RUB CASA Securing the Digital Society

Chair for Software Security

PogoFuzz: Profile-Guided Optimization for Fuzzing

Tobias Holl Leon Weiß Kevin Borgolte

Ruhr University Bochum

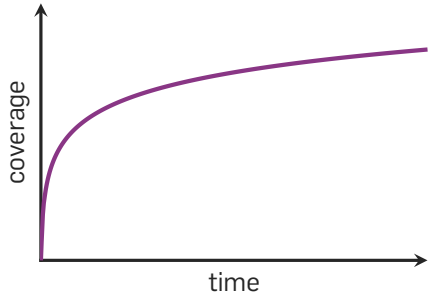
5th International Fuzzing Workshop (FUZZING) · 27 February 2026

Fuzzing at Scale

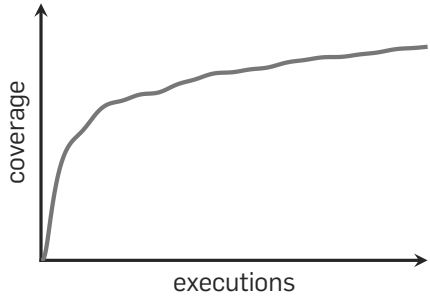
- Fuzzing campaigns benefit from scale (more compute generally means more bugs).
- OSS-Fuzz continuously fuzzes over 1300 projects at on average over 80 cores each.
- We can only guess at large companies' internal fuzzing setups.

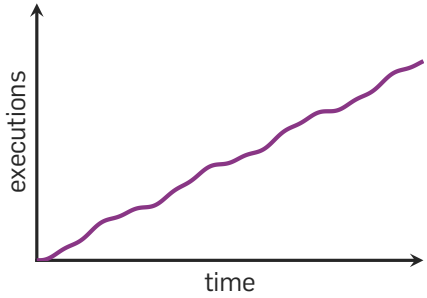
Even small improvements matter, as long as they scale.

Improving Fuzzers

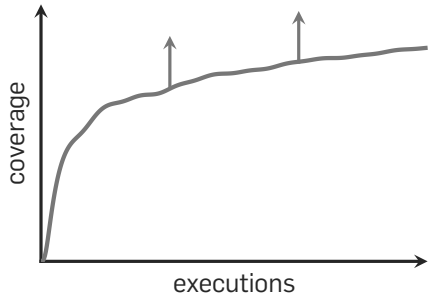


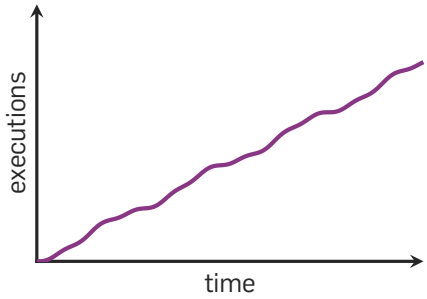
Improving Fuzzers





Improving Fuzzers

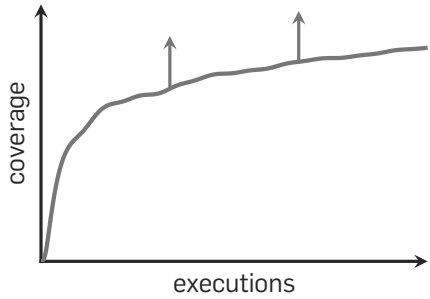




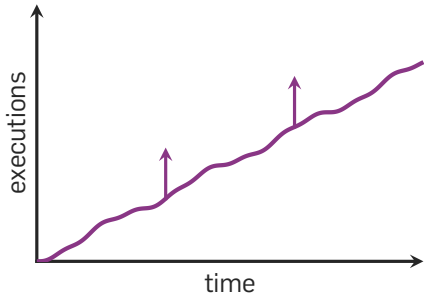
“Fuzz smarter”

(this is what most people try to do)

Improving Fuzzers



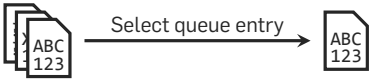
“Fuzz smarter”
(this is what most people try to do)



“Fuzz faster”
(this is what we do)

