Poster: What Makes Fuzzers Struggle? Toward Breaking Coverage Plateaus via Multi-Instance Profiling

Yuichi Sugiyama, Yudai Fujiwara
Ricerca Security, Inc.
Email: {yuichis, yudaif}@ricsec.co.jp

***Abstract*—Coverage-guided fuzzers quickly saturate, leaving most branches unexplored. Existing tools list unreached code but cannot tell which branches are hard for the fuzzer versus unreachable. We run multiple independent fuzzing instances and measure each branch's breakthrough rate (fraction of runs that cover it) to quantify difficulty. Evaluation on 3 OSS-Fuzz targets shows reproducible measurement and that hard branches are largely fuzzer-specific: both fuzzers reach the same code but get stuck at different points.**

1. Introduction

Coverage-guided fuzzers such as AFL++ [1] and libFuzzer [2] reach coverage plateaus early in their campaigns. As Figure 1 shows, all fuzzers plateau within 1–2 hours at only 10–33% branch coverage, leaving the vast majority of branches unexplored even after extended campaigns. These uncovered branches represent improvement opportunities, but their sheer number makes prioritization essential.

Existing tools such as Fuzz Introspector [3] list which code is reached and which is not, but they do not help researchers decide *where to focus*. The unreached set is large and noisy: it mixes branches that the fuzzer finds genuinely hard (e.g., magic-byte checks that a better mutation strategy could overcome) with branches that are fundamentally unreachable under the current setup (e.g., `malloc` failure paths or API functions the harness never calls) [4]. Without distinguishing these, researchers cannot tell which branches are worth investigating.

We propose running N fully independent fuzzing instances and measuring, for each branch that remains only partially covered, the fraction of instances that manage to take both directions (the *breakthrough rate*, M/N). A low but nonzero rate shows the branch is reachable yet hard; resolving it lets the fuzzer reach deeper code. We focus on $M > 0$ branches whose reachability is proven; $M = 0$ branches are set aside as harness/environment limitations cannot be separated from genuine difficulty. We evaluate on 3 OSS-Fuzz targets: reproducibility (**RQ1**) and cross-fuzzer differences (**RQ2**).

2. Methodology

Our approach measures how hard each branch is for a given fuzzer through three steps.

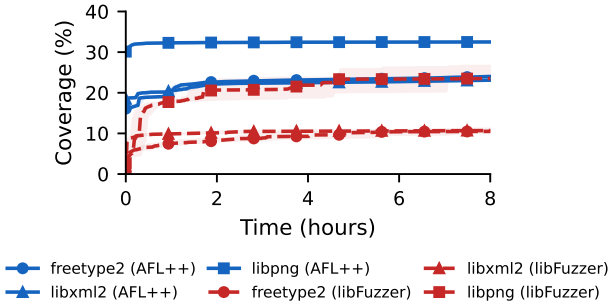


Figure 1: Branch coverage on 3 targets (10 instances, 8 h): all fuzzers plateau within 1–2 hours at 10–33%, leaving the majority of branches uncovered.

(1) Independent fuzzing. For each target and fuzzer, we run N fully independent instances with distinct seeds and no shared state (no `-M/-S`, no corpus sharing). Each instance is an i.i.d. sample from the fuzzer's coverage distribution. The approach integrates with OSS-Fuzz [5].

(2) Branch coverage collection. We replay each instance's final corpus through an LLVM SanitizerCoverage-instrumented binary and record, for each conditional branch, whether each direction was taken at least once. This gives a per-instance binary coverage vector over all branch directions in the target.

(3) Difficulty measurement. A branch is *hard* for a fuzzer if the fuzzer reaches it but consistently takes only one direction: specifically, if $\geq \theta \cdot N$ instances never take the other direction (θ: consensus threshold). The *breakthrough rate* M/N is the fraction of instances that do take both directions. $M > 0$ marks the branch as a proven-reachable improvement target: resolving it opens paths to deeper code and further hard branches. $M = 0$ branches are set aside as they may reflect harness or environment limitations.

