Thin Slicing

Authored by Manu Sridharan, Stephen J. Fink, Rastislav Bodík

University of California, Berkeley & IBM T.J. Watson Research Center
PLDI 2007

Presented by Jing Pu
Outline

- Motivation
- Approach
- Definitions & Key Techniques
- Evaluation
- Conclusion
Motivation

- Debugging and Program Understanding tasks
  - Finding buggy statement, diagnose bug (find most relevant statements to the bug)
  - Understanding relevant statements, e.g., aliasing, important conditionals

- Drawbacks of traditional static slicing techniques
  - Overly Broad relevance definition
  - Slice Pollution
Approach

- **Thin Slicing**
  - *Redefine relevance* – intuitive semantic definition
  - *Hierarchical expansion* – providing additional information

- **Terminology**
  - *Seed statement*
  - *Producer statement*
  - *Explainer statement*
  - *Dependences*
Definitions – producer statements

- **Seed statement**
  A statement or value of interest, e.g., the position in a program where an error occurs

- **Producer statement**
  - Statement $s_1$ is a producer for statement $s_2$ if $s_1$ is part of a chain of assignments that computes and copies a value to $s_2$
  - $S_2$ – seed / other producer
  - Direct uses of memory locations (variables & object fields)

- **Explainer statement**
  None-producer statements:
  - Heap-based value flow
  - Control flow
Definitions – Dependences

- Thin slicing
  Producer flow dependences
  Ignore:
    - Base pointer flow dependences
    - Control dependences

- Most Tasks
  Very few explainers are needed

- Answer more questions, need more explanation
  Expand thin slices, need ignored dependences
Example 1: A dependence graph for a program

```
1  x = new A();
2  z = x;
3  y = new B();
4  w = x;
5  w.f = y;
6  if (w == z) {
7      v = z.f;  // the seed
8  }
```

Heap-based value flow

Control flow

Direct uses of memory locations in the thin slicing for the seed
Example 2:
thin slices expansion

Diagnose bug:
Which statements cause the ‘this’ pointers of close() and isOpen() to be aliased
Computing thin slices

- **Step 1**: Do precise pointer analysis:
  - Compute call graph
  - Compute may-mod and may-use sets for each method
- **Step 2**: Build a CFG for each method in the program
- **Step 3**: Compute each CFG’s control and data dependences
- **Step 4**: Build PDG for each CFG
- **Step 5**: Connect the PDGs to form the SDG:
  - Context-Insensitive/Context-sensitive: Add direct edges for heap access statements

Reference: “Slicing Java Programs that Throw and Catch Exceptions”
Computing thin slices — Modification

Data dependences for heap access statements
For a statement $x.f := e$, we add an edge to each statement with an expression $w.f$ on its right-hand side, such that the precomputed points-to analysis indicates $x$ may-alias $w$.

Context-Insensitive Thin Slicing
- Add direct edges across procedure
- Do not use heap parameters

Context-sensitive Thin Slicing
- Add direct edges in the same procedure
- Use extra parameters and return values to model heap access
Evaluation

Experimental Setup

- Implemented thin slicing (context-insensitive / context-sensitive) & traditional slicing (context-insensitive / context-sensitive) using WALA
- Used SUN JDK 1.4.2_09 standard library code
- Machine: A Lenovo ThinkPad t60p with dual 2.2GHz Intel T2600 processors and 2GB RAM
- Analyzer: ran on the Sun JDK 1.5_07 using at most 1GB of heap space.
- Benchmarks: SIR, SPECjvm98
Benchmark characteristics: derived from methods discovered during on-the-fly call graph construction, including Java library methods
Evaluation (cont’d)

Key points

- uses injected bugs from the SIR suite
- Seed: as the point of failure
- Desired statements: the cause of the bug
- Control dependence: manually pre-determined

Scalability

- context-insensitive (thin/traditional): good
- context-sensitive traditional for small codes: good
- context-sensitive traditional for large codes: bad

Precision

- Traditional: Measuring Slice Size
- New Approach: Use a breadth-first traversal strategy to simulate the statements inspecting process done by the user, terminated when discover all required statements for original problem
- context-sensitive thin: not practical
Experiment 1 – Locating bugs

- **Target**
  - Check if thin slices include the buggy statement
  - Compare inspected slice size (thin / traditional)

