 trigger an error or cause performance issues, in descending order.

BanditFuzz for Performance Fuzzing: 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.

BanditFuzz for Bug Localization: In the fuzzing for bug localization fuzzing mode, BanditFuzz performs the above-described analysis to produce a ranked list of grammatical constructs that maximize the likelihood that a target solver T will disagree with reference solver R. The only difference between the performance fuzzing algorithm and the bug localization algorithm is in the calculation of rewards. At a high level, BanditFuzz in bug localization mode, tries to identify the common features across a set of inconsistency revealing inputs. This is in contrast to delta-debugging-like techniques that focus on a single input. We describe both the performance fuzzing and bug localization modes in greater detail below.

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 [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: We first describe the generator component of the fuzzer. Initially, BanditFuzz generates a random well-formed instance. This is done by the Fuzzer box in Figure 1 using the input grammar G. In this paper, we exclusively consider the input grammar of the FP and string SMT-LIB theories. We use the random abstract syntax tree (AST) generation procedure for SMT-LIB theory of strings built into the StringFuzz [9] random fuzzer. 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 pre-determined 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, a grammatical construct of the same type (predicate, operator, or rounding mode, etc.) will uniformly at random

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

be selected. 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 an 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:

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

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

For further analysis, consider the additional asserted equation:

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

if the agent elects to insert $fp.add$, then there are four[5] 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 the overall calculated reward (e.g., the fuzzing objective such that the performance difference between a solver-under-test and a reference solver is maximized).

3.3 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, that 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

[5] 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.

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

```
1: procedure BanditFuzz($G$)
2:     Instance $I \leftarrow$ a randomly-generated instance over $G$          ▷ Fuzzer
3:     Run target solver $T$ and reference solver $R$ on $I$
4:     Compute $PerfScore(I)$                                                   ▷ OutputAnalyzer
5:     $\theta = 2\cdot$ Solver timeout
6:     while fuzzing time limit not reached or $PerfScore(I) < \theta$ do
7:         $construct \leftarrow RL\ AGENT$ picks a grammatical construct       ▷ RL Agent
8:         $I' \leftarrow$ Mutate $I$ with $construct$                          ▷ Fuzzer
9:         Run each solver $s \in P$ on $I$
10:        if $PerfScore(I', P) > PerfScore(I, P)$ then                         ▷ OutputAnalyzer
11:            Provide reward to $RL\ AGENT$ for $construct$
12:            $I \leftarrow I'$
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
```

iterations of BanditFuzz's feedback loop. We deploy two differing fuzzing feedback loops (see Algorithm 1 and 2). In the following subsections, we discuss them 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). BanditFuzz works as follows in the performance fuzzing mode. Initially, the fuzzer generates a well-formed input I (sampled uniform-at-random). BanditFuzz then executes both the target solver T and reference solver R on I, and records their respective runtime (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 runtime(I, T) refers to the runtime of the target solver T on I, and runtime(I, R) the runtime of the reference solver R on I. If the target solver reaches the timeout, we set runtime(I, T) to be $2 * timeout$—PAR-2 scoring in the SMT competition. In the same iteration, BanditFuzz mutates the input I to I' and computes the 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

Algorithm 2 BanditFuzz's Bug Localization Feedback Loop. Also refer to BanditFuzz architecture in Figure 1.

```
 1: procedure BANDITFUZZBUG($S, G$)
 2:     $D \leftarrow \emptyset$
 3:     while fuzzing time limit not reached do
 4:         $I \leftarrow$ a randomly generated input over $G$
 5:         if $I$ is inconsistent then
 6:             $D \leftarrow D \cup \{I\}$
 7:             Continue Loop
 8:         else
 9:             $construct \leftarrow$ Have $AGENT$ select a grammatical construct (action).
10:             $I' \leftarrow$ Mutate $I$ with $construct$
11:             if $I'$ is inconsistent then
12:                 Provide reward to $AGENT$ for $construct$
13:             else
14:                 Provide no reward to $AGENT$ for $construct$
15:                 $D \leftarrow D \cup \{I'\}$
16:             end if
17:         end if
18:     end while
19:     return $D$,Ranking of Constructs from $RL\ AGENT$
20: end procedure
```

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 remain 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.