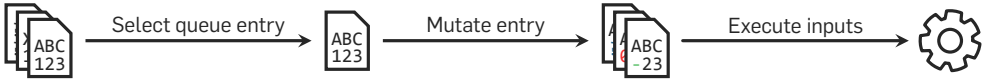
The Fuzzing Loop

The Fuzzing Loop

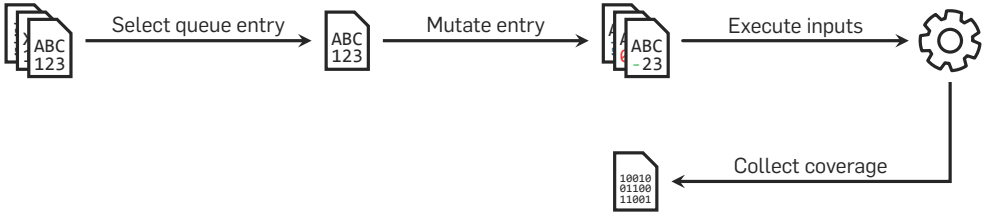


The Fuzzing Loop

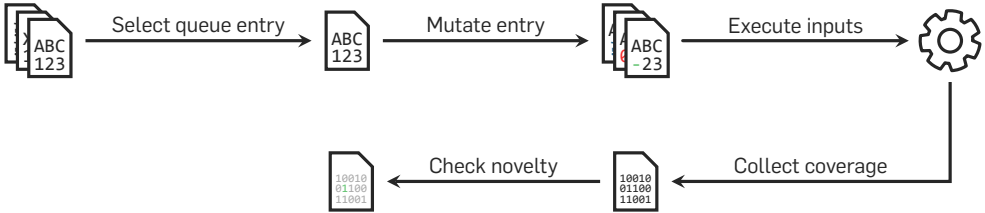
The Fuzzing Loop



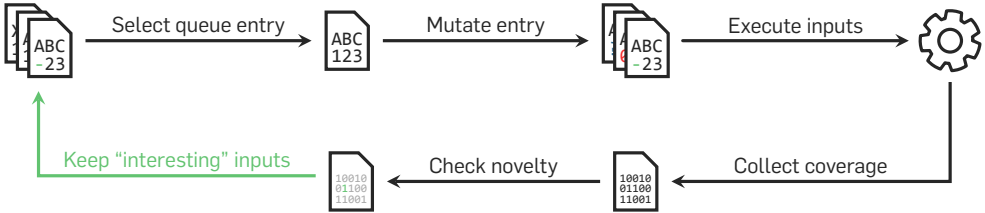
The Fuzzing Loop



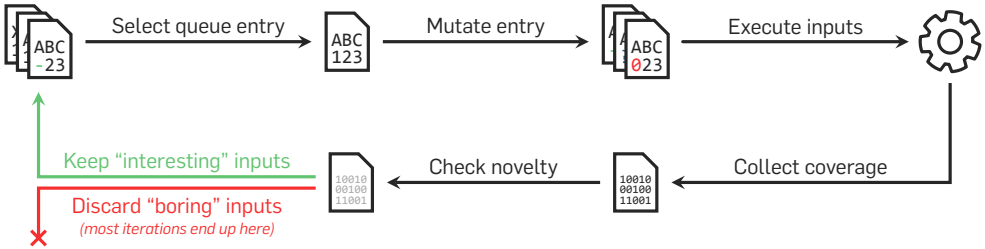
The Fuzzing Loop



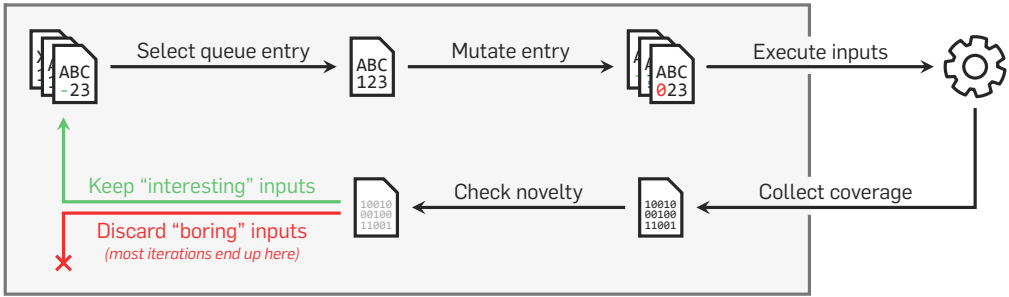
The Fuzzing Loop



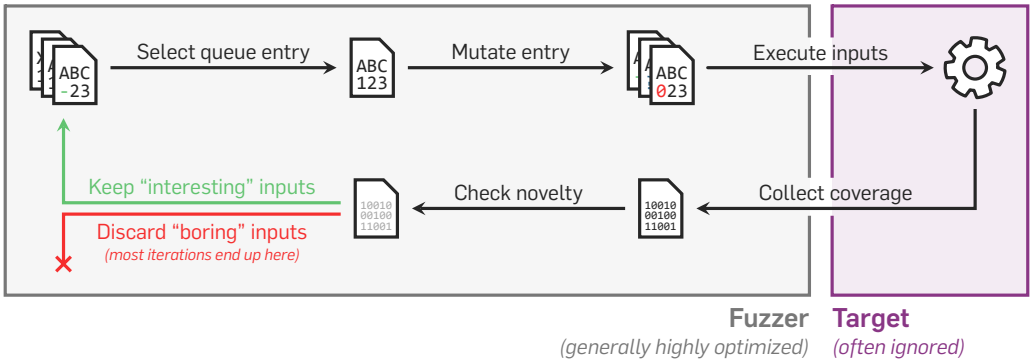
The Fuzzing Loop



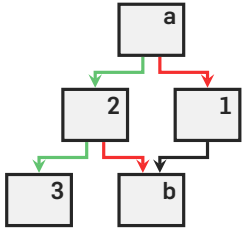
The Fuzzing Loop



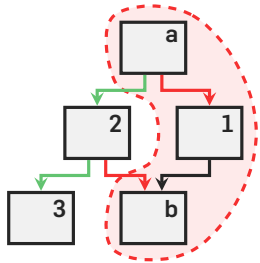
The Fuzzing Loop



Profile-Guided Optimization

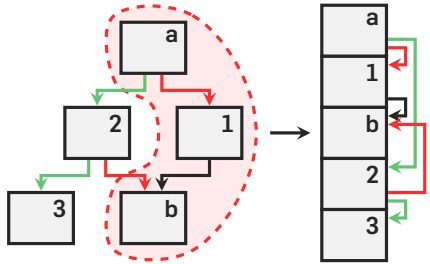


Profile-Guided Optimization



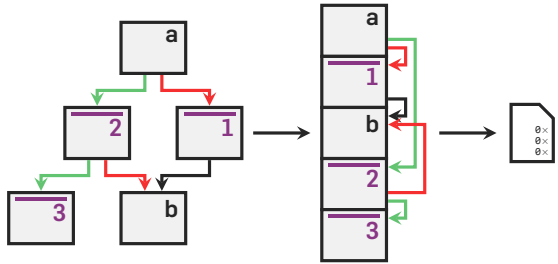
