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

Joseph Scott, Federico Mora, and Vijay Ganesh

July 6th, 2020

Outline

1. Context and Motivation
2. BanditFuzz Overview
3. Empirical Evaluation
4. Conclusions and Future Work

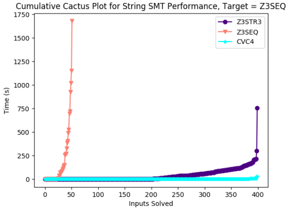
Outline

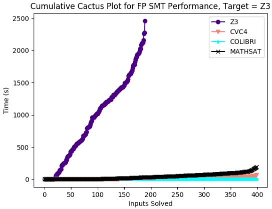
Motivation for BanditFuzz

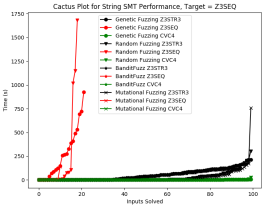
1. SMT solvers are integral tools in
    1. Program Analysis
    2. Testing
    3. Verification
2. SMT solvers are remarkably efficient tools
3. Having said that, there exists inputs such that one solver may be remarkably slower than others, despite very similar algorithms
4. How can we find these performance issues automatically?

BanditFuzz

1. In this talk, I introduce BanditFuzz, a performance fuzzer for SMT solvers
2. Relative performance fuzzing
    1. Given a target solver
    2. A set of reference solvers
    3. Find inputs that maximize the performance margin
3. BanditFuzz uses reinforcement learning, specifically multi-armed bandits (MABs), using a feedback loop between the fuzzer and programs-under-test







What is a fuzzer?

A fuzzer is a software testing tool that generates inputs for a program-under-test.

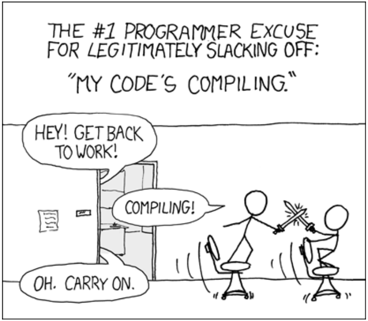
The generated inputs can be used to expose:

1. Errors
2. Performance slowdowns

Core fuzzing techniques:

1. Input generators
2. Input mutators

Fuzzers typically implement these via random/fixed strategies that are oblivious to online feedback from the programs under test. How can we do better?



Credit: xkcd #303

What is a fuzzer?

A fuzzer is a software testing tool that generates inputs for a program-under-test.

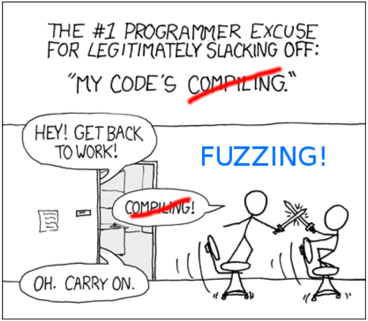
The generated inputs can be used to expose:

1. Errors
2. Performance slowdowns

Core fuzzing techniques:

1. Input generators
2. Input mutators

Fuzzers typically implement these via random/fixed strategies that are oblivious to online feedback from the programs under test. How can we do better?



Credit: xkcd #303 + Gary Kwong

What is a fuzzer?

A fuzzer is a software testing tool that generates inputs for a program-under-test.

The generated inputs can be used to expose:

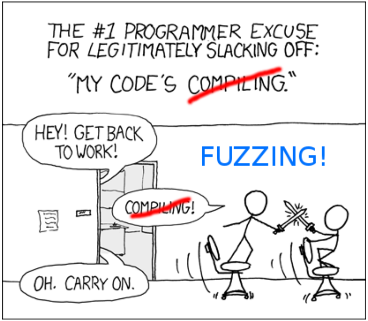
1. Errors
2. Performance slowdowns

Core fuzzing techniques:

1. Input generators
2. Input mutators

Fuzzers typically implement these via random/fixed strategies that are oblivious to online feedback from the programs under test. How can we do better?

Reinforcement Learning



Why Reinforcement Learning for fuzzing?

Fuzzing can be blackbox (oblivious to program behavior):

1. + lightweight
2. - low quality – small performance margin

Fuzzing can be whitebox (leverage program analysis)

1. - costly
2. + high quality – large performance margin

Problem: Can we develop performance fuzzers that are both lightweight and generate high-quality inputs?

Why Reinforcement Learning for fuzzing?