Computing Rewards for Fuzzing for Bug Localization: Algorithm 2 describes BanditFuzz's bug-finding mode. We elaborate on its reward computation procedure here. Just like in its performance fuzzing mode, BanditFuzz initially generates a well-formed input I and executes both the target and reference solvers on I. (For simplicity, assume that the reference solver produces the correct answer on I.) If the answers given by T and R differ, it means that there is an error in T. When a bug-revealing input I' is identified, BanditFuzz's OutputAnalyzer assigns a reward to the construct γ that differentiates I' from I. The RL agent uses Thompson sampling to update the ranks of the grammatical constructs in I based on the rewards thus obtained.

At a high level, BanditFuzz collects the bug-revealing inputs (to return them to the user upon completion), and learns which language features are common among the set. This learning helps us produce more inconsistency revealing

Target Solver	BanditFuzz	Random	Mutational	Evolutionary	% 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

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

inputs, and explain to the solver developer which language features are likely to be responsible for the bugs. The explanation is the ranked list described earlier.

4 Empirical Evaluation: Fuzzing for Performance

In this section, we present an evaluation of BanditFuzz for performance fuzzing. 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, and otherwise ran each with default settings.

4.1 Experimental Setup

Baselines: We compare BanditFuzz with three different widely-used approaches: random, mutation, and evolutionary fuzzing. We describe the three approaches below. We do not compare against several existing fuzzing systems, such as AFL and SkyFire [48, 46], because it is difficult to apply them to large and sophisticated grammars, such as those used by SMT solvers. There is a large literature on fuzzing specific pieces of software [27], in addition to SMT Solvers [9, 14, 13].

1. **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.

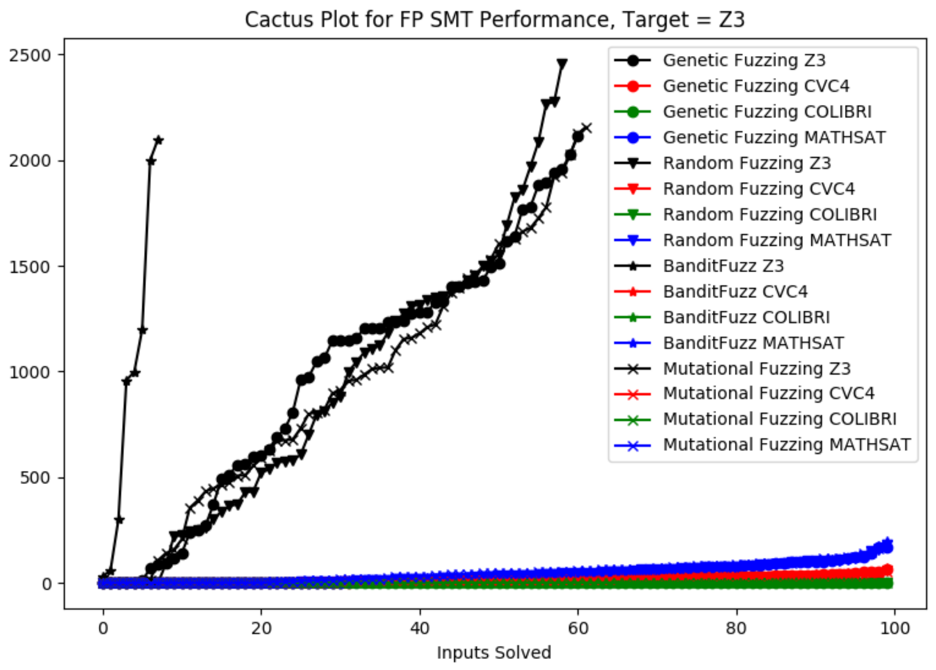

Fig. 2. Cactus Plot for targeting the Z3 FP Solver against reference solvers CVC4, Colibri, and MathSAT.