Compilers make **educated guesses** about how a program is going to be used in order to make optimization decisions, but those guesses can be wrong.

Profile-Guided Optimization



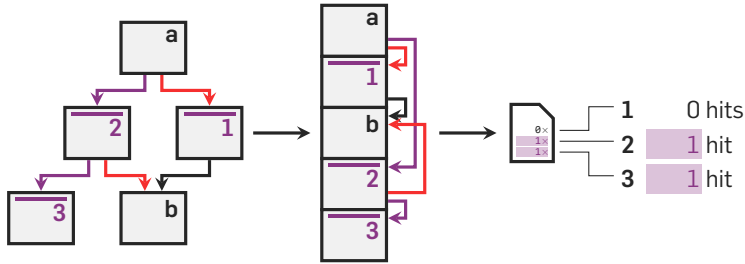
Compilers make **educated guesses** about how a program is going to be used in order to make optimization decisions, but those guesses can be wrong.

Profile-Guided Optimization



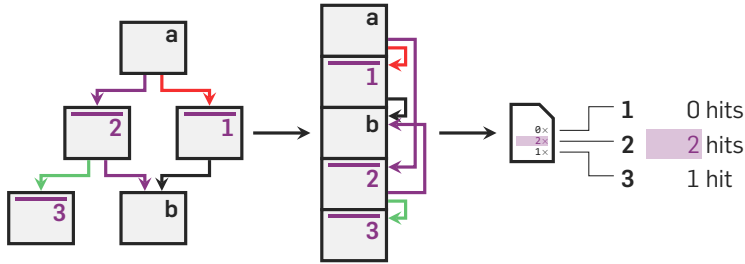
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



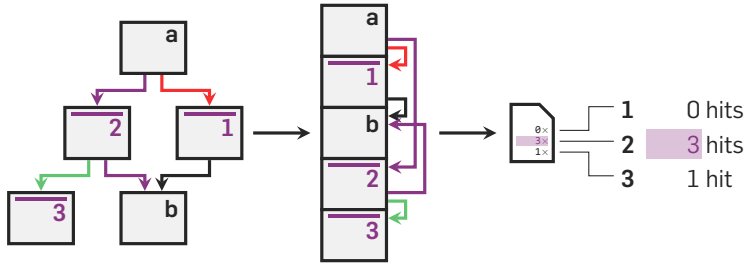
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



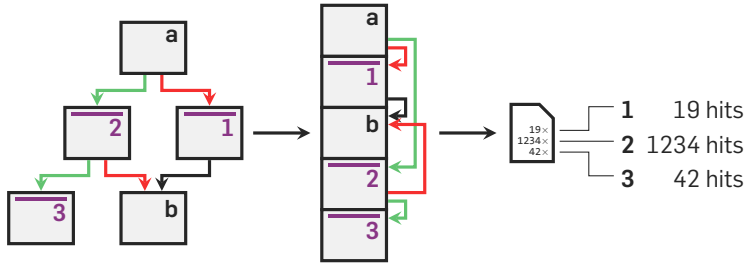
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



