BanditFuzz: Fuzzing SMT Solvers with Multi-Agent Reinforcement Learning *

Joseph Scott[1], Trishal Sudula[1], Hammad Rehman[1], Federico Mora[2], and Vijay Ganesh[1]

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

Abstract. We present BanditFuzz, a multi-agent reinforcement learning (RL) guided performance fuzzer for state-of-the-art Satisfiability Modulo Theories (SMT) solvers. BanditFuzz constructs inputs that expose performance issues in a set of target solvers relative to a set of reference solvers, and is the first performance fuzzer that supports the entirety of the theories in the SMT-LIB initiative. Another useful feature of BanditFuzz is that users can specify the size of inputs they want, thus enabling developers to construct very small inputs that zero-in on a performance problem in their SMT solver relative to other competitive solvers. We evaluate BanditFuzz across 52 logics from SMT-COMP '20 targeting competition-winning solvers against runner-ups. We baseline BanditFuzz against random fuzzing and a single agent algorithm and observe a significant improvement, with up to a 82.6% improvement in the margin of `PAR-2` scores across baselines on their respective benchmarks. Furthermore, we reached out to developers and contributors of the CVC4, Z3, and Bitwuzla solvers and provide case studies of how BanditFuzz was able to expose surprising performance deficiencies in each of these tools.

1 Introduction

In recent years, efficient Satisfiability Modulo Theories (SMT) solvers have dramatically impacted many areas of software engineering and security. Applications of these tools range from program analysis [13,24,19], synthesis [44,39] , model checking [26,14,1], test case generation [12], and neural network verification [25], to name just a few.

With efficient SMT solvers being the catalyst for numerous developments in academia and industry, there is an insatiable demand for evermore powerful solvers. To this end, researchers have spent decades optimizing these tools. Regrettably, despite these advances, SMT solvers are prone to hard-to-find performance deficiencies. While the worst-case complexity of the problems solved

by SMT solvers can be very high, they can be frustratingly slow on relatively simple formulas. Such performance deficiencies can be due to developer oversight (e.g., missing rewrite rules or unoptimized code and data structures) or the result of hard-to-entangle interactions of solver heuristics. If solvers are to continue to impact industry and fuel further research, it is imperative that there be an initiative to find and eliminate such performance deficiencies where possible.

In this paper, we make a case for the use of software performance fuzzing [47,45,28] to systematically find such deficiencies in state-of-the-art SMT solvers. Software fuzzing techniques have had tremendous impacts in making SMT Solvers more robust [7,10,49,48,34], and there is no reason why performance fuzzers cannot have a similar impact. While it is still a relatively a new field, performance fuzzing is already showing promise in many domains despite the difficulty of the problem of finding suitable inputs that expose performance issues in programs-under-test [27,23].

This paper presents the BanditFuzz tool, a performance fuzzer that supports the entirety of the theories in SMT-LIB. We define the notion of "performance issue", in the SMT solver setting, in a relative sense. That is, we say that solver A is less performant on an input I relative to solver B, if solver B (that supports the same input language as A) can solve I significantly faster. This is very natural, since if both solvers-under-test are not able to solve an input, it doesn't unambiguously point to a performance issue in either. However, when one solver is significantly faster than a competing one on a given input, there is no question that the slower solver has a performance issue.

How BanditFuzz works: The input to BanditFuzz is a set of target solvers, a set of reference solvers, and a constraint (e.g., size of input desired, the input language of the solvers), and its output is a single benchmark or test input, such that a quantity we refer to as the "performance margin" between the target solver and reference solver is maximized (the tool is designed to be run over multiple processes to create a benchmark suite). Intuitively, the performance margin can be defined as difference between the runtimes (or `PAR-2` scores [31]) of a target and a reference solver on a given input.

Internally, BanditFuzz uses a two-agent reinforcement learning (RL) method to mutate a randomly-generated input such that over time the performance margin between a target and a reference solver is maximized. More precisely, first an input benchmark is randomly-generated and queried across all solvers. One of the agents learns how to mutate a benchmark by inserting and replacing the grammatical constructs of the SMT-LIB language, respecting the size constraints set forth by the user. More precisely, this agent manages an exploration vs. exploitation trade-off between trying new grammatical constructs (explore) vs. inserting ones that have been shown to increase the performance margin (exploit). The other agent manages the exploration vs. exploitation trade-off between generating new inputs (explore) or mutating the best-observed input (exploit). Fuzzers, including single-agent RL fuzzers, are notorious for getting stuck in local minima [41,16,29,17]. This two-agent RL method, by contrast, may avoid getting stuck in local minima.

Contributions.

Specifically, we make the following contributions in this paper:

1. **BanditFuzz: A Multi-Agent RL Performance Fuzzing Algorithm.** To the best of our knowledge, BanditFuzz is the first multi-agent RL performance fuzzing algorithm for SMT solvers that supports the entirety of SMT-LIB. The BanditFuzz tool includes two agents, one which learns how to mutate the best observed input [42] and another to help prevent the tool getting stuck in a local minimum (Section 3).
2. **Empirical Evaluation.** We provide an implementation of the performance fuzzer BanditFuzz [42] and lift it to the entirety of the theories in the SMT-LIB initiative, namely, Arrays, Bit-Vectors, Booleans, Floating-Point, Integers, Reals, Strings, Uninterpreted Functions, and all combinations thereof in both quantified and quantifier-free logics (Sections 3, 5, 6). To test Bandit-Fuzz, we perform an extensive empirical evaluation across all 52 logics that were tested in SMT-COMP '20, with the aim of finding benchmarks where competition winners are slow relative to runner up solvers. We provide to the community a set of 1500 benchmarks across all logics, exposing relative performance issues in state-of-the-art competition-winning SMT Solvers. To validate the efficiency of BanditFuzz, we baseline it against random fuzzing and observe BanditFuzz to consistently outperform random fuzzing by up to a 82.6% increase in `PAR-2` margins (Sections 3 and 5).
3. **Case studies of BanditFuzz with Solver Developers.** To further demonstrate the usefulness of BanditFuzz, we include three case studies of Bandit-Fuzz being used by solver developers and contributors. Specifically, developers were able to use BanditFuzz to find performance issues in the Z3 [15], CVC4 [3], and Bitwuzla [32,33] SMT Solvers (Section 3).