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 [48, 17, 30, 27]. 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 3.2 .
3. **Evolutionary Fuzzing** – Evolutionary fuzzing algorithms maintain a population of inputs. Every generation, only the fittest members of the population survive, and new members are created through random generation, breeding, and mutation. [34, 39]. We implement evolutionary fuzzing as follows:
   1 Initialize: Random Fuzzing is deployed for ten queries to initialize the population.
   2 In a loop until global timeout:
      (a) Remove all inputs from the population except the four highest scoring inputs.
      (b) Compute a new population composed of
         i. mutated inputs for each of the surviving inputs, and
         ii. fresh randomly generated inputs.
      (c) Solve all new inputs.

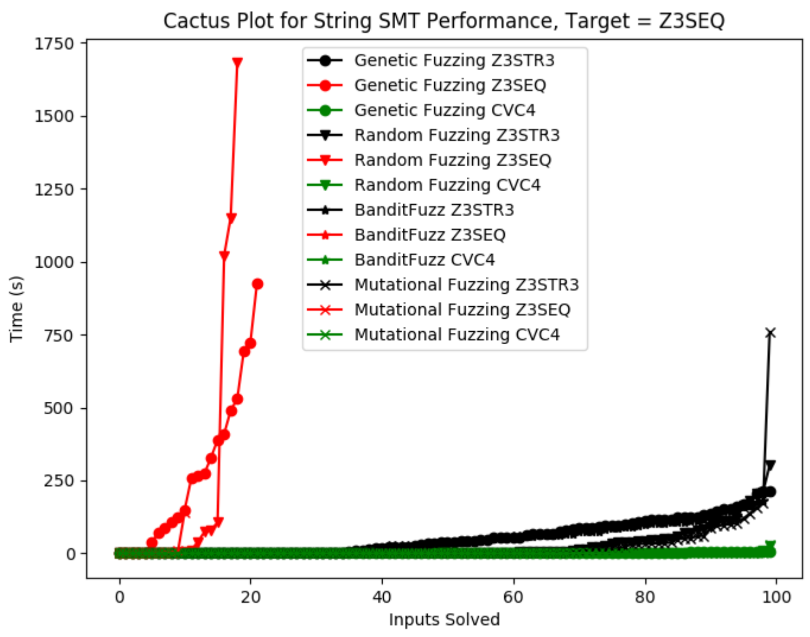

Fig. 3. Cactus Plot for targeting the Z3Seq string solver against reference solvers CVC4 and Z3str3.

(d) If a mutated input improves on its derived input, discard its derived input.

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 24 hours with the aim of constructing a inputs that maximize 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.

Quantitative Evaluation: For each solver/input pair, we record the CPU time (i.e., the process time for the solver run on the fuzzer-generated input). 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 labeled 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.

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 define the notion of perfect PAR-2 margin over a set of n inputs to be $n{*}2{*}\textit{timeout}$, which in the above example with 3 inputs and a timeout of 2500 is 15,000 $(3 * 2 * 2500)$. Note that the fuzzing algorithm that maximizes the PAR-2 margin over all fuzzed inputs for a given target solver is deemed the best fuzzer for that target solver. The fuzzer 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 the performance separation from the target solver and reference solvers.

4.2 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 [20], **MathSAT5** v5.5.3. a multi theory SMT solver [18], **CVC4** v1.7 - a multi theory open source SMT Solver [4], and **Colibri** v2070 - A proprietary CP Solver with specialty in FP SMT [10, 29].

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 presents 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.