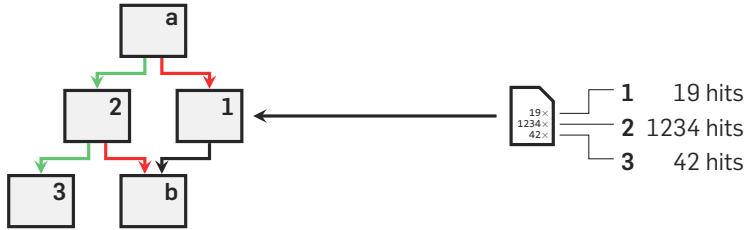
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



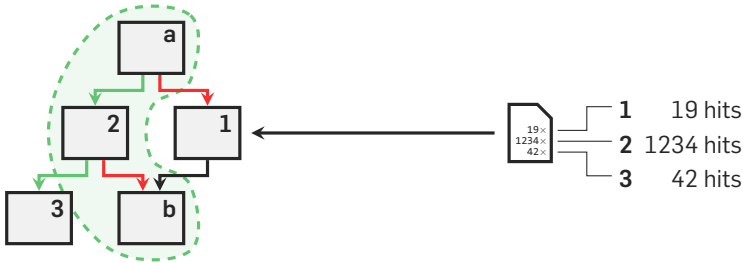
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



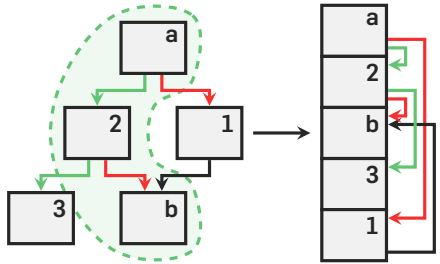
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



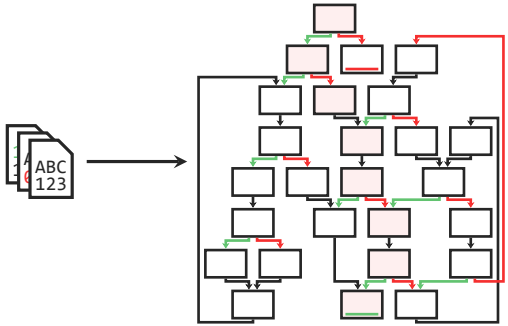
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

Profile-Guided Optimization



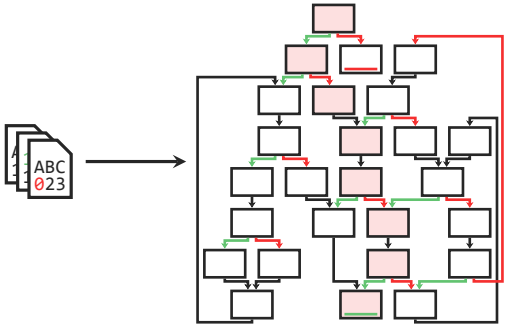
Profile-guided optimization (PGO) optimizes a program based on observed behavior from previous executions. The profile tracks how often each basic block is executed.

PogoFuzz
Core Idea



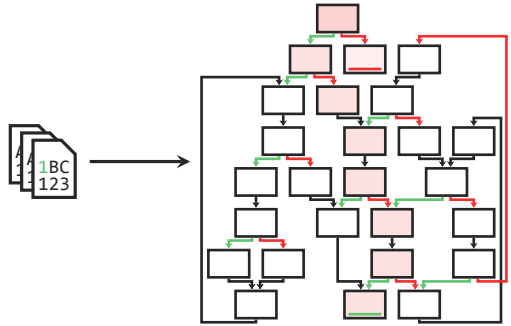
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea



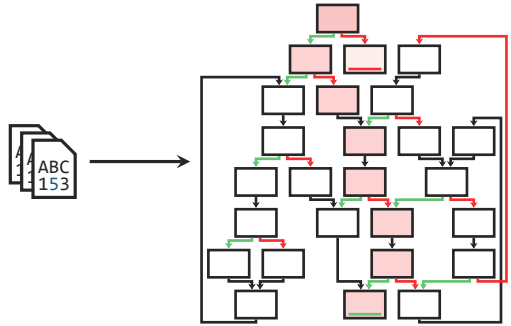
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea



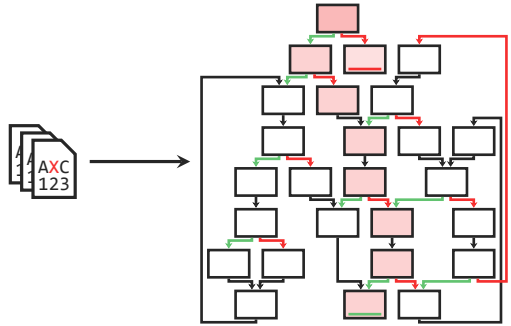
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea



