Alex Groce
School of Informatics, Computing & Cyber Systems
Northern Arizona University

September 30, 2019

Dear editor and reviewers:

We addressed all reviewer comments where this seeemed feasible within the turnaround time for the special issue, and in particular added a whole new experimental section on non-compiler fuzzer taming to address the generalizability threat. This new section includes use of mutants to "fix" use-after-free faults, which are quite different in character than wrong-code compiler faults, and suggests a possible more general, but low-cost, solution for fuzzer taming/triage. We thank the reviewers for their numerous comments, which allowed us to clarify many subtle points and to considerably extend the scope of related work that is addressed (adding 8 new references, plus much more discussion of some previously cited work). This new version is nearly 4 pages (over 10%) longer than the previous version.

Reviewer 1:

While the proposed approach is not very useful for fuzzing GCC or LLVM (unless new advances in compiler fuzzing appear, which does happen), it is likely very useful in fuzzing, e.g., Swift, Solidity, or any other compiler less aggressively fuzzed to date. Building a high quality fuzzer and establishing ground truths in such a setting, unfortunately, is beyond the scope of this paper.

We have added a number of comments to address these general concerns, and show that our technique can be both useful and (fairly) cheap in non-compiler fuzzing with a new case study, comparing to semantic crash bucketing.

A link to the released compiler mutants and all tests we used is now included in the paper (as is a link to the SCB comparison example). We welcome other uses of our mutants and the tests!

Reviewer 2:

We added an application beyond compiler bugs, and associated discussion, as suggested. The comparison with SCB also suggests a promising avenue of future work, which we are discussing now with the SCB team.

- **pg 2. Line 27. Is it valid for s' to range over all S? Should it instead range over $S\ r_j|j < i$? Otherwise, isn't $min_{j<i}(d(s', r_j))$ guaranteed to be 0? For example, for $d(r_0, r_0)$).** If s' is an already-ranked item, then indeed the *min* will be zero. That means that any s will satisfy the inequality. But for s' not already ranked, the *min* is only over distance to already-ranked items ($r_0 \ldots r_{i-1}$), and could be larger, thus falsifying the inequality. In other words, the condition means that ranking any item next is as good as ranking an already-ranked item, but this only matters when all items are either already ranked or at distance zero from some already ranked item. The condition can be re-worded as follows: "for all s', the closest item to s is at least as far away as the closest item to s'." This is trivially true when $s' = s$ and when $s' = r_j$ for some $j < i$. But the other possible s', not-ranked and not equal to s, force the ranking of the "maximum minimum" item. We added a clarifying sentence, though it's still a somewhat subtle formula, we agree.
- **pg 5. Line 19.** Thanks! This makes the metric much easier to understand.
- **pg 6. Line 23.** Again, thanks, we fixed the notation to make this easier to follow, as proposed.
- All other detailed comments have been addressed, and indeed were errors/possible improvements.

Reviewer 3:

We added a discussion of the idea of using combinatorial approaches to handle compiler optimizations in the

section on possible speed improvements.

While unfortunately mutant vs. correctness kills is hard to explore experimentally, in the compiler context, we added a discussion of the distinction to related work, in the process of considerably extending our examiniation of related work using mutants to localize faults.

Sincerely,

Alex Groce (agroce@gmail.com)
Associate Professor
School of Informatics, Computing &
Cyber Systems
Northern Arizona University

Josie Holmes
Affiliated Researcher
School of Informatics, Computing &
Cyber Systems
Northern Arizona University