4.3 Performance Fuzzing – 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** v 4.8.0 [20], and CVC4 v1.7 [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 4.2, 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, maximizing the performance margins between Z3seq and the other solvers collectively. The line for BanditFuzz Z3seq is not rendered on the plot as none of the returned inputs were solved by Z3seq. Cactus plots for the target solvers of Z3str3 and CVC4 can be found on the anonymized BanditFuzz webpage.[6]

5 Empirical Evaluation: Fuzzing for Bug Localization

In this section, we present an evaluation of BanditFuzz for finding bugs in SMT Solvers. Specifically, we ask, how many inputs can we discover that are *inconsistent* (See Section 3.3) across solvers? In the previous section, the fuzzing oracle was wall-clock time. A fuzzing oracle for bugs is less obvious. We use the solver output *inconsistency* as a fuzzing oracle. More details are in Section 3.3.

5.1 Experimental Setup

Baselines: We refer to the baselines from Section 4 for bug fuzzing. Random fuzzing is left unmodified with the exception of the fuzzing objective. Evolutionary fuzzing is dropped due to the binary nature of fitness. Mutation Fuzzing is modified to follow the structure of Algorithm 2 with random grammatical construct selection.

Visualization and Quantitative Evaluation: The aim of this particular fuzzing experiment is to generate as many syntactically unique inputs that trigger a bug while minimizing net time [27]. Anything further begins to leave the

[6] https://sites.google.com/view/banditfuzz

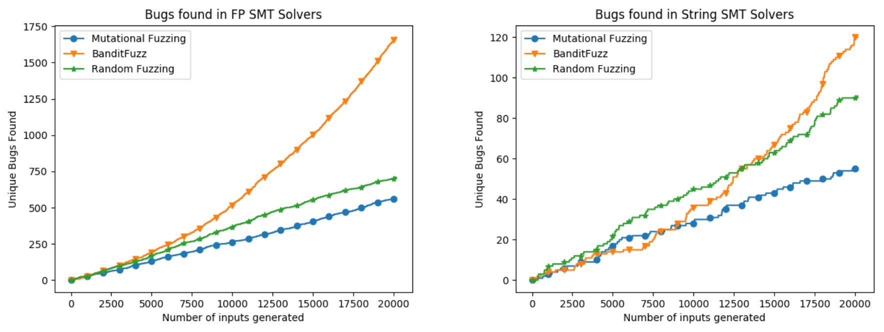

Fig. 4. Plots of bugs over inputs generated and solved for all considered fuzzing algorithms for the FP SMT solvers (left) and string SMT solvers(right).

scope of blackbox fuzzing requiring grey/whitebox fuzzing. We visualize performance by showing bugs over the total number of inputs generated and solved. We evaluate performance based on the total number of syntactically unique bugs upon termination.[7]

We generate inputs with three to five assertions and with depth of two to four. We run each solver with a 60 second timeout. We run each fuzzing algorithm for 20,000 input queries over the set of solvers S. The generator constraints are intentionally set to create short instances. The rationale for this is to create instances that are short to both aid the debugging process and help limit the runtime of the solver.

5.2 Bug Fuzzing – FP SMT Solvers

In our bug fuzzing evaluation of BanditFuzz we consider the same set of FP solvers used in Section 4.2. Figure 4 (left) presents the bug inputs generated and solved for each considered fuzzing algorithm. Unlike performance fuzzing where only the final inputs were presented, we are able to visualize the learning process of the bandit as it converges to an action selection policy. In the beginning, BanditFuzz is in a purely exploration phase of the learning process, behaving very similarly to random mutational fuzzing. However, we observe that at around 7500 queries the bandit begins to separate itself, isolating the keywords that are most likely to induce a bug.

Similarly as before in Section 4, we can extrapolate from the learned model which grammatical constructs are most likely to reveal an inconsistent input over the set of solvers. The top 3 such are RNE, fp.eq, and fp.isNegative.

[7] We do note that syntactically unique inputs don't necessarily correspond to distinct errors in a program-under-test. While we do not do so here, code coverage is another metric by which fuzzers can be compared.