The rest of this paper is structured as follows: Section 2 provides the necessary background on reinforcement learning, SMT, and software fuzzing. Section 3 gives a technical description of BanditFuzz. Section 5 describes how to use BanditFuzz, Section 6 gives an experimental evaluation of BanditFuzz over 52 from the SMT-LIB community, Section 7 goes over case studies with solver developers using BanditFuzz, Section 8 mentions threats to the validity of BanditFuzz, and Section 10 concludes the paper and discusses future work.

2   Background on Reinforcement Learning

In this section, we provide the necessary background and terminology of Reinforcement Learning, Satisfiability Modulo Theories, and Software Fuzzing for this paper.

2.1   Reinforcement Learning

There is a large literature on reinforcement learning and we refer the reader to the book by Sutton et al., on this topic [46]. In RL, an agent navigates an

environment by taking actions to maximize the received reward. The multi-armed bandit (MAB) problem is a well-known RL problem based on a Markov Decision Process with a single state and a finite set of actions A. Since there is only a single state, MABs can be solved using computationally cheap algorithms relative to algorithms for other RL problems [46]. The agent solving the MAB computes an approximation of the 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, as the arm or lever of a slot machine is referred to as a one-armed bandit, and MABs refer to several slot machines. The goal of the MAB agent is to maximize its reward by playing a sequence of actions (e.g., selecting which band/lever to pull).

In practice, MAB problems are commonly modelled so rewards are sampled from an unknown Bernoulli distribution (e.g., rewards are in $\{0, 1\}$). The MAB agent attempts to approximate the expected value of reward from the Bernoulli distribution for each action in A. Over time, the agent uses these distributions to form a *policy* – a stochastic process of how to select actions from A. This policy must remain privy to the exploration/exploitation trade-off, i.e., an MAB algorithm selects every action an infinite number of times, but selects the action(s) with the highest expected reward more frequently.

There are several algorithms for the MAB problem. In this paper, we exclusively consider *Thompson Sampling*. In Thompson Sampling, an agent maintains a Beta distribution for each action in the action space A. Beta distributions are derived from Gamma distributions, and have a long history with numerous applications. We refer the reader to Gupta et al. on Beta and Gamma distributions [20]. In the context of Thompson Sampling, a Beta distribution acts as continuous model of the expected value of a Bernoulli distribution. It is maintained by two shape parameters α the samples of 1, and β the samples of 0, from the underlying Bernoulli distribution. The agent selects an action by sampling each action's Beta distribution and greedily picks the action based on the maximum over the sampled values. Upon taking the action, α is incremented on reward, otherwise β is incremented. For more on Thompson sampling we refer to Russo et al. [40].

2.2   Satisfiability Modulo Theories

Satisfiability Modulo Theories (SMT) solvers are decision problems on first-order theories such as integers, bit-vectors, arrays, floating-point, and strings that are particularly suitable for verification, program analysis, and testing. The SMT-LIB provides a standardized syntax and semantics for several first-order theories and logics [4]. In this paper, we use the following acronyms (given in brackets), optionally in combination, to refer to various SMT-LIB logics: Quantifier Free (QF), Theory of Arrays (A), Uninterpreted Functions (UF), Bit-Vectors (BV), Floating-Point (FP), Strings (S), Integers (I), Reals (R), or mixed (IR). Further, arithmetic over I and R can be linear (e.g., LIA, LRA), or nonlinear (e.g., NIA, NRA), or difference logics (IDL, RDL). For combinations of such, their order
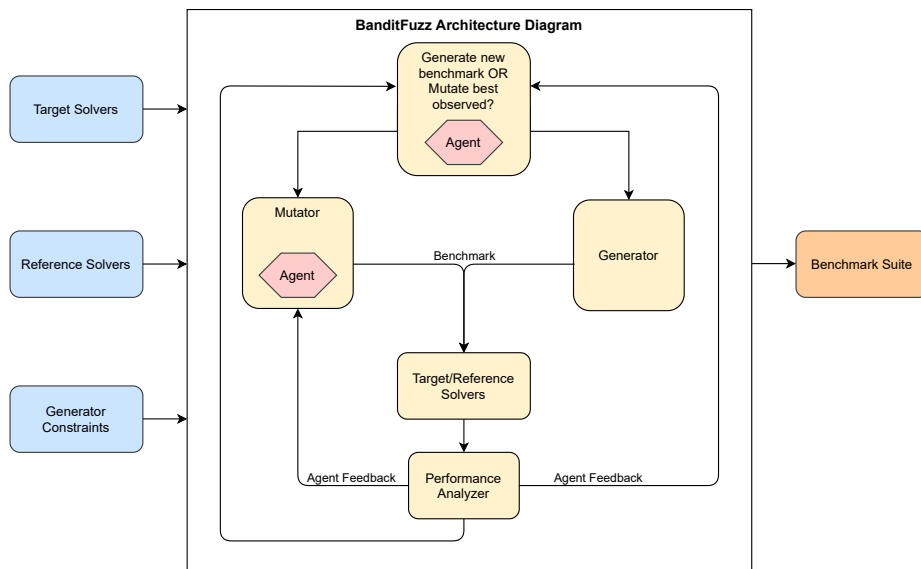
Fig. 1   Architecture Diagram of BanditFuzz. *The BanditFuzz tool deploys two unique agents: one is a mutator agent that learns how to mutate the best observed input, while the other agent aims to assist in the prevention of getting stuck in local minima. Both agents learn an action selection policy in a feedback loop based on the empirically collected data over the course of running the target and reference solvers over the generated benchmarks.*

appears in the order they are written above. Each theory defines various operators/functions, predicates, terms, sorts, and generalized keywords (e.g., assert, check-sat, Int, Float32). For a full list of these, as well as more details on syntax and semantics, we refer to the SMT-LIB standard [4].

