Using Likely Invariants for Automated Software Fault Localization

ASPLOS-2013
Goal

- Automated System to detect the root causes of bugs.
- Efficient, scalable, low false positive rate.
Contributions

1. Likely invariants using auto generated inputs “close” to failing input.
2. Invariants + Dynamic slicing to locate potential root causes 
3. False positive reduction heuristics.
4. Seems to narrow root-cause to 5-17 locations for real Apache/Squid bugs.
Why is this tool necessary?

Similar to delta debugging.
But Delta debugging doesn’t scale --- comparing memory states is expensive.

Key insight: Use likely invariants to compare runs.
High-Level Flow

Program

Generate Inputs

Generate Invariants

Test with Bad Input

Backward Slicing

Dependence Filtering

Multi-faulty Inp Filter

Bad Inputs

Good Inputs

Instrumented Program

Failed Invariants

False+ve Filters
Generating Inputs

1. Generate inputs close to failing input
2. 1-character edit-distance inputs generated by using `ddmin` tool.
3. If inputs are formally specified, then the grammar is used to generate good inputs.
4. Generated inputs should either pass the test or fail the test in the same manner.
Diagnosis With Invariants

- Use likely range invariants
- Range of values computed for correct runs
- Violated invariants => candidate root cause
- Invariants on load/store/return (LLVM instructions)

<table>
<thead>
<tr>
<th>Value type</th>
<th>Instruction</th>
<th>Invariant</th>
</tr>
</thead>
<tbody>
<tr>
<td>return</td>
<td>return int %weekday</td>
<td>0&lt;=%weekday&lt;=6</td>
</tr>
<tr>
<td>load</td>
<td>%value = load int* %p</td>
<td>0&lt;=%value</td>
</tr>
<tr>
<td>store</td>
<td>store int %val, int* %q</td>
<td>100&lt;=%val&lt;=100</td>
</tr>
</tbody>
</table>
Dynamic Backward Slicing

Given a fault, find all instructions which lead to the fault location.

Backward slicing determines which instructions affected a particular value in the failing instruction (backward data-flow analysis).
Filtering False Positives

Dynamic backward slicing effectively removes nearly 80% of the false positives (From Results)
False positives are also avoided with “good” inputs
  ● Control flow and program values are close to failing runs
  ● Few in number, allows for more control
Ensures that the training runs are close to the failing run
Narrows invariants down making them more relevant
Dependence Filtering
Filtering False Positives

Dependance filtering removes chains of false invariants. Dependance filtering on average removes 58% of bugs.

\[ \text{var } x \text{ fails a range invariant because it is negative} \]

Then a subsequent invariant which is a computation that uses \( x \) also fails.

We don't need both.
Evaluation

Evaluation Criteria

1. Can the true root of a bug be found
2. Number of false positives generated

Summary and Limitations

- The route cause was not found for one out of eight bugs
- 9.25(average) Invariants generated per bug
- Cannot currently handle missing code bugs
- Without robust inputs and proper training, bugs can be dropped
- Diagnosis time for bugs ranged from 8 min. to 4 hours (not an issue)
1. Are only 8 bugs across 3 projects a sufficient evaluation criteria? (6 bugs from 1 project)
2. 5 of the 8 bugs have stack depth<5, which can be easily fixed manually
3. How much useful information can be lost through filtering
Research Questions

1. Having good inputs seems to be critical. Can it always be done?
2. If your input has a clear specification, why not automate test generation?
3. Is instruction level the right way, or some higher-level semantics easier to slice on? Basic blocks, functions, variables, etc?
4. What can the take-away from this work be? It seems to throw the kitchen-sink at the problem and produce non-stellar results.