5.3 Bug Fuzzing – String SMT Solvers

In our bug fuzzing evaluation of BanditFuzz, we consider a different set of string solvers.

1. **Z3STR3** v4.8.6 – The release build of the Z3 Solver used in Section 4
2. **Z3STR3-DEBUG** v4.8.6 – The debug build of the above solver.

These two solvers are very similar, resulting in a very challenging fuzzing problem. This experiment is included to test the limits of BanditFuzz.

Figure 4 (right) presents the bugs inputs generated and solved for each considered fuzzing algorithm. Unlike the FP experiment, BanditFuzz took several additional iterations to latch on to the precise subset of the action space to induce bugs more frequently than in the FP setting. However, well before 15,000 iterations, BanditFuzz surpasses all considered baselines. We suspect this slowdown to be cause by the increased difficulty of the problem, while in the FP setting, the solvers are significantly more diverse.

Once again, we can extrapolate from the learned model which grammatical constructs were most likely to cause an inconsistent answer amongst Z3STR3 and Z3STR3-DEBUG. The top three grammatical constructs were: str.in_re, str.substr, and str.++.

6 Related Work

SMT Benchmarks and SMT Fuzzing: The SMT-LIB community has collected a large set of benchmarks coming from a wide variety of domains to test and evaluate their solvers. Unfortuantely, these benchamrks are static and expensive to create. BanditFuzz can be used to cheaply and automatically add to these benchmark suites. Fuzzing is a dominant approach for testing software systems, and the literature is too vast for a complete review here. We refer the reader to Takanen et al. [45] and Sutton [41] for a detailed overview of fuzzing. In the context of SMT solvers, fuzzers exist for specific theories such as strings [9] and bit-vectors [14]. However, we are not aware of any fuzzer for FP SMT solvers. Further, we are not aware of any fuzzers, other than BanditFuzz, that can be used to isolate or localize grammatical constructs that are the root causes of bugs or performance issues.
Fuzzing and Machine Learning: Existing fuzzers have used machine learning in some capacity. Bottinger et al. [11] introduce a deep Q learning algorithm for fuzzing model-free inputs. This approach 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 this end, Godefroid et al. [21] uses neural networks to learn an input grammar over complicated domains such as PDF and then use the learned grammar for model-guided fuzzing. It is not clear to us whether these methods can deal with the complex grammars of SMT solvers. 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 errors or performance issues in programs-under-test.

More specifically, bandit algorithms have been used in various aspects as fuzzing, but never to implement a mutation, and instead as wrappers to existing fuzzers. Karamcheti et al. [24] trained bandit algorithms to select model-less bitwise mutation operators from an array of fixed operators for greybox fuzzing. Woo et al. [47] and Patil et al. [33] used bandit algorithms to select configurations of global hyper-parameters of fuzzing software. Rebert et al. [35] 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 listed 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 or random or evolutionary fuzzing algorithm.

Delta Debugging: BanditFuzz has a close relationship with delta debugging but solves a different problem. In delta debugging, a bug-revealing input E is given, and the task of a delta-debugger is to minimize E to get E', by choosing which part of that input to keep (and which one to omit), while ensuring that E' exposes the same error in the program-under-test as E [32, 31, 2, 49]. BanditFuzz, on the other hand, generates and examines a set of—bug or performance revealing—inputs by leveraging reinforcement learning. The goal of BanditFuzz is to discover patterns over the entire generated set. Specifically, BanditFuzz finds and ranks the language features that are the root cause of errors in the program-under-test. Furthermore, unlike delta-debugging, BanditFuzz generates inputs that reveal relative performance differences on pairs of programs.

7 Conclusions and Future Work

In this paper, we presented BanditFuzz, a 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 compared BanditFuzz against a baseline of random, and evolutionary fuzzing techniques, and found that it outperforms existing approaches. More specifically, our fuzzer gave an 31% improvement in performance fuzzing and up to an 81% improvement in bug fuzzing. In our experiments, we were able to learn a policy that describes how to maximize the difference between solver runtimes. In the future, we plan to extend BanditFuzz to other SMT decision procedures as well non SMT settings.

