The Use of Likely Invariants as Feedback for Fuzzers

Andrea Fioraldi¹, Daniele Cono D’Elia², Davide Balzarotti¹

¹EURECOM, ²Sapienza University of Rome

@andreafioraldi
fioraldi@eurecom.fr
Coverage-guided Fuzzing

The Use of Likely Invariants as Feedback for Fuzzers
Coverage-guided Fuzzing

char buf[BUF_SIZE];

char foo(int x) {
    if (x < sizeof(buf))
        return buf[x];
    return 0;
}
Coverage-guided Fuzzing

```c
char buf[BUF_SIZE];

char foo(int x) {
    if (x < sizeof(buf))
        return buf[x];
    return 0;
}
```

The Use of Likely Invariants as Feedback for Fuzzers
Coverage-guided Fuzzing

```c
char buf[BUF_SIZE];

char foo(int x) {
    if (x < sizeof(buf))
        return buf[x];
    return 0;
}
```
Coverage-guided Fuzzing

```c
char buf[BUF_SIZE];

char foo(int x) {
    if (x < sizeof(buf))
        return buf[x];
    return 0;
}
```

Now the fuzzer cannot find any new coverage.
Property-based Testing

```c
char buf[BUF_SIZE];

char foo(int x) {
    assert (x >= 0);
    if (x < sizeof(buf))
        return buf[x];
    return 0;
}
```

Note: x does not come directly from the input, otherwise it is easy for a fuzzer to trigger the bug thanks to the random generation.
Mining Invariants

- Learn invariants from static analysis
  - Over-approximation, no false positives, too generic invariants
Mining Invariants

- Learn invariants from static analysis
  - Over-approximation, no false positives, too generic invariants

- Learn *likely* invariants from dynamic analysis
  - Needs a corpus of testcases
  - Suffers from the “Coverage Problem”, likely invariants are in worst case only local properties of the observed executions
  - So many false positives
Idea: Likely Invariants as Feedback

- In Fuzzing, we are interested in storing testcases that trigger new program states. Violations of likely invariants can distinguish “unusual” states of the variables.
Idea: Likely Invariants as Feedback

● In Fuzzing, we are interested in storing testcases that trigger new program states. Violations of likely invariants can distinguish “unusual” states of the variables.

● Coverage Problem? Not a problem, local properties are interesting constraints to separate the space of the possible values.
Idea: Likely Invariants as Feedback

- In Fuzzing, we are interested in storing testcases that trigger new program states. Violations of likely invariants can distinguish “unusual” states of the variables.
- Coverage Problem? Not a problem, local properties are interesting constraints to separate the space of the possible values.
- Basic block, not function, level (ok for memory unsafe languages)
- $Y \in \{0, 2, 256\}$
- $Y < X$

The space described by the variables in the basic block is divided in 4 subspaces, A B C D
Pruning Invariants

- Compute set of related variables and do not generate invariants between unrelated variables (*Comparability* calculation algorithm)
Pruning Invariants

- Compute set of related variables and do not generate invariants between unrelated variables (*Comparability* calculation algorithm)
- Compute basic invariants using intra-procedural *Range Analysis* and instruct the learner to not output inviolable checks
Pruning Invariants

● Compute set of related variables and do not generate invariants between unrelated variables (*Comparability* calculation algorithm)

● Compute basic invariants using intra-procedural *Range Analysis* and instruct the learner to not output inviolable checks

● Traverse the *Dominator Tree* to avoid emitting the same check over the same variables different times in different basic blocks
Prototype: InvsCov

Based on the AFL++ Fuzzing Framework and LLVM. We provide two LLVM passes and Clang wrappers.

Likely invariants mined using the Daikon dynamic invariants detector.
Instrumentation

We extend the classic AFL-style instrumentation to keep track of the combination of violated invariants.

```c
// Original AFL edge-coverage code
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;

// Extended to capture violations of invariants
__afl_area_ptr[cur_loc ^ prev_loc]++;
prev_loc = cur_loc >> 1;
prev_loc ^= __daikon_constr_1(var0);
prev_loc ^= __daikon_constr_2(var2, var3);
...
```
Evaluation

We evaluated our prototype using old versions of 8 real-world subjects used in previous works. The initial corpus is the result of a previous 24h fuzzing campaign, then we run the fuzzers for 48h.

- 62% growth on the fuzzer’s queue size (no state explosion)
- 8% slowdown on the average testcase execution speed
- 1% growth on total edge coverage
Evaluation

- 56% growth on bugs found vs. traditional code coverage
- 41% growth on bugs found vs. context-sensitive coverage
- 67% of the bugs that only InvsCov could find were reached but not triggered by CodeCov
Future Directions

- On-demand learning of invariants during fuzzing. Implement a faster version of Daikon that is also adaptive.
Future Directions

- On-demand learning of invariants during fuzzing. Implement a faster version of Daikon that is also adaptive.
- Track implicit memory values not used in a block but still interesting (e.g. the size of buffers).
Future Directions

- On-demand learning of invariants during fuzzing. Implement a faster version of Daikon that is also adaptive.
- Track implicit memory values not used in a block but still interesting (e.g. the size of buffers).
- Identify other ways to remove impossible-to-violate likely invariants at compile time.
Future Directions

- On-demand learning of invariants during fuzzing. Implement a faster version of Daikon that is also adaptive.
- Track implicit memory values not used in a block but still interesting (e.g. the size of buffers).
- Identify other ways to remove impossible-to-violate likely invariants at compile time.
- What happens when we guide the fuzzer towards “rare” invariants violations?
Thank you!

Questions?