2.3   Software Fuzzing

Software fuzzing is a vast and active field of research. We refer to a recent survey that provides an overview of the field [28]. A *fuzzer* is a program that automatically generates inputs for a target program-under-test. A fuzzer is *blackbox* if it does not have access to the program-under-test. Fuzzers that are *model-based*, implement a *generator* which samples from the space of well-formed inputs. A *mutation* is a mapping from the space of inputs to the space of inputs.

3   BanditFuzz: A Multi-Agent RL-Guided Performance Fuzzer

In this section, we describe the BanditFuzz fuzzing framework. BanditFuzz leverages reinforcement learning (RL) to guide a random fuzzer through a space of inputs. The architecture of BanditFuzz is presented in Figure 1.

BanditFuzz takes as input a set T of target solvers, a set R of reference solvers and a set of constraints (e.g., size of the generated test input desired, and the SMT language L of the solvers in T and R (we refer to this as Generator Constraints in Figure 1), and outputs a benchmark I that maximizes the performance margin between the solvers in T and R. The output of BanditFuzz is a benchmark or benchmark suite.

3.1   The Performance Margin

Formally, BanditFuzz solves a search problem to find a solver input I over the language L that maximizes the performance margin between T and R

$$\max_{I \in L} \phi(T, R, I)$$

where ϕ is a scoring function. In this paper, we will exclusively consider a scoring function of the `PAR-2` score margin between the best performing target solver and worst performing reference solver. More formally, we score each input I with respect to T, R as follows:

$$\phi(T, R, I) = \min_{t \in T}(\texttt{PAR-2}(t, I)) - \max_{r \in R}(\texttt{PAR-2}(r, I))$$

where the `PAR-2` function returns twice the wallclock timeout if the solver fails to solve the input, otherwise, the wallclock runtime. `PAR-2` is a useful metric that quantifies a tools' performance over a benchmark suite and is used to determine winners in the SAT competitions [31]. These calculations correspond to the 'Performance Analyzer' in Figure 1.

3.2   BanditFuzz: The Multi-agent RL-based Fuzzing Algorithm

This tool paper presents an implementation of the BanditFuzz algorithm with three key improvements. First, BanditFuzz supports all theories in the SMT-LIB standard [4] and all their combined corresponding logics. The action set of the mutator agent is the set of grammatical constructs across all enabled logics.

The second major change is that this tool is now multi-agent. The BanditFuzz tool includes a second agent to assist in preventing the tool from getting stuck in local minima, which is a major problem in fuzzing in general [41,16,29,17]. A key contribution of this paper is an additional agent with the following **action set**: {Mutate the best observed benchmark, Randomly Sample from L}. The previous approach had a fixed alternation scheme of randomly sampling and mutating the best observed input, which posteriori, resulted in the algorithm

frequently getting stuck in local minima, as it is oblivious to all previously collected empirical data. The second agent uses a similar **reward signal** if the most recent benchmark improves on the best observed benchmark, reward is received, otherwise no reward is received.

3.3   The Original BanditFuzz Algorithm vs. The Current Version

The original BanditFuzz fuzzing algorithm proposed by Scott et al. [42] is a mutational fuzzer that leverages single-agent RL method to find performance deficiencies in the quantifier-free floating-point and quantifier free-string logics. Specifically, the original algorithm learns how to make a *fuzzing mutation* (i.e., a minimal modification to an input seed). The key differences between their tool and the one presented here are the following: first, the current version of BanditFuzz uses a multi-agent RL method that avoids getting stuck in local minima that afflicted their tool, and second, the current version supports the entirety of SMT-LIB, whereas the previous one only supported floating-point and strings.

4   Implementation and Engineering

In this section, we discuss some of the implementation and the engineering of BanditFuzz. The BanditFuzz tool is written in Python3 and contains 5,000 lines of code. BanditFuzz is lightweight with minimal dependencies and can be installed in seconds.

Input/Output: Reference/Target solvers are provided to BanditFuzz as paths to an executable file which acts as an interface to the solver. These executable files will run internally within BanditFuzz during its main runtime loop. The generator constraints are command-line arguments bounding formula sizes with several theory specific constraints (e.g., bit-width, UF arity, etc.). The output of BanditFuzz is a directory of benchmarks with timing and memory analysis. A single run BanditFuzz will produce a single benchmark. To build a benchmark suite, BanditFuzz can safely be run in parallel and tested on several major cloud computing environments (e.g., AWS). We provide an interface that allows for this to be done via the command line.

Generator: The generator module is responsible for producing well-formed SMT-LIB inputs. Internally, BanditFuzz an Abstract Syntax Tree (AST) data structure to represent a benchmark. Each AST is asserted with root nodes of a boolean sort and positive arity. Each AST is populated by randomly sampling from the set constructs of the required sort with leaf nodes of variables or theory literals. All ASTs are full with respect to the maximum depth specified by the user.

Mutator: A mutator is a python method that takes as input a benchmark and a grammatical construct. The output is a perturbation of the input benchmark that contains a novel occurrence of the input construct. The mutator works by constructing the set nodes of the input construct's sort and then uniformly at

random replaces the selected construct with the input construct. The mutator then applies a procedure to ensure the resulting benchmark remains well-formed as the node replacement may result in an arity change. This is done by deleting or generating new subtrees that are consistent with the generator constraints. Like the generator, the resulting AST from the mutator is full with respect to the maximum depth.

Agents: We use a context-free MAB approach in this paper for our agent, namely Thompson Sampling. However, in principle, this can be lifted to several more RL paradigms. Our agent implementation is lightweight and makes a single external API call (i.e., the NumPy beta distribution sampling method [21]). Furthermore, unlike several RL paradigms, our MAB solution only has a single hyper-parameter, the exponential decay of the observed empirical mean.