Fuzzing can be blackbox (oblivious to program behavior):

1. + lightweight
2. - low quality – small performance margin

Fuzzing can be whitebox (leverage program analysis)

1. - costly
2. + high quality – large performance margin

Problem: Can we develop performance fuzzers that are both lightweight and generate high-quality inputs?

Solution:

1. How? Reinforcement learning
2. Exploit the input-output behavior of the program-under-test (environment) via a feedback loop with the fuzzer (agent)

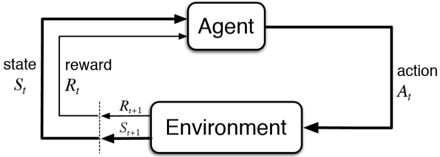
Reinforcement Learning

In reinforcement learning, an agent learns how to select actions within an environment in a way to maximize the cumulative reward.

Reinforcement learning is typically modelled through a Markov Decision Process (MDP), a 4-tuple (S, A, T, R), where:

1. S – is a set of states
2. A – is a set of actions
3. T – modelled transitions
4. R – modelled rewards

A learned agent's output is a *policy* $\pi : S \to A$, that selects the action that maximizes cumulative future reward.



Multi-Armed Bandit Problem

The Multi-Armed Bandit (MAB) problem is a stateless formulation of reinforcement learning (No states/transitions!! Just actions and rewards!).

An MAB agent must manage the *exploration/exploitation* tradeoff.

1. Select the best known action (exploitation)
2. Sample less known actions (exploration)

MAB solutions have been popularized due to their relative simplicity and success in noisy environments.

1. Online Advertisements
2. Finance
3. SAT solver branching

The name comes from a gambler at a row of slot machines (one-armed bandits), who has to decide which machines to play.

Outline

BanditFuzz – A performance fuzzer

1. BanditFuzz is a performance fuzzer that aims to find relative performance slowdowns
2. Given a target solver T
3. A set of reference solvers $R_1, R_2, R_3 ...$
4. A run of BanditFuzz seeks a input B that maximizes the performance margin

$$\mathrm{PAR2}(T, B) - \max(\mathrm{PAR2}(R_1, B), \mathrm{PAR2}(R_2, B), \mathrm{PAR2}(R_3, B), ...)$$

Fuzzer Components

1. Input Generation – BanditFuzz uses the StringFuzz approach to random abstract syntax tree (AST) generation
    1. Generate a fixed number of asserting ASTs
    2. Fill out each asserting AST in a depth-first manner
        1. Randomly select a root node from the set of predicates in the logic
        2. Randomly select non-root/leaf nodes from the set of functions/operators in the logic
        3. Fill out leaf nodes with variables
2. Mutation – Given a input B and grammatical construct γ
    1. Depending on the type of the construct (i.e., predicate, operator/function, special term), construct the set C of all constructs of the same type as γ in B, but not equal to γ.
    2. Randomly sample a construct γ' from C, and replace it with γ.
    3. On an arity increase, generate new sub-ASTs of appropriate depth.
    4. On an arity decrease, drop the right-most children.

Mutation Example

Consider a fixed depth of two, with variables (x_0, x_1), and a single rounding mode $\{RNE\}$.
Consider a single asserting AST

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

If the agent selects to mutate with *fp.add*, then we have the following possible outputs:

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

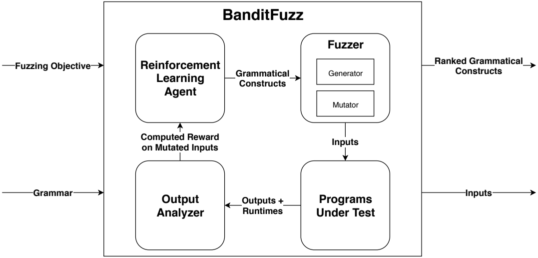
$$(fp.eq\ (fp.add\ RNE\ x_0\ x_1)(fp.abs\ x_1))$$

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

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

BanditFuzz works by reducing a fuzzing mutation to an instance of the MAB problem, and learns how to mutate inputs in a feedback loop.

1. Action Space: Grammatical Constructs of the logic.
   1. Predicates
   2. Operators and Functions
   3. Special Terms (e.g, rounding modes)
2. Rewards:
   1. 1 if the performance margin increases
   2. 0 otherwise



Thompson Sampling

Thompson Sampling is an algorithm that solves the MAB problem.

Thompson sampling presupposes that rewards are from Bernoulli distribution $\{0, 1\}$ and uses a beta distribution to model each action's expected value.