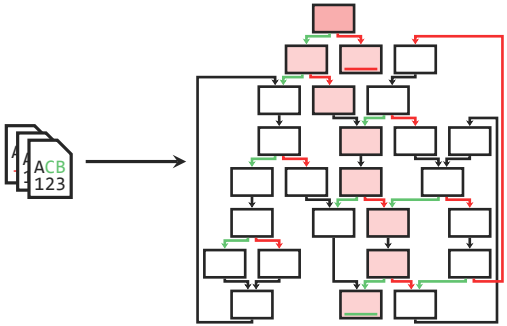
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PoGoFuzz
Core Idea



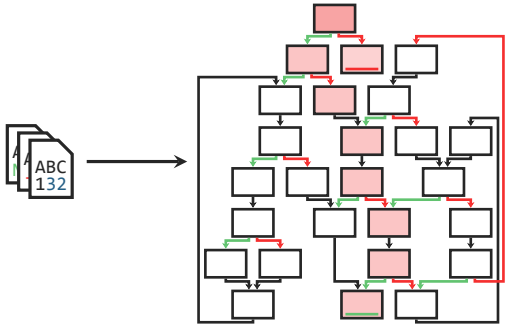
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PoGoFuzz
Core Idea



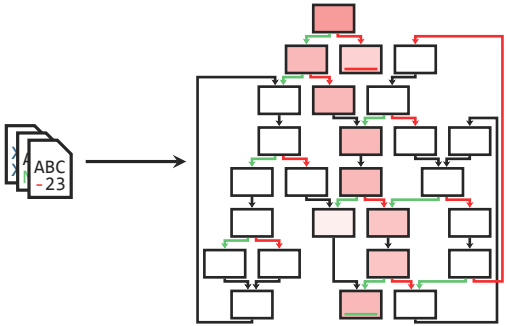
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea



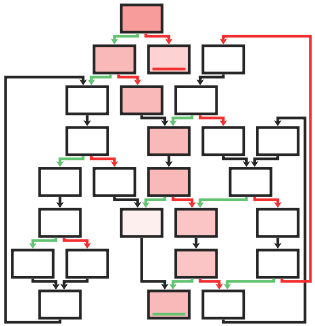
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea



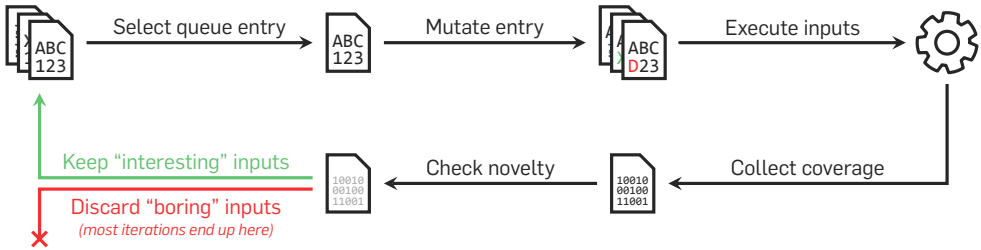
Most mutations do not achieve new coverage or target behavior.
Instead, the target has to run the same code **over and over again**.

PogoFuzz
Core Idea

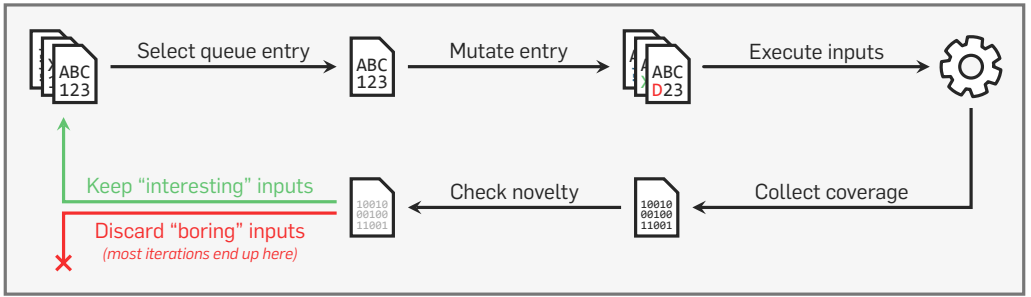


If we **(re)optimize the target** for this common case, the fuzzing loop should run faster.