Performance Analyzer: We include a performance analyzer to monitor the subprocesses' resource consumption (e.g., wall-clock runtimes and memory). Processes are killed if they violate the user's constraints. When calling solvers, target solvers are run first under the provided constraints. Afterward, reference solvers are ran using a dynamic timeout scheme based best-observed performance margin. This prevents wasting time on the reference solvers when it is no longer possible for the current benchmark to have a higher margin than the best observed.

5   Using the BanditFuzz Fuzzing Framework

In this section, we demonstrate how to use BanditFuzz. The BanditFuzz package has two core tools:

- `smtfuzz` – A fuzzer (i.e., a tool that generates inputs to a program-under-test) for all SMT-LIB theories. In principle, this fuzzer can be used in any fuzzing context, but in this paper it is the core fuzzer in BanditFuzz's fuzzing algorithm.
- `banditfuzz` – An implementation of the BanditFuzz performance fuzzing algorithm. This program calls `smtfuzz` in a loop and inherits all of its command line arguments.

5.1   Using `smtfuzz`

The `smtfuzz` tool generates random Abstract Syntax Trees (ASTs) based on the enabled theories. `smtfuzz` is designed to be extremely flexible. Users can modify the problem size easily by setting the `--num-asserts` and `--depth` parameters to increase the number of assertions and size of each assertion respectively. The generator in `smtfuzz` supports all core theories in the SMT-LIB initiative [4]. Each theory can be enabled by setting its respective flag. `smtfuzz` will automatically set the problem's logic based on the enabled theories.

Argument	Description
`--num-asserts`	Set the number of assertions in the generated benchmark
`--depth`	Set the depth of each asserting AST
`--num-vars`	The number of theory variables
`-q --quantifiers`	Enable quantifiers
`-a --arrays`	Enable arrays
`-uf --uninterpreted-functions`	Enable uninterpreted functions
`-str --strings`	Enable strings
`-fp --floating-point`	Enable floating-point
`-bv --bit-vectors`	Enable bit-vectors
`-int --integer`	Enable integers
`-r --real`	Enable reals
`-8 -16 -32 -64 -128 -256`	Bit width for bit-vectors and floating-point arithmetic
`-l --linear`	Enforce integer and real constraints to be linear

Table 1: Sample of generator arguments for the BanditFuzz tool

```
$ smtfuzz -qf -bv -fp - uf --num-asserts 1 --num-vars 1 --num-ufs 1
(set-logic QF_UFBVFP)
(declare-const bool_0 Bool)
(declare-const fp_0 (_ FloatingPoint 8 24))
(declare-const bv_0 (_ BitVec 32))
(declare-fun uf_0 (Bool (_ BitVec 32) Bool Bool (_ FloatingPoint 8 24)) Bool)
(assert (uf_0 (fp.isPositive (fp.roundToIntegral RTZ fp_0)) (bvsub (bvsmod bv_0
#x2ad75270 ) (bvxnor bv_0 #x3a990975 )) (fp.isNaN (fp.abs fp_0)) (bvuge (bvor bv_0
bv_0) (bvnor #x0a1b63c9  #x52911167 )) (fp.roundToIntegral RTN (fp.neg (fp #b1
#b11110100 #b11000100101000110101000)))))
(check-sat)
(exit)
```

Fig. 2 *Example usage of `smtfuzz` to generate a benchmark in the logic of QF_UFBVFP*

5.2 Using `banditfuzz`

The `banditfuzz` script is an implementation of the BanditFuzz algorithm and uses `smtfuzz` as its primary fuzzer. Furthermore, it has four key additional arguments:

1. `--target-solvers` – The set of executables that BanditFuzz will try to expose relative performance deficiencies on.
2. `--reference-solvers` – The set of reference executables that BanditFuzz will try to expose relative performance deficiencies with respect to.
3. `--query-timeout` – This parameter is the wallclock timeout of each query of a solver on an input benchmark.
4. `--global-timeout` – This parameter is the global timeout of `banditfuzz`. When this time is met, `banditfuzz` will return the benchmark that had the

highest performance margin between the target solvers and the reference solvers.

6 Empirical Evaluation

In this section, we present an evaluation of BanditFuzz vs. standard performance fuzzing algorithms.

6.1 Experimental Setup

Experimental Objective: Here we describe our evaluation of BanditFuzz against random fuzzing and previous similar work by Scott et al. [42]. The objective of the experiment is as follows: given the same amount of resources, which of the three tools maximizes the performance margin for a given target solver vs. a set of reference solvers over all the 52 logics used in SMT-COMP '20. For target solvers, we choose the most performant solvers from SMT-COMP '20 competition, and as reference solvers we used the runner-up solver(s) from the same track in the competition. In the case where a solver was not able to run in our setup, often due to environmental hard-codings, we replace it with the next most performant alternative.
Baselines: As baselines, we use random fuzzing (i.e., `smtfuzz` from Section 5 in a loop) and the original performance fuzzer by Scott et al. [42] when possible since it is limited to floating-point and string logics. While there are many other fuzzers for SMT Solvers [10,49,48,34], they are mostly aimed at finding errors and not performance issues.

Other general purpose fuzzers like AFL [52] and PerfFuzz [27] are built around bit-string manipulation. We attempted to use these tools but, as we suspected, neither were able to produce a well-formed input given significant amounts of resources. Unfortunately, it is known that general purpose bit-string fuzzers do not to scale to programs with strict grammars like SMT Solvers, despite the fact that AFL has some capacity to add custom grammar [51].
Computational Environment: All experiments were performed on the Compute Canada computing service [2], a CentOS V7 cluster of Intel Xeon Processor E5-2683 running at 2.10 GHz with 8 GB of memory. Wallclock runtimes are rounded to the nearest second.
Generator: Fuzzers were set to generate benchmarks with 5 variables per sort, 5 assertions, and a maximum depth of 3 in logics that were neither just linear, just arrays, bit-vectors, nor just uninterpreted functions. Otherwise 10 variables, 10 assertions, and depth 5 was used. We use bit-widths of 64.