Algorithm:

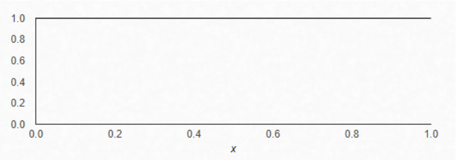
- For each action, initialize a beta distribution $\mathrm{Beta}(\alpha = 1, \beta = 1)$
- While Training
  - When queried, sample each action's beta distribution.
  - Select an action by computing an argmax over each action's sampled value.
  - On reward, increment the α parameter.
  - Otherwise, increment the β parameter.

A few iterations of BanditFuzz

Consider an action set $\{+, *\}$, and an input I_1

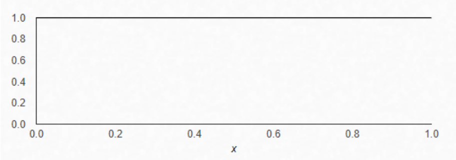
Iteration $\#1$

PDF for $+$



$\alpha = 1, \beta = 1, \mu = 0.5, \sigma = 0.2887$

PDF for *



$\alpha = 1, \beta = 1, \mu = 0.5, \sigma = 0.2887$

The agent samples both distributions, and computes an argmax.

Action $*$ is selected, then I_2 is created by adding a new occurrence of $*$ into I_1!

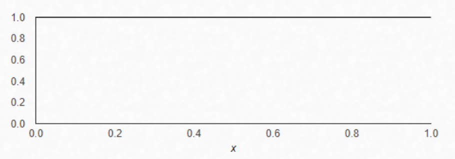
I_2 has a higher performance margin than I_1. The agent gets a reward!

A few iterations of BanditFuzz

Consider an action set $\{+, *\}$, and an input I_1

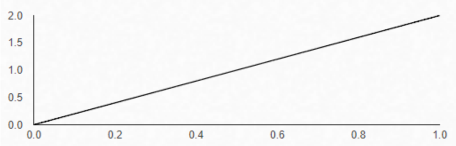
Iteration #2

PDF for $+$



$\alpha = 1, \beta = 1, \mu = 0.5, \sigma = 0.2887$

PDF for *



$\alpha = 2, \beta = 1, \mu = 0.6667, \sigma = 0.2357$

The agent samples both distributions, and computes an argmax.

Action $+$ is selected, then I_3 is created by adding a new occurrence of $+$ into I_2!

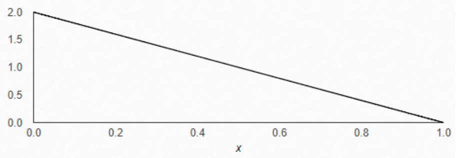
I_3 has a lower performance margin than I_2. The agent does not receive reward.

A few iterations of BanditFuzz

Consider an action set $\{+, *\}$, and an input I_1

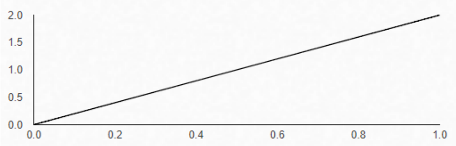
Iteration #3

PDF for +



$\alpha = 1, \beta = 2, \mu = 0.3334, \sigma = 0.2357$

PDF for *



$\alpha = 2, \beta = 1, \mu = 0.6667, \sigma = 0.2357$

The agent samples both distributions, and computes an argmax.

Action + is selected, then I_4 is created by adding a new occurrence of + into I_2!

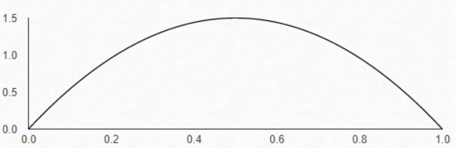
I_4 has a higher performance margin than I_2. The agent receives reward.

A few iterations of BanditFuzz

Consider an action set $\{+, *\}$, and an input I_1

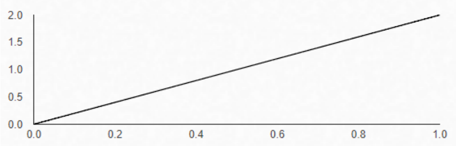
Iteration $\#4$

PDF for $+$



$\alpha = 2, \beta = 2, \mu = 0.5, \sigma = 0.2236$

PDF for *



$\alpha = 2, \beta = 1, \mu = 0.6667, \sigma = 0.2357$

The agent samples both distributions, and computes an argmax.

Action $*$ is selected, then I_5 is created by adding a new occurrence of $*$ into I_4!

I_5 has a higher performance margin than I_4. The agent receives reward.