PogoFuzz
Approach

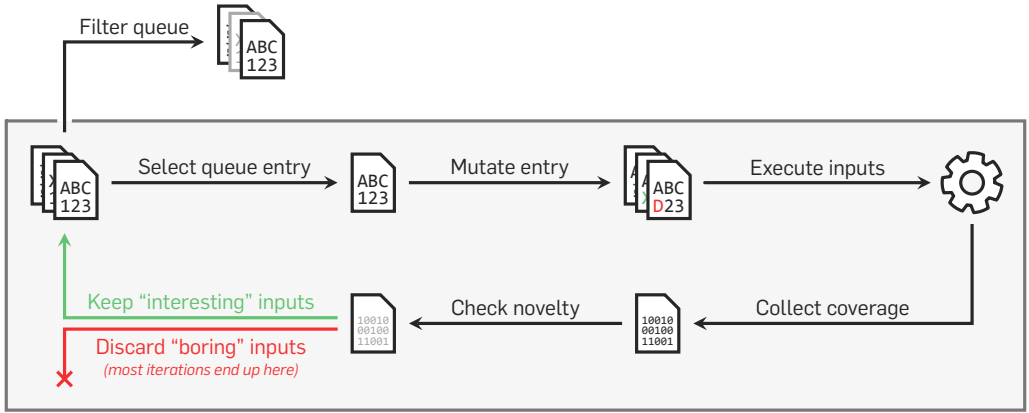


PoGoFuzz
Approach



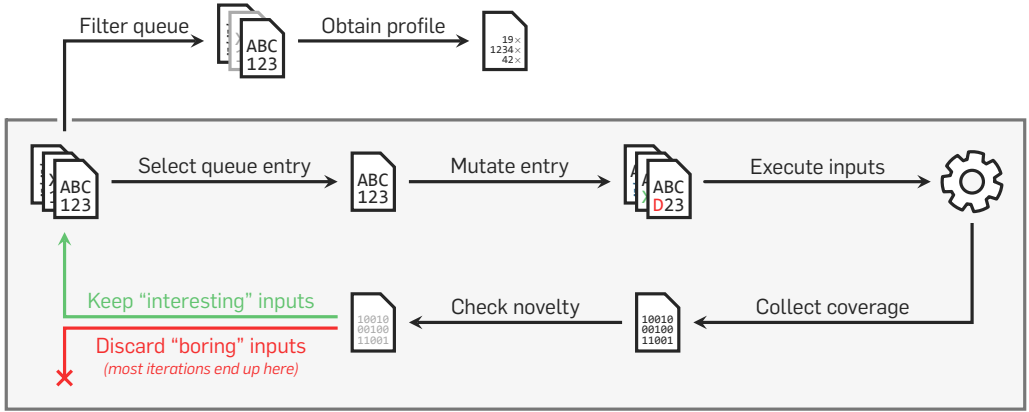
e.g., **AFL++**

PogoFuzz
Approach



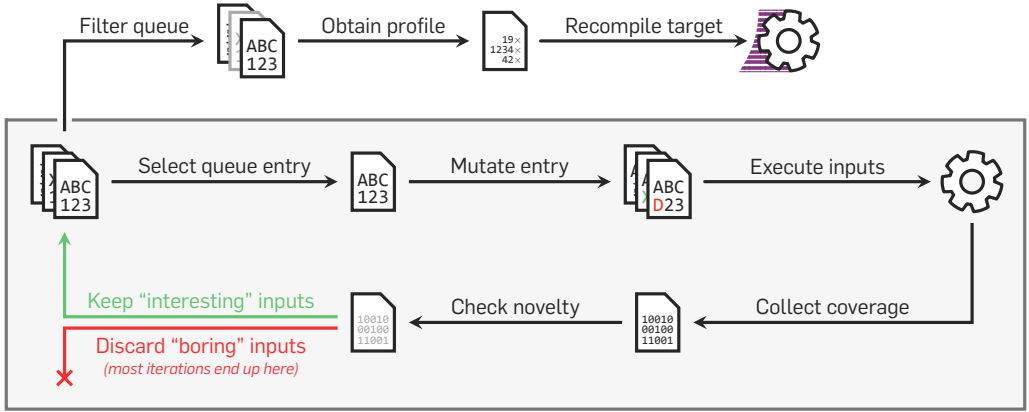
e.g., **AFL++**

PoGoFuzz
Approach



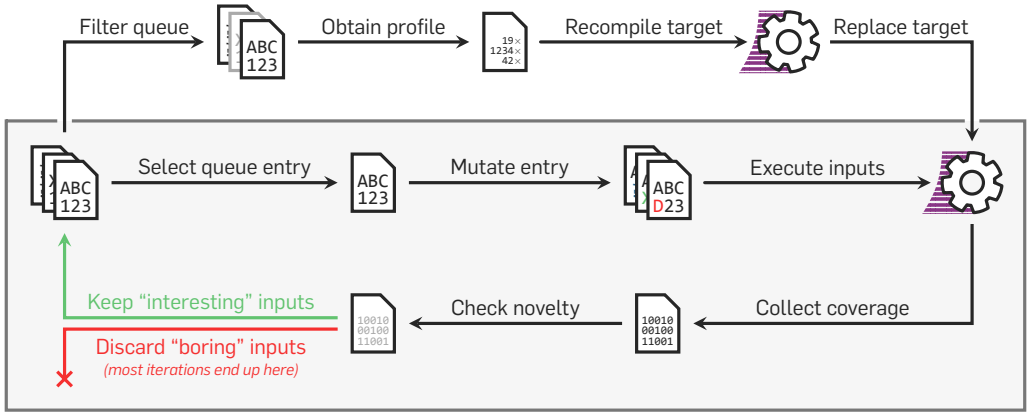
e.g., **AFL++**

PogoFuzz
Approach



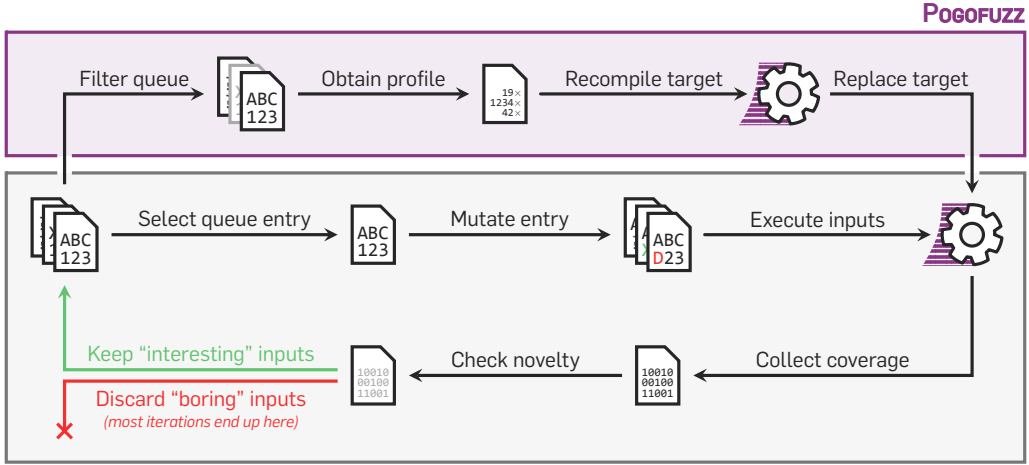
e.g., **AFL++**

PoGoFuzz
Approach



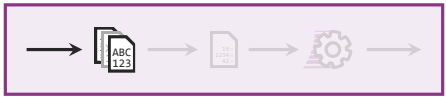
e.g., **AFL++**

POGOFUZZ
Approach



e.g., **AFL++**

PogoFuzz
Queue Filtering



- Obtaining a PGO-compatible profile for a test case requires running it through an instrumented target binary.
- This is not much different from a binary instrumented for fuzzing.
- Doing this on the entire queue is still wasteful: some low-energy test cases are very unlikely to be reexamined by the fuzzer.

We apply a threshold to the fuzzer's internal energy metric to only pick relevant inputs.

PogoFuzz
Profiling

- We obtain a PGO profile using a binary with LLVM's IR PGO instrumentation.