6.2 Results

Using the aforementioned experimental setup, we evaluated BanditFuzz against random fuzzing across 52 logics from the SMT-COMP '20. Tables 2, 3 summarizes our experimentation against random fuzzing across all 52. The first column

in these tables denote the logic of the experiment, the second and third column denote the solver that was targeted and referenced respectively. The fourth and fifth column denotes the cumulative `PAR-2` margin across 25 runs of random fuzzing and BanditFuzz respectively. The sixth column reports improvement of BanditFuzz over random fuzzing based on the absolute difference between their `PAR-2` margins. We observe BanditFuzz to consistently outperform random fuzzing across all 52, with up to a 82.6% improvement in `PAR-2` margin in the `UFLIA` logic.

To visually illustrate the benchmark testing suites generated by BanditFuzz, we include a cactus plot on the highly industrial logic of QF_BV in Figure 3. A cactus plot is a visualization of a solver's performance on a benchmark suite the X-axis represents the number of benchmarks solved and the Y-axis represents time (in seconds) taken per benchmark. Every benchmark is the resultant of run a complete run of the tool. In SMT-COMP '20, the SMT Solver Bitwuzla had a strong performance, winning numerous gold medals including the QF_BV track over competing solvers CVC4, Z3, MathSAT, and Yices. However, the cactus plot clearly shows that Bitwuzla is least performant on the benchmarks produced by BanditFuzz by an extremely large margin.

In Table 4, we further compare against previous work by Scott et al. [42]. We baseline BanditFuzz against this work on the only two logics it supports, QF_S and QF_FP. We observe that BanditFuzz consistently outperforms against the baseline and achieves a maximum possible score in both logics, while the baseline fails to do so.

7   Case Studies with Solver Developers

In this section, we provide some case studies of BanditFuzz and how it enabled developers to find surprising performance deficiencies in state-of-the-art SMT solvers.

7.1   CVC4, Bitwuzla, and SymFPU

We contacted the developers of CVC4, Bitwuzla, and the SymFPU bit-blaster [9] for floating-point problems. While CVC4 and Bitwuzla have significantly different underlying bit-vector engines, they both utilize the SymFPU tool for bit-blasting floating-point operations. To this end, we proposed an experiment where we target both CVC4 and Bitwuzla (the target solvers) against Z3 (the reference).

The resulting benchmarks showcase performance issues in the SymFPU bit-blaster and possibly the CVC4 and Bitwuzla solvers themselves. We ran an analogous experiment to what was described in Section 6, with a 2400 second wallclock timeout over a 24 hour period. BanditFuzz produced 25 benchmarks that significantly separated Bitwuzla and CVC4 from Z3 on the logic of `QF_FP`. On these benchmarks that BanditFuzz produced, Z3 had a `PAR-2` score of 3,018

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on Baseline [%]
			Random	BanditFuzz	
ABVFP	CVC4	Z3	2,716	5,579	105
ABVFPLRA	CVC4	Z3	21,376	60,000	181
ALIA	CVC4	Z3	4,238	11,340	168
ANIA	CVC4	Z3	34,883	60,000	72
AUFLIA	CVC4	Z3	34,229	60,000	75
AUFLIRA	CVC4	Z3	7,650	30,428	298
AUFNIA	CVC4	Z3	279	753	170
AUFNIRA	CVC4	Z3	7,967	16,949	113
BV	CVC4	Z3	50,561	60,000	19
BVFP	CVC4	Z3	319	758	138
BVFPLRA	CVC4	Z3	50,844	60,000	18
FP	CVC4	Z3	3,700	10,674	188
FPLRA	CVC4	Z3	15,325	49,528	223
LIA	CVC4	Z3	5,635	19,050	238
LRA	CVC4	Z3	11,184	25,560	129
NIA	CVC4	Z3	48,752	60,000	23
NRA	CVC4	Z3	27,066	60,000	122
QF_ABV	Bitwuzla	Yices2	16,280	45,814	181
QF_ABVFP	Bitwuzla	CVC4	48,484	60,000	24
QF_ABVFPLRA	CVC4	COLIBRI	1,652	4,431	168
QF_ALIA	Yices2	Z3	17,670	60,000	240
QF_ANIA	CVC4	Z3	34,444	60,000	74
QF_AUFBV	Yices2	Bitwuzla	5,375	14,704	174
QF_AUFLIA	Yices2	CVC4	21,836	56,345	158
QF_AUFNIA	CVC4	Z3	35,817	60,000	68
QF_AX	Yices2	CVC4	3,251	5,153	59

Table 2: Table of results comparing BanditFuzz to Random fuzzing across logics of SMT-COMP '20. The improvement column is the percentage improvement of BanditFuzz over Random Fuzzing. Rows are sorted alphabetically by logic.

seconds, while CVC4 and Bitwuzla had 91,408 seconds and 120,000 seconds respectively [3].

In discussions with Aina Niemetz and Mathias Preiner, members of CVC4 and Bitwuzla teams: "In general, the benchmarks produced by BanditFuzz can be super helpful for us to figure out what's missing in our solvers". For example, in their Bitwuzla tool, BanditFuzz found several benchmarks where the rewrite level (-rwl) was configured to be too high. Furthermore, Martin Brain, the author of SymFPU, said: "BanditFuzz is interesting because it gives us an abundant supply of something valuable but previously very rare; small benchmarks with significant performance differentials."