3. Evaluation

We evaluate on 3 OSS-Fuzz targets (freetype2, libxml2, libpng) with AFL++ and libFuzzer. **RQ1:** Is difficulty measurement reproducible? **RQ2:** How do hard branches differ across fuzzers?

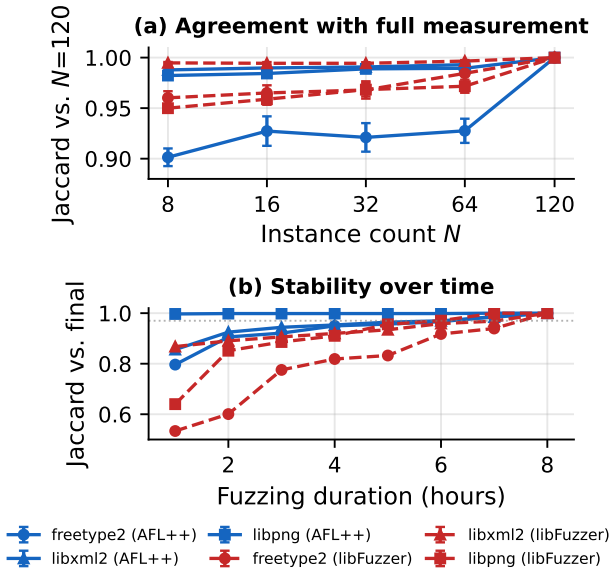


Figure 2: Hard-branch detection stability ($\theta = 0.95$). (a) Jaccard of N-instance subset vs. full 120-instance reference (AFL++ and libFuzzer). (b) Jaccard vs. final over time (10 instances, 8 h).

3.1. RQ1: Reproducibility

We run 120 independent instances per fuzzer for 8 h with $\theta = 0.95$. Figure 2(a) measures how closely a random N-instance subset reproduces the full 120-instance result (Jaccard vs. reference). At $N = 8$, Jaccard exceeds 0.90 for all configurations; most targets reach 0.95+ by $N = 32$.

Figure 2(b) tests temporal convergence (10 instances, 8 h): at each hourly checkpoint, we compare the current hard-branch set to the final result. All configurations reach Jaccard > 0.95 within 5–6 hours.

3.2. RQ2: Cross-Fuzzer Comparison

Using the same 120-instance, 8 h experiment ($\theta = 0.95$), we compare hard branches across AFL++ and libFuzzer. For each branch that is hard for one fuzzer, we check the other fuzzer's status: *not reached* (never reaches the branch), *also hard* (reaches it, same problem), or *not hard* (not a hard branch for that engine). Figure 3 classifies all hard branches into five categories accordingly.

Hard branches are fuzzer-specific. 47–70% of hard branches are in code the other fuzzer never reaches (70% on freetype2, 52% on libxml2, 47% on libpng). AFL++ reaches more code overall, encountering 1.5–2.2× more hard branches than libFuzzer. Complementary cases account for at most 3.7%; combining fuzzers resolves almost none of the hard branches.

Hard branches do not overlap. Among branches with $M > 0$, nearly zero are shared: 0 on freetype2 (49 vs. 41), 2 on libpng (30 vs. 29), 1 on libxml2 (51 vs. 38). Even within the same source file, AFL++ and libFuzzer struggle at different lines

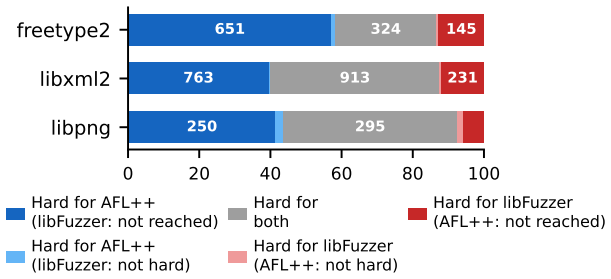


Figure 3: Hard-branch composition (5 categories, 120 instances, $\theta = 0.95$): most are fuzzer-specific; complementary cases are <4%.