Outline

Baselines

1. **Random Fuzzing** – Randomly generate inputs via random AST generation in a loop
2. **Random Mutation Fuzzing** – Randomly mutate the best observed input in a loop
3. **Evolutionary Fuzzing** – Maintain a population of inputs by carrying over the best observed, randomly mutating the best observed, and randomly generated new ones

Each run of a fuzzer is allocated 12 hours to find a single input that maximizes the margin between the target solver and reference solvers.

Every fuzzing configuration is ran 100 times, to produce a input suite of 100 different inputs.

Setup and Solvers

We consider the logics *QF_FP* and *QF_S*.

QF_FP

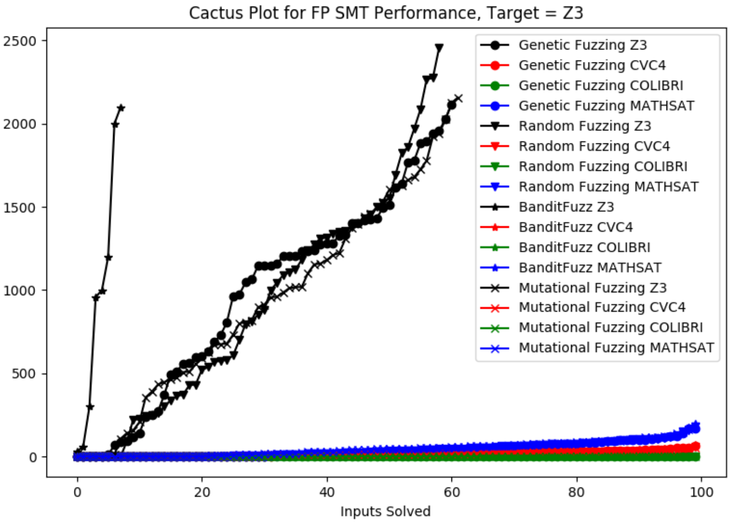
1. **Z3** v4.8.0
2. **MathSAT5** v5.5.3
3. **CVC4** CVC4 1.7-prerelease
4. **Colibri** v2070

QF_S

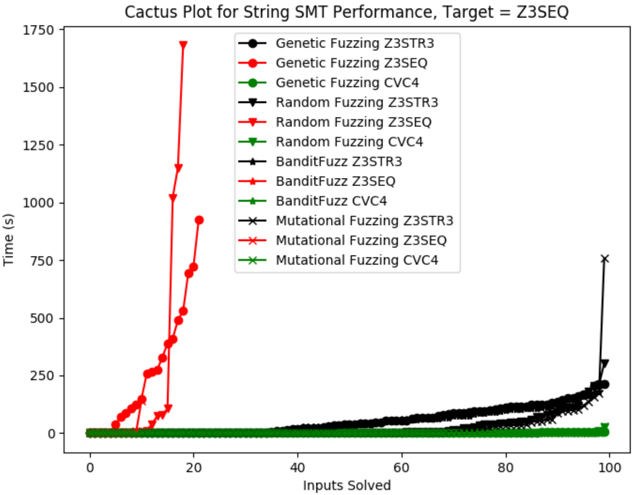
1. **Z3str3** v4.8.0
2. **Z3seq** v4.8.0
3. **CVC4** v1.6

All experiments were performed on 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.

Results: Cactus Plot – Targeting Z3 for QF_FP



Results: Cactus Plot – Targeting Z3SEQ for QF_S



Quantitative Evaluation

We will use PAR-2 to quantify the performance of a solver input pair. PAR-2 is defined as the sum of all successful runtimes, with unsolved inputs labelled as twice the timeout.

As we are fuzzing for performance with respect to a target solver T, we evaluate the returned test suite D 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 R.

$$\mathrm{PAR\text{-}2Margin}(T, R, D) := \sum_{I \in D} (\mathrm{PAR\text{-}2}(I, T) - \max_{r \in R}(\mathrm{PAR\text{-}2}(I, r)))$$

Results: PAR-2Margin tables

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

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%

Outline

Conclusions and Future Work

1. In this talk, I presented BanditFuzz a reinforcement learning performance fuzzer
2. Using BanditFuzz, we were able to generate multiple testing suites exposing significant relative performance difference, improving on considered baselines by up to 81%

Future Work:

1. Currently BanditFuzz only supports two logics, extending to all logics is work in progress
2. Repeat evaluation with new solvers

Code: `https://github.com/j29scott/BanditFuzz_Public`
Email: joseph.scott@uwaterloo.ca