[3] Bitwuzla timed out on all benchmarks produced by BanditFuzz

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on
			Random	BanditFuzz	Baseline [%]
QF_BV	Bitwuzla	CVC4	22,142	52,681	138
QF_BVFP	Bitwuzla	CVC4	29,949	60,000	100
QF_BVFPLRA	CVC4	COLIBRI	23,053	55,228	140
QF_FP	Bitwuzla	COLIBRI	37,692	60,000	59
QF_FPLRA	COLIBRI	CVC4	2,030	4,053	100
QF_LIA	CVC4	Yices2	3,217	5,399	68
QF_LIRA	Yices2	CVC4	1,795	7,584	323
QF_LRA	CVC4	Yices2	4,571	14,184	210
QF_NIA	CVC4	Yices2	32,540	60,000	84
QF_NIRA	CVC4	Yices2	8,348	32,509	289
QF_NRA	Yices2	CVC4	22,861	60,000	162
QF_S	CVC4	Z3str4	35172	60,000	71
QF_SLIA	CVC4	Z3str4	3,956	15,381	289
QF_UF	Yices2	Z3	5,607	15,362	174
QF_UFBV	Yices2	Bitwuzla	34,315	60,000	75
QF_UFFP	Bitwuzla	COLIBRI	12,909	20,373	58
QF_UFLIA	Yices2	CVC4	1,696	2,428	43
QF_UFLRA	Yices2	Z3	8,431	26,529	215
QF_UFNIA	CVC4	Yices2	3,864	13,564	251
QF_UFNRA	Yices2	CVC4	53,374	60,000	12
UF	CVC4	Z3	3,469	13,368	285
UFBV	CVC4	Z3	37,751	60,000	59
UFLIA	CVC4	Z3	174	868	399
UFLRA	CVC4	Z3	1,567	5,159	229
UFNIA	CVC4	Z3	5,671	17,419	207
UFNRA	CVC4	Z3	17,219	60,000	248

Table 3: Table of results comparing BanditFuzz to Random fuzzing across logics of SMT-COMP '20. The improvement column is the percentage improvement of BanditFuzz over Random Fuzzing. Rows are sorted alphabetically by logic.

7.2   Z3 String Solver

We also released the BanditFuzz tool to the developers of the Z3str4 string solver [5,6], so that they could independently use it to expose performance issues in their solver. The Z3str4 team used BanditFuzz to find performance deficiencies in experimental builds of their solver, namely Z3str4-ACF and Z3str4-NCF (the target solvers) against CVC4 and Z3seq [15]. They were able to produce thousands of benchmarks demonstrating performance separations. Mitja Kulczynski, one of the authors of Z3str4, observed: "BanditFuzz is extremely easy to use! When targeting Z3str4-NCF, BanditFuzz was able to find benchmarks in the form of disjunctions over substring operations. While this issue was already known to us, BanditFuzz provided us with a benchmark suite to improve our tool. Furthermore, when targeting Z3str4-ACF, BanditFuzz found a class
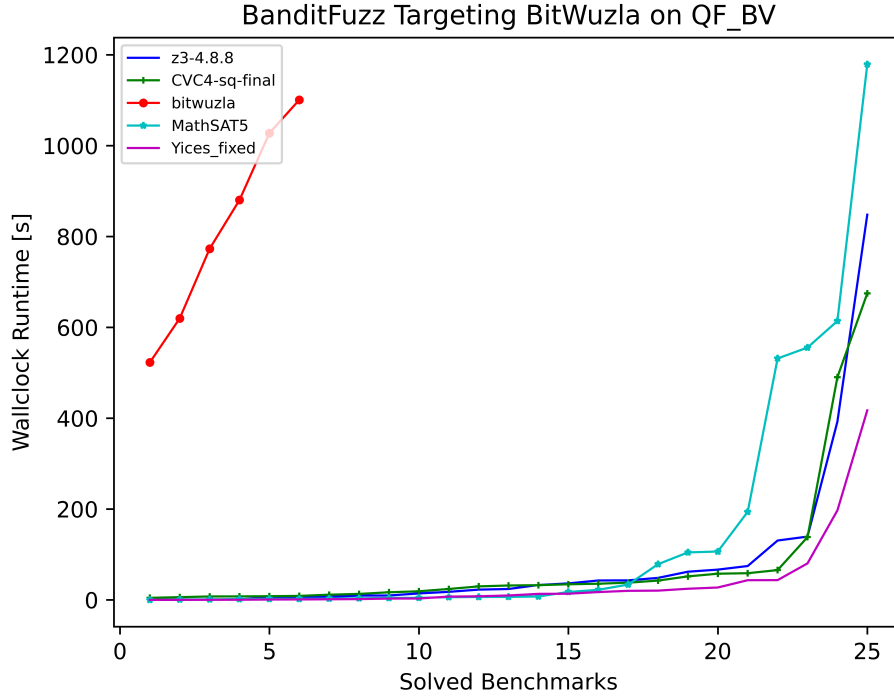
Fig. 3 *Cactus plot for targeting Bitwuzla (winner of SMT-COMP '20 in the QF_BV division) against reference runner-up solvers that competed in the division. The X-axis represents the number of benchmarks solved and the Y-axis represents time (in seconds) taken.*

of benchmarks of conjunctions of `str.at` where the solver was extremely slow. This was completely unknown to us!"

8 Threats to Validity

One major drawback of BanditFuzz and many other blackbox fuzzing approach is the inability to determine how many unique issues have been discovered. For example, in recent work by Winterer et al. [49] thousands of soundness bugs were reported, but upon inspection the total number of resolved issues was less. However, in our discussions with solver developers (Section 7), it was reported to us the benchmarks returned by BanditFuzz were not all of a common cause. In fact, in both case studies, while several of the discovered performance issues by BanditFuzz have been discussed and fixed by developers, several still have open issues.

With the recent advances in reinforcement learning, it is natural to wonder if the idea of BanditFuzz can be lifted to more powerful algorithms (e.g., deep

Q learning, actor critic, etc.). While these model-based techniques can be very powerful, relative to MAB algorithms, they require significantly more data to train effective agents (i.e., tens of thousands of steps on simple environments). As BanditFuzz aspires to generate benchmarks that take a significant amount of time to solve, a single environmental step can take several minutes.