(e.g., `sfobjs.c`: AFL++ at 7 lines, libFuzzer at 9, none in common). Both engines achieve similar breakthrough rates (3–4% on average), so the difficulty level is comparable—only the *location* of hard branches differs.

Case study. On freetype2, both engines enter all format parsers, but explore to different depths: AFL++ reaches 105 entries in the TrueType bytecode interpreter (`ttinterp.c`) while libFuzzer has zero; libFuzzer reaches 27 entries in Type 1 parsing (`t1load.c`) while AFL++ has one. The depth difference is systematic across all 120 instances, not a property of individual lucky runs.

4. Conclusion and Future Work

We presented a method that runs N independent fuzzing instances and measures breakthrough rates to quantify how hard each branch is for a given fuzzer. The measurement is reproducible (Jaccard ≥ 0.90 vs. full reference at $N = 8$) and converges within 5–6 hours. Cross-fuzzer comparison reveals that hard branches are almost entirely non-overlapping between AFL++ and libFuzzer, with 47–70% in code the other fuzzer never reaches, suggesting targeted per-fuzzer improvements over naive combination. Future work includes difficulty-based triage and difficulty-aware directed fuzzing.

Acknowledgment

This paper is based on results obtained from a project, JPNP24003, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).

References

[1] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++: Combining incremental steps of fuzzing research," in *USENIX WOOT*, 2020.

[2] LLVM Project, "libFuzzer – a library for coverage-guided fuzz testing," https://llvm.org/docs/LibFuzzer.html, 2024.

[3] D. Korczynski and A. Korczynski, "Fuzz Introspector," https://fuzz-introspector.readthedocs.io/, 2023.

[4] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, "Evaluating fuzz testing," in *ACM CCS*, 2018.

[5] Google, "OSS-Fuzz: Continuous fuzzing for open source software," https://github.com/google/oss-fuzz, 2024.

Poster: What Makes Fuzzers Struggle?
Toward Breaking Coverage Plateaus via Multi-Instance Profiling

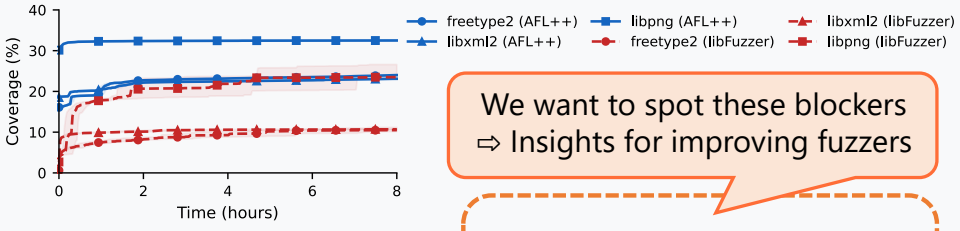
Yuichi Sugiyama, Yudai Fujiwara
Ricerca Security, Inc.

RICERCA SECURITY

1. Motivation: What is to Blame for Coverage Plateaus?

Fact: Coverage saturates within hours.
⇨ Inputs cannot reach some basic blocks

Question: What hinders our fuzzing?
⇨ There exist several types of "Blockers"



We want to spot these blockers
⇨ Insights for improving fuzzers

```c
void Harness(uint8_t *input) {
  FuncNeg(*(uint32_t*)input);
}
```

```c
int64_t FuncNeg(int64_t v) {
  if (v == INT64_MIN)
    FatalError();
  return -v;
}
```

Unreachable due to harness issue

① Harness Code

```c
int InitBuffer(ctx_t *ctx) {
  void *p = malloc(BUF_SZ);
  if (!p) {
    ReleaseContext(ctx);
    return -1;
  }
  ...
}
```

Unreachable under normal setup

② Environment constraints

```c
int LoadFile(file_t *f) {
  if (f->type != MAGIC_EXT)
    LoadLegacyFile(f);
  else
    LoadExtendedFile(f);
  ...
}
```

