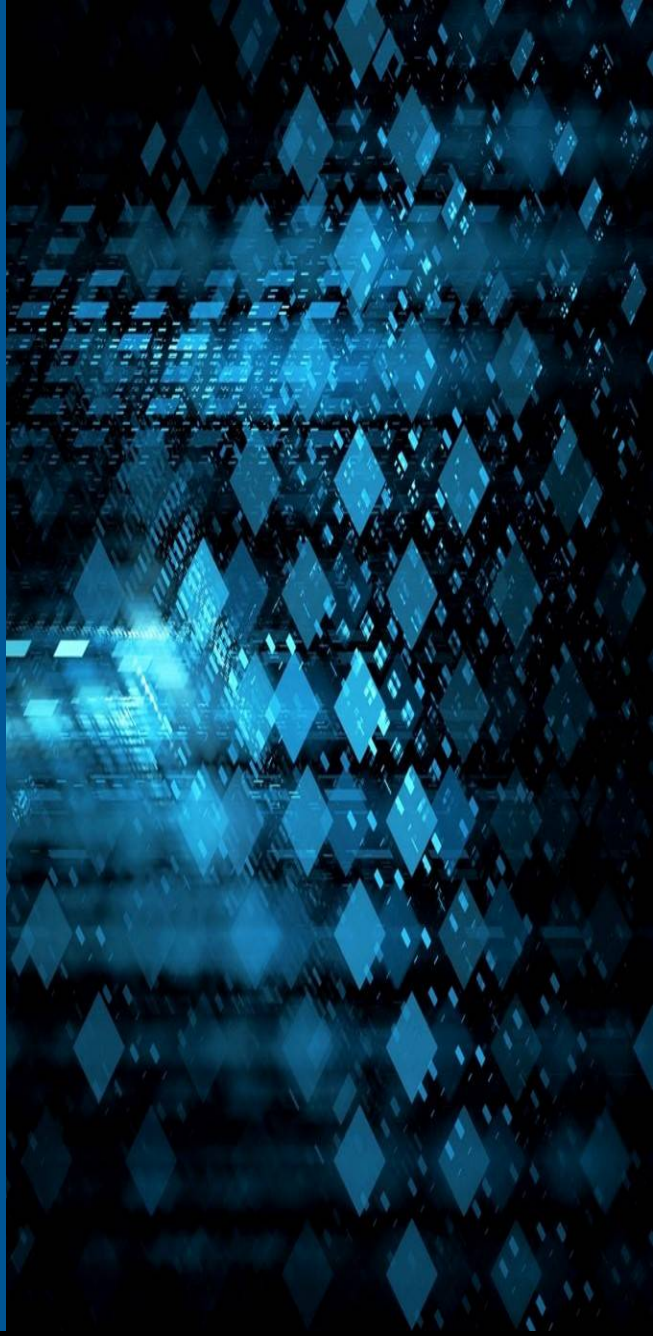
Challenges and Solutions for Automated Repair of C Code

Will Klieber

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA

IEEE SecDev Conference
September 2017



This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM17-0676

Why automated repair?

Static analysis tools help find bugs, but they typically:

- are used late in the development process, and
- produce an enormous number of false positives.

Reducing false positives is difficult, but we don't need to.

It's okay to 'repair' false positives, if we don't break code.

- The small runtime overhead is often acceptable.

Why at source code level (instead of a compiler pass)?

Repair on source code	Compiler pass
One-time change to source code.	Permanent change to the build process.
Repairs can be easily manually audited.	Must trust the tool.

Gap between static analysis and repair

Static Analysis to detect bugs

works best on an intermediate representation (IR)

Gap

Repair

on the original souce code

Difficulty: must map IR back to source.

Design for IR↔source mapping

- We augment the IR with *tags* that record how to transform the IR code back to source.
  - Tags do not affect the semantics of the IR.
  - The nodes of the original abstract syntax tree (AST) are tagged with the exact text of corresponding source code (including whitespace and macros).
- We have developed a set of reversible transformations that start with the original AST and transform it step-by-step to the IR.
- The IR is repaired and then transformed back to source using the tags.
  - If a repair invalidates a tag, then the tag is ignored.
- IR-to-source transformation should be sound (semantics-preserving).
- **Notice any problems?**

Sequence Points

An IR usually has a well-defined evaluation order, but C does not.

- E.g.:

```
int x = 0;
x = x++;
printf("x=%i\n", x);
```

- What value is printed?
  - With gcc 4.6, "x=1" is printed.
  - With gcc 5.2, "x=0" is printed.

Therefore, naively reconstructing C expressions from totally-ordered IR instructions is unsound in general.

We solve this problem by introducing IR instructions that explicitly indicate a partial order for execution.

IR instructions to indicate unsequenced ops

```
     spawn_unseq [L1, ..., Ln], end;
L1:;
     body1;
     merge_unseq end;
L2:;
     body2;
     merge_unseq end;
     ...
Ln:;
     bodyn;
     merge_unseq end;
end:;
```

Semantics: for all $i \neq k$, every instruction in $body_i$ is unsequenced w.r.t. every instruction in $body_k$.

IR instructions to indicate weak sequencing

```
    weak_seq mid, end;
    body1;
    merge_weak end;
mid:;
    body2;
end:;
```

Semantics: side effects in $body_1$ are unsequenced w.r.t. $body_2$, and value computations in $body_1$ are sequenced before $body_2$.

Note: There are also other IR instructions needed to indicate other types of partial order in C.