9   Related Work

The work that is most similar to this paper is by Scott et al. [42]. In Sections 1, 3, 6 of this paper we highlight the novel contributions on of this work. Specifically, our tool uses a multi-agent RL method over the single-agent by Scott et al., and hence has a lower propensity to get stuck in local minima. Additionally, we support all of SMT-LIB, while their tool only supports floating point and strings. Another closely related tool is PerfFuzz which is a bit-string performance fuzzer. However, PerfFuzz is not grammar-aware and hence is unlikely to produce well-formed SMT formulas.

Fuzzing and Fuzzing SMT Solvers: Software fuzzing is a large field of research, and we refer to the survey by Manes et al. as a basis for the current research [28]. There are tools and fuzzers for finding bugs in specific SMT theories [35,7,11,10,30,30].

Machine Learning for Fuzzing: Bottinger et al. [8] introduce a deep Q learning algorithm for fuzzing model-free inputs. Godefroid et al. [18] use neural networks to learn an input grammar over complicated domains such as PDF and then use the learned grammar for model-guided fuzzing. Woo et al. [50] and Patil et al. [36] used MAB algorithms to select configurations of global hyperparameters of fuzzing software. Rebert et al. [38] used MABs to select from a list of valid inputs seeds to fuzz on.

Machine Learning and SMT Solvers: Other works have leveraged machine learning to learn models relating to SMT solving performance. Healy et al. leveraged supervised learning for analyzing SMT solver performance in the context of software verification [22]. The MachSMT solver leverages machine learning SMT Solver algorithm selection [43] and the MelodySolver leverages reinforcement learning for online algorithm selection [37].

10   Conclusions

In this paper, we present BanditFuzz, a performance fuzzer for SMT Solvers. BanditFuzz is the first multi-agent RL-based performance fuzzer to support all of SMT-LIB and leverages reinforcement learning to find relative performance deficiencies in state-of-the-art SMT Solvers. We evaluated BanditFuzz across

---

³ The previous code framework by Scott et al. [42] only supports two logics QF_FP and QF_S. When evaluating outside of these logics, we baseline by using the BanditFuzz code base with the second agent disabled.

Logic	Target	Reference	PAR-2 Performance Margin		Improvement on
			Scott et al. [42]	BanditFuzz	Baseline [%]
QF_FP	Bitwuzla	COLIBRI	51,893	60,000	14.4
QF_S	CVC4	Z3str4	53,231	60,000	11.9
ABVFPLRA	CVC4	Z3	46,237	60,000	25.9
ANIA	CVC4	Z3	54,120	60,000	10.3
AUFLIRA	CVC4	Z3	22,314	30,428	30.7
FP	CVC4	Z3	9,109	10,674	15.8
FPLRA	CVC4	Z3	31,808	49,528	43.5
LIA	CVC4	Z3	17,009	19,050	11.3
LRA	CVC4	Z3	16,098	25,560	45.4
NRA	CVC4	Z3	39,116	60,000	42.1
QF_ALIA	Yices2	Z3	35,912	60,000	50.2
QF_ANIA	CVC4	Z3	56,198	60,000	6.5
QF_AUFBV	Yices2	Bitwuzla	11,103	14,704	27.9
QF_BVFPLRA	CVC4	COLIBRI	47,180	55,228	15.7
QF_LIRA	Yices2	CVC4	3,152	7,584	82.6
QF_NIA	CVC4	Yices2	58,199	60,000	3.0
QF_NIRA	CVC4	Yices2	17,009	32,509	62.6
QF_NRA	Yices2	CVC4	42,188	60,000	34.9
QF_UFLRA	Yices2	Z3	22,092	26,529	18.3
QF_UFNIA	CVC4	Yices2	9,917	13,564	31.1
UFNIA	CVC4	Z3	14,282	17,419	19.8
UFNRA	CVC4	Z3	27,901	60,000	73.0

Table 4: Table of select results comparing BanditFuzz to the work of Scott et al. [42] across select logics. The improvement column is the percentage improvement of BanditFuzz over the baseline.[4]

52 logics from SMT-COMP '20 targeting competition-winning solvers against runner-up solvers. We compare BanditFuzz against random fuzzing and a single-agent tool with up to a 82.6% improvement in the margin of `PAR-2` score on the `UFLIA` logic. We further provide several case studies demonstrating the utility of BanditFuzz to state-of-the-art SMT solver developers.

11   Acknowledgements

References

1. Armando, A., Mantovani, J., Platania, L.: Bounded model checking of software using SMT solvers instead of SAT solvers. International Journal on Software Tools

for Technology Transfer **11**(1), 69–83 (2009)

2. Baldwin, S.: Compute Canada: Advancing Computational Research. In: Journal of Physics: Conference Series. vol. 341, p. 012001. IOP Publishing (2012)

3. 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

4. Barrett, C., Fontaine, P., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org (2016)

5. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: a String Solver with Theory-Aware Heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59. IEEE (2017)

6. Berzish, M., Mora, F., Kulczynski, M., Nowotka, D., Ganesh, V.: Z3str4 String Solver: System Description. SMT-COMP 2020 (2020)

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. Böttinger, K., Godefroid, P., Singh, R.: Deep reinforcement fuzzing. arXiv preprint arXiv:1801.04589 (2018)

9. Brain, M., Schanda, F., Sun, Y.: Building better bit-blasting for floating-point problems. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 79–98. Springer (2019)

10. 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)

11. Bugariu, A., Müller, P.: Automatically Testing String Solvers. In: International Conference on Software Engineering (ICSE), 2020. ETH Zurich (2020)

12. 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)

13. Calzavara, S., Grishchenko, I., Maffei, M.: Horndroid: Practical and sound static analysis of android applications by smt solving. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 47–62. IEEE (2016)