Unreachable due to fuzzer algorithm

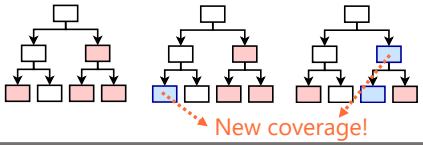
③ Fuzzing algorithm

2. Approach: How to Spot Weaknesses of Fuzzers

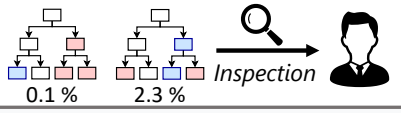
1 Replication
Run *N* independent fuzzers

Distinct seeds, no shared state

2 Identification
Find coincident breakthroughs

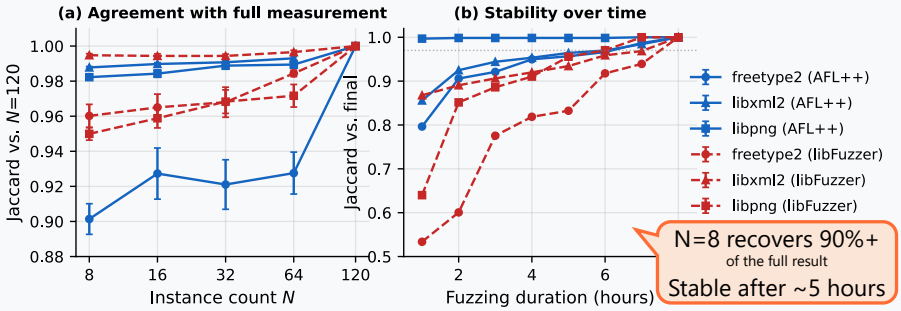


3 Inspection
Rare breakthroughs
= Fuzzer weaknesses



3. Results

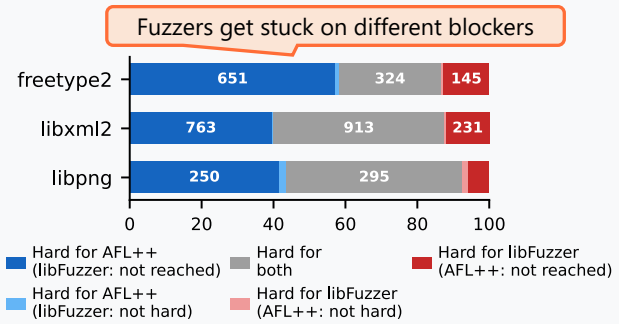
RQ1. Can we detect blockers stably?



N=8 recovers 90%+ of the full result
Stable after ~5 hours

RQ2. Are there fuzzer-specific blockers?

Fuzzers get stuck on different blockers



4. Finding: freetype2

Both fuzzers reach all format parsers,
but each gets stuck at different points within them.

662 are *AFL++*-specific
- ✓ Goes deep into TrueType (ttinterp.c: 105 entries, libFuzzer: 0)
- ✓ Stuck on: bytecode dispatch, glyph loading

152 are *libFuzzer*-specific
- ✓ Goes deep into Type 1 (t1load.c: 27 entries, AFL++: 1)
- ✓ Stuck on: font init, keyword parsing

5. Conclusion & Future Work

✅ Key Results

Identifying rare breakthroughs in multi-instance fuzzing provides **practical insights for improving fuzzers**.
⇨ Reveals **weaknesses specific to each fuzzer** (47–70%).

🚩 Future Work
- Larger-scale experiments
- Root-cause analysis of blockers

References: [1] Fioraldi et al., AFL++, WOOT 2020. [2] LLVM, libFuzzer. [3] Korczynski, Fuzz Introspector, 2023. [4] Klees et al., CCS 2018. [5] Google, OSS-Fuzz.

K Program 経済安全保障重要技術育成プログラム

47th IEEE Symposium on Security and Privacy (IEEE S&P 2026)

This paper is based on results obtained from a project, JPNP24003, commissioned by the New Energy and Industrial Technology Development Organization (NEDO).