- Queue entries with higher energy are more likely to be picked by the fuzzer, so they should have higher weight in the profile.
- We want to avoid slow post-processing of partial profiles.

We use a custom LLVM pass to directly scale the profile counts of each instrumentation point by the energy of the current test case.

PoGoFuzz
Recompilation

- One of the cores suspends fuzzing to perform the profiling and recompilation.
- Meanwhile, other cores can continue fuzzing on the old binary.
- Once recompilation completes, the target binary is replaced *in situ* through the normal fork server restart mechanism.

The cost of recompilation should amortize as more cores are added.

Preliminary Experiments
Target Selection

- PogoFuzz's impact is likely to vary depending on the target.
- The baseline **execution speed** of the target program plays a key role in this.
- The value of reoptimization depends on **how diverse** the input set is.

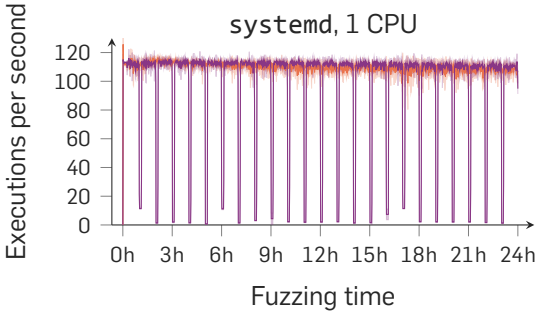
We picked **four FuzzBench targets** for diversity on these factors, based on execution speed and queue size at the end of a vanilla 24h FuzzBench run.