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. Biere, A., Heule, M., van Maaren, H.: Handbook of satisfiability, vol. 185. IOS press (2009)
8. Blanchette, J.C., Fleury, M., Lammich, P., Weidenbach, C.: A verified sat solver framework with learn, forget, restart, and incrementality. Journal of Automated Reasoning **61**(1-4), 333–365 (2018)
9. 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)
10. Bobot-CEA, F., Chihani-CEA, Z., Iguernlala-OCamlPro, M., Marre-CEA, B.: Fpa solver
11. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. arXiv preprint arXiv:1801.04589 (2018)
12. 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)
13. Brummayer, R.: Fuzzsmt (2009)
14. 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)
15. 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)
16. Cesare Tinelli, Clark Barret, P.F.: Theory of unicode strings (draft) (2019), http://smtlib.cs.uiowa.edu/theories-UnicodeStrings.shtml
17. Cha, S.K., Woo, M., Brumley, D.: Program-adaptive mutational fuzzing. In: 2015 IEEE Symposium on Security and Privacy. pp. 725–741. IEEE (2015)
18. 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)
19. Committee, I.S., et al.: 754-2008 ieee standard for floating-point arithmetic. IEEE Computer Society Std **2008**, 517 (2008)
20. 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)
21. Godefroid, P., Kiezun, A., Levin, M.Y.: Grammar-based whitebox fuzzing. In: ACM Sigplan Notices. vol. 43, pp. 206–215. ACM (2008)

22. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. ACM SIGPLAN Notices **43**(6), 281–292 (2008)
23. Gupta, A.K., Nadarajah, S.: Handbook of beta distribution and its applications. CRC press (2004)
24. 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)
25. Koza, J.R.: Genetic programming (1997)
26. 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)
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. Marijn Heule, Matti Järvisalo, M.S.: Sat race 2019 (2019), http://sat-race-2019.ciirc.cvut.cz/
29. Marre, B., Bobot, F., Chihani, Z.: Real behavior of floating point numbers. In: The SMT Workshop (2017)
30. Miller, C., Peterson, Z.N., et al.: Analysis of mutation and generation-based fuzzing. Independent Security Evaluators, Tech. Rep (2007)
31. Misherghi, G., Su, Z.: Hdd: hierarchical delta debugging. In: Proceedings of the 28th international conference on Software engineering. pp. 142–151. ACM (2006)
32. 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)
33. Patil, K., Kanade, A.: Greybox fuzzing as a contextual bandits problem. arXiv preprint arXiv:1806.03806 (2018)
34. 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)
35. 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)
36. Rümmer, P., Wahl, T.: An smt-lib theory of binary floating-point arithmetic. In: International Workshop on Satisfiability Modulo Theories (SMT). p. 151 (2010)
37. Russell, S.J., Norvig, P.: Artificial intelligence: a modern approach. Malaysia; Pearson Education Limited, (2016)
38. 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)
39. Seagle Jr, R.L.: A framework for file format fuzzing with genetic algorithms (2012)
40. Sigaud, O., Buffet, O.: Markov decision processes in artificial intelligence. John Wiley & Sons (2013)
41. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
42. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT press (2018)
43. Sutton, R.S., Barto, A.G., et al.: Reinforcement learning: An introduction. MIT press (1998)
44. Szepesvári, C.: Algorithms for reinforcement learning. Synthesis lectures on artificial intelligence and machine learning **4**(1), 1–103 (2010)

45. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
46. Wang, J., Chen, B., Wei, L., Liu, Y.: Skyfire: Data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). pp. 579–594. IEEE (2017)
47. 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)
48. Zalewski, M.: American fuzzy lop (2015)
49. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering **28**(2), 183–200 (Feb 2002)