Challenges and Solutions for Automated Repair of C Code

Will Klieber

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA

IEEE SecDev Conference
September 2017
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)?

<table>
<thead>
<tr>
<th>Repair on source code</th>
<th>Compiler pass</th>
</tr>
</thead>
<tbody>
<tr>
<td>One-time change to source code.</td>
<td>Permanent change to the build process.</td>
</tr>
<tr>
<td>Repairs can be easily manually audited.</td>
<td>Must trust the tool.</td>
</tr>
</tbody>
</table>
Gap between static analysis and repair

Static Analysis to detect bugs

- works best on an intermediate representation (IR)

Repair

- on the original source 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.:
  
  ```c
  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

\[
\text{spawn}_\text{unseq} \ [L_1, \ldots, L_n], \text{ end};
\]

\[L_1:;\]
\[\text{body}_1;\]
\[\text{merge}_\text{unseq} \ \text{end};\]

\[L_2:;\]
\[\text{body}_2;\]
\[\text{merge}_\text{unseq} \ \text{end};\]

\[\ldots\]

\[L_n:;\]
\[\text{body}_n;\]
\[\text{merge}_\text{unseq} \ \text{end};\]
\[\text{end};\]

Semantics: for all \(i \neq k\), every instruction in \(\text{body}_i\) is unsequenced w.r.t. every instruction in \(\text{body}_k\).
IR instructions to indicate weak sequencing

\[
\text{weak_seq } \text{mid, end};
\]
\[
\text{body}_1;
\]
\[
\text{merge_weak } \text{end};
\]
\[
\text{mid};
\]
\[
\text{body}_2;
\]
\[
\text{end};
\]

Semantics: side effects in $\text{body}_1$ are unsequenced w.r.t. $\text{body}_2$, and value computations in $\text{body}_1$ are sequenced before $\text{body}_2$.

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