14. Cordeiro, L., Fischer, B., Marques-Silva, J.: Smt-based bounded model checking for embedded ansi-c software. IEEE Transactions on Software Engineering **38**(4), 957–974 (2011)

15. 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)

16. Duchene, F.: Fuzz in the dark: Genetic algorithm for black-box fuzzing. In: Black-Hat (2013)

17. Gerlich, R., Prause, C.R.: Optimizing the parameters of an evolutionary algorithm for fuzzing and test data generation. In: 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 338–345. IEEE (2020)

18. 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)

19. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. ACM SIGPLAN Notices **43**(6), 281–292 (2008)

20. Gupta, A.K., Nadarajah, S.: Handbook of beta distribution and its applications. CRC press (2004)

21. Harris, C.R., Millman, K.J., van der Walt, S.J., Gommers, R., Virtanen, P., Cournapeau, D., Wieser, E., Taylor, J., Berg, S., Smith, N.J., Kern, R., Picus, M., Hoyer, S., van Kerkwijk, M.H., Brett, M., Haldane, A., del Río, J.F., Wiebe, M., Peterson, P., Gérard-Marchant, P., Sheppard, K., Reddy, T., Weckesser, W., Abbasi, H., Gohlke, C., Oliphant, T.E.: Array programming with NumPy. Nature **585**(7825), 357–362 (Sep 2020). https://doi.org/10.1038/s41586-020-2649-2, https://doi.org/10.1038/s41586-020-2649-2

22. Healy, A., Monahan, R., Power, J.F.: Predicting SMT solver performance for software verification. In: Dubois, C., Masci, P., Méry, D. (eds.) Proceedings of the Third Workshop on Formal Integrated Development Environment, F-IDE@FM 2016, Limassol, Cyprus, November 8, 2016. EPTCS, vol. 240, pp. 20–37 (2016). https://doi.org/10.4204/EPTCS.240.2, https://doi.org/10.4204/EPTCS.240.2

23. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. ACM SIGPLAN Notices **47**(6), 77–88 (2012)

24. Junker, M., Huuck, R., Fehnker, A., Knapp, A.: Smt-based false positive elimination in static program analysis. In: International Conference on Formal Engineering Methods. pp. 316–331. Springer (2012)

25. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification. pp. 97–117. Springer (2017)

26. Komuravelli, A., Gurfinkel, A., Chaki, S.: Smt-based model checking for recursive programs. Formal Methods in System Design **48**(3), 175–205 (2016)

27. 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)

28. 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)

29. Manès, V.J., Kim, S., Cha, S.K.: Ankou: guiding grey-box fuzzing towards combinatorial difference. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1024–1036 (2020)

30. 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)

31. Marijn Heule, Matti Järvisalo, M.S.: Sat race 2019 (2019), http://sat-race-2019.ciirc.cvut.cz/

32. Niemetz, A., Preiner, M.: Bitwuzla at the SMT-COMP 2020. CoRR **abs/2006.01621** (2020), https://arxiv.org/abs/2006.01621

33. Niemetz, A., Preiner, M.: Ternary propagation-based local search for more bit-precise reasoning. In: 2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020. pp. 214–224. IEEE (2020). https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29, https://doi.org/10.34727/2020/isbn.978-3-85448-042-6_29

34. Niemetz, A., Preiner, M., Biere, A.: Model-based API testing for smt solvers. In: Proceedings of the 15th International Workshop on Satisfiability Modulo Theories, SMT. pp. 24–28 (2017)

35. 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)
36. Patil, K., Kanade, A.: Greybox fuzzing as a contextual bandits problem. arXiv preprint arXiv:1806.03806 (2018)
37. Pimpalkhare, N., Mora, F., Polgreen, E., Seshia, S.A.: Medleysolver: Online SMT algorithm selection. In: Li, C., Manyà, F. (eds.) Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings. Lecture Notes in Computer Science, vol. 12831, pp. 453–470. Springer (2021). https://doi.org/10.1007/978-3-030-80223-3_31, https://doi.org/10.1007/978-3-030-80223-3_31
38. 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)
39. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.: Counterexample-guided quantifier instantiation for synthesis in smt. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 198–216. Springer International Publishing, Cham (2015)
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. Saavedra, G.J., Rodhouse, K.N., Dunlavy, D.M., Kegelmeyer, P.W.: A review of machine learning applications in fuzzing. arXiv preprint arXiv:1906.11133 (2019)
42. Scott, J., Mora, F., Ganesh, V.: BanditFuzz: Fuzzing SMT Solvers with Reinforcement Learning. UWSpace. http://hdl.handle.net/10012/15753 (2020)
43. Scott, J., Niemetz, A., Preiner, M., Nejati, S., Ganesh, V.: Machsmt: A machine learning-based algorithm selector for SMT solvers. In: Groote, J.F., Larsen, K.G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part II. Lecture Notes in Computer Science, vol. 12652, pp. 303–325. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_16, https://doi.org/10.1007/978-3-030-72013-1_16
44. Srivastava, S., Gulwani, S., Foster, J.S.: From program verification to program synthesis. In: Proceedings of the 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 313–326 (2010)
45. Sutton, M., Greene, A., Amini, P.: Fuzzing: brute force vulnerability discovery. Pearson Education (2007)
46. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT press (2018)
47. Takanen, A., Demott, J.D., Miller, C.: Fuzzing for software security testing and quality assurance. Artech House (2008)
48. Winterer, D., Zhang, C., Su, Z.: On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. Proceedings of the ACM on Programming Languages 4(OOPSLA), 1–25 (2020)
49. Winterer, D., Zhang, C., Su, Z.: Validating SMT solvers via semantic fusion. In: PLDI. pp. 718–730 (2020)
50. 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)
51. Zalewski, M.: afl-fuzz: making up grammar with a dictionary in hand (2015), https://lcamtuf.blogspot.com/2015/01/afl-fuzz-making-up-grammar-with.html

52. Zalewski, M.: American Fuzzing Lop (2015)