Using Likely Invariants For Automated Software Fault Localization

Swarup Sahoo, John Criswell, Chase Geigle, Vikram Adve

Presented By: Matthew Perez and Thomas Rupp
Motivation

• Detecting software bugs is time consuming and difficult
  • Lots of code to sift through
  • Multiple failures simultaneously

• Popular debugging methods: manual search

• Compiler shows program crash, but not cause of error
Goal/Related Works

- Automated system for bug detection
- Input: Source code, config files, failing input, input grammar (optional)
- Output: Line(s) of code where failure originated (root cause)

- Other works?
  - Tarantula/Ochiai
Technical Overview and Insights

- Generating Input
- Likely Program Invariants
- Dynamic Backwards Slicing
- Dependence Filtering
- Filtering Using Multiple Failing Inputs
Input Generation

• Generate “good inputs”
  - Similar: ddmin algorithm, minimize distance
  - Don’t cause program failure

• Delta debugging: learn about system using similar inputs

• Select 8 inputs compared to thousands
  - Reason: less likely to miss root cause
Likely Program Invariants

• Functions w/ expected output (ex. monthsOfYear)

• Good inputs generate range invariants (ex. [1, 12])
  - Broken range invariants => candidate root cause

• Checked on load, store, and return

• Key: Small number of inputs => weak invariants
  => broken invariants => candidate root causes
Dynamic Backwards Slicing (Giri)

• Most failed invariants are not root causes

• Traceback failure using data/control flow

• Remove failed invariants that are not part of slice.

• Effectiveness? Removes 58% of false positives
Dependence Filtering

- Faulty instructions tend to trickle down
- If $x$ fails an invariant $x \times 2$ probably will too
- Dynamic Dependence Graph (DDG)
- Removes an additional 55% of false positives
Filtering w/ Multiple Failing Inputs

- Utilizes redundancy of failing inputs
- Creates a similar input that produces the same fail
- Repeat the previous techniques, generate sets
- Find the intersection of these sets
Experimental Methodology

Used: HTTP Proxy server, MySQL, and Apache (MySQL has millions of lines)

Used a variety of errors to demonstrate robustness
Experimental Evaluation

Reduced MySQL to 0.002% of its original size

This method seems to be much more consistent than others

### Table 3. Number of Reported Root Causes

<table>
<thead>
<tr>
<th>Bug#</th>
<th>#InvS</th>
<th>Failed Invs</th>
<th>Slice</th>
<th>Depe-</th>
<th>Multiple</th>
<th>Src-</th>
<th>Root Cause in final step?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug-1</td>
<td>3358</td>
<td>357</td>
<td>30</td>
<td>9</td>
<td>9</td>
<td>49</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-2</td>
<td>5917</td>
<td>95</td>
<td>36</td>
<td>16</td>
<td>12</td>
<td>48</td>
<td>No</td>
</tr>
<tr>
<td>Bug-3</td>
<td>5942</td>
<td>93</td>
<td>27</td>
<td>9</td>
<td>6</td>
<td>64</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-4</td>
<td>6847</td>
<td>156</td>
<td>44</td>
<td>14</td>
<td>8</td>
<td>28</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-5</td>
<td>4566</td>
<td>130</td>
<td>34</td>
<td>18</td>
<td>17</td>
<td>89</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-6</td>
<td>4652</td>
<td>83</td>
<td>13</td>
<td>7</td>
<td>5</td>
<td>26</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-7</td>
<td>5836</td>
<td>153</td>
<td>35</td>
<td>17</td>
<td>11</td>
<td>152</td>
<td>Yes</td>
</tr>
<tr>
<td>Bug-8</td>
<td>2295</td>
<td>120</td>
<td>12</td>
<td>6</td>
<td>6</td>
<td>171</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### Table 4. Tarantula/Ochiai Results

<table>
<thead>
<tr>
<th>Bug</th>
<th>Tarantula</th>
<th>Ochiai</th>
<th>Our Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bug-1</td>
<td>[6 - 5266]</td>
<td>[2 - 5255]</td>
<td>49</td>
</tr>
<tr>
<td>Bug-2</td>
<td>[34 - 34 ]</td>
<td>[34 - 34]</td>
<td>48</td>
</tr>
<tr>
<td>Bug-3</td>
<td>[55 - 9409]</td>
<td>[55 - 9409]</td>
<td>64</td>
</tr>
<tr>
<td>Bug-4</td>
<td>[776 - 9847]</td>
<td>[694 - 9762]</td>
<td>28</td>
</tr>
<tr>
<td>Bug-5</td>
<td>[1 - 19]</td>
<td>[1 - 16]</td>
<td>89</td>
</tr>
<tr>
<td>Bug-6</td>
<td>[5680 - 6878]</td>
<td>[5678 - 6876]</td>
<td>26</td>
</tr>
<tr>
<td>Bug-7</td>
<td>[8 - 6060]</td>
<td>[8 - 6060]</td>
<td>152</td>
</tr>
<tr>
<td>Bug-8</td>
<td>[28 - 5372]</td>
<td>[1 - 5357]</td>
<td>171</td>
</tr>
</tbody>
</table>
Conclusion

Seems to work generally better than Tarantula and Ochiai

Missing code bugs are impossible for this technique to detect

Apparently the authors have improved the input generation so they catch bug-2

Can be combined with other tools to create a complete picture

Generation of invariants for function arguments, reduce expression tree size
Discussion and Questions

- How is this useful in real world, is it worth using over other tools?

- Filtering using multiple failing inputs removed a root cause in the experiments, is the technique worth including in a tool like this?

- Missing Code bugs are something that this technique does not detect, does that significantly impact its usefulness?