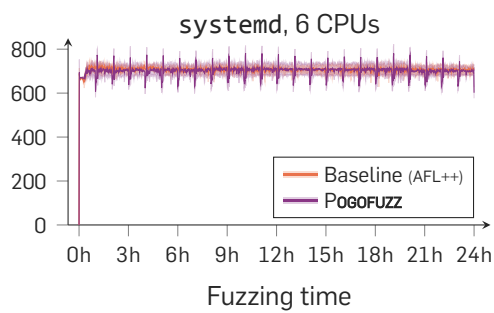
- `libxml2` (slow, large queue)
- `libxslt` (fast, large queue)
- `systemd` (slow, small queue)
- `zlib` (fast, small queue)

Preliminary Experiments
Configurations

- We compare against an unmodified baseline AFL++ (same version).
- For every target, we run an experiment with 1, 2, 3, 4, and 6 cores assigned to fuzzing.
- For `libxslt` and `zlib`, we later extended this to 8 cores.
- Each configuration was evaluated in five trials of 24 hours each.

Results
Impact of Reoptimization





Even the worst-case throughput loss (over 10% for single-core runs of `systemd`) amortizes quickly with more cores.

Results
Overall Performance

Target	Cores	Baseline (AFL++) Median	Baseline (AFL++) σ	PoGoFuzz Median	PoGoFuzz σ	Δ
libxml2	1	100.7	29.9	**104.0**	28.2	+3.3%
	2	192.8	59.6	**214.4**	78.6	+11.2%
	3	304.5	88.3	**382.9**	62.9	**+25.7%**
	4	403.1	80.4	**413.0**	95.1	**+2.5%**
	6	762.2	213.0	**793.2**	124.7	**+4.1%**
libxslt	1	360.6	10.2	**362.3**	9.5	+0.5%
	2	**731.6**	17.0	707.4	45.6	−3.3%
	3	1121.9	13.9	**1149.6**	19.1	**+2.5%**
	4	1476.8	19.9	**1520.1**	23.9	**+2.9%**
	6	2260.7	33.3	**2290.0**	37.3	**+1.3%**
	8	2985.1	37.3	**3096.8**	14.9	**+3.7%**

Target	Cores	Baseline (AFL++) Median	Baseline (AFL++) σ	PoGoFuzz Median	PoGoFuzz σ	Δ
systemd	1	**9.5**	0.1	8.5	0.1	−10.6%
	2	**19.9**	0.6	19.1	0.6	−4.0%
	3	29.8	0.8	**29.9**	0.7	+0.3%
	4	**40.8**	2.0	38.8	0.7	−4.9%
	6	**61.4**	1.4	60.9	2.4	−0.8%
zlib	1	631.8	129.3	**670.7**	21.8	+6.2%
	2	**1540.9**	25.9	1495.9	20.1	−2.9%
	3	2312.8	46.0	**2330.2**	109.3	+0.8%
	4	**3160.1**	16.9	3137.8	8.6	−0.7%
	6	**4758.3**	67.7	4694.2	51.8	−1.3%
	8	6337.6	62.3	**6407.2**	135.7	+1.1%

PoGoFuzz achieves **consistent gains** at 3+ cores for `libxml2` and `libxslt`.

Open Questions

- Can the reoptimization process confuse the the fuzzer instrumentation (e.g., regarding the ordering in the feedback map)?
  - We think this is unlikely (and did not see it in our experiments).
  - PGO-informed optimizations generally occur **after** the instrumentation is inserted.
- Is LLVM's IR-based PGO really the tool of choice here? LLVM offers alternatives:
  - Context-sensitive IR
  - Clang front-end PGO
  - Sampling-based approaches (e.g., CSSPGO)
- How can we determine optimal recompilation points?

Proposed Experiments
Scaling

- The relative cost of recompilation drops as the number of cores increases.
- Does this trend continue with more cores?

We will extend our experiments with additional configurations up to at least **24 cores**.

Proposed Experiments
Generalizability: Targets

- We know that the value of PGO is target-dependent.
- How does PogoFuzz perform on other fuzzing targets?

We will extend our experiments to cover **all FuzzBench targets**.

Proposed Experiments

Generalizability: Fuzzers

- Different fuzzers explore the input space differently.
- How does the choice of fuzzer affect the performance gain?

We will port POGOFUZZ to **other fuzzers** (e.g., Honggfuzz or LibAFL) for evaluation.

Proposed Experiments
Recompilation Interval

- Is recompiling every hour necessary, or is it sufficient to do this less frequently?
- Can we skip recompilation if the queue has not changed significantly?

We will evaluate the impact of **different recompilation intervals** (e.g., 2, 4, or 8 hours), and explore **heuristics** to extend (or shorten) that interval.

Takeaways

PogoFuzz augments existing fuzzers, speeding up fuzzing by **regularly (re)optimizing** the target for the current queue.

Paper and slides are available at
`softsec.link/fz26.pogofuzz`

Funded by
DFG Deutsche Forschungsgemeinschaft
German Research Foundation

CASA
Securing the Digital Society

WWTF
Vienna Science and Technology Fund