<table>
<thead>
<tr>
<th>Bug</th>
<th># Thin</th>
<th># Trad.</th>
<th>Ratio</th>
<th># Control</th>
<th># ThinNoObjSens</th>
<th># TradNoObjSens</th>
</tr>
</thead>
<tbody>
<tr>
<td>nanoxml-1</td>
<td>12</td>
<td>32</td>
<td>2.67</td>
<td>0</td>
<td>12</td>
<td>32</td>
</tr>
<tr>
<td>nanoxml-2</td>
<td>25</td>
<td>113</td>
<td>4.52</td>
<td>0</td>
<td>431</td>
<td>1675</td>
</tr>
<tr>
<td>nanoxml-3</td>
<td>29</td>
<td>123</td>
<td>4.24</td>
<td>0</td>
<td>472</td>
<td>1883</td>
</tr>
<tr>
<td>nanoxml-4</td>
<td>12</td>
<td>33</td>
<td>2.75</td>
<td>1</td>
<td>17</td>
<td>44</td>
</tr>
<tr>
<td>nanoxml-5</td>
<td>35</td>
<td>156</td>
<td>4.46</td>
<td>1</td>
<td>159</td>
<td>45</td>
</tr>
<tr>
<td>nanoxml-6</td>
<td>12</td>
<td>52</td>
<td>4.33</td>
<td>0</td>
<td>35</td>
<td>90</td>
</tr>
<tr>
<td>jtopas-1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>jtopas-2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ant-1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>ant-2</td>
<td>4</td>
<td>5</td>
<td>1.25</td>
<td>0</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>ant-3</td>
<td>34</td>
<td>55</td>
<td>1.62</td>
<td>15</td>
<td>251</td>
<td>501</td>
</tr>
<tr>
<td>ant-4</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>xml-security-1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Experiment 2 – Program understanding

- What is a tough cast?
- Thin Slicing – Understanding the safety of tough cast

```java
1. class Node {
   2.     final int op;
   3.     static int ADD_NODE_OP = 1;
   4.     Node(int op) { this.op = op; }
   5. }
6. class AddNode extends Node {
   7.     AddNode(...) {
   8.         super(ADD_NODE_OP);...
   9.     }
10. }
11. void simplify(Node n) {
   12.     int op = n.op;
   13.     switch (op) {
   14.         case ADD_NODE_OP:
   15.             AddNode add = (AddNode) n;
   16.             ...
   17.     }
18. }
```

Tough Cast
Investigate 10 random tough casts for each SPEC benchmark

Compared BFS traversal sizes to manually identified required statements size

<table>
<thead>
<tr>
<th>Cast</th>
<th># Thin</th>
<th># Trad.</th>
<th>Ratio</th>
<th># Control</th>
<th># ThinNoObjSens</th>
<th># TradNoObjSens</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtrt-1</td>
<td>22</td>
<td>51</td>
<td>2.32</td>
<td>0</td>
<td>22</td>
<td>51</td>
</tr>
<tr>
<td>mtrt-2</td>
<td>23</td>
<td>52</td>
<td>2.26</td>
<td>0</td>
<td>23</td>
<td>52</td>
</tr>
<tr>
<td>jess-1</td>
<td>6</td>
<td>7</td>
<td>1.17</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>jess-2</td>
<td>13</td>
<td>39</td>
<td>3</td>
<td>0</td>
<td>25</td>
<td>93</td>
</tr>
<tr>
<td>jess-3</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>jess-4</td>
<td>6</td>
<td>7</td>
<td>1.17</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>jess-5</td>
<td>6</td>
<td>7</td>
<td>1.17</td>
<td>2</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>jess-6</td>
<td>6</td>
<td>6</td>
<td>1</td>
<td>2</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>javac-1</td>
<td>57</td>
<td>910</td>
<td>16</td>
<td>1</td>
<td>57</td>
<td>910</td>
</tr>
<tr>
<td>javac-2</td>
<td>43</td>
<td>853</td>
<td>19.8</td>
<td>1</td>
<td>43</td>
<td>853</td>
</tr>
<tr>
<td>javac-3</td>
<td>65</td>
<td>2224</td>
<td>34.2</td>
<td>1</td>
<td>65</td>
<td>2267</td>
</tr>
<tr>
<td>javac-4</td>
<td>45</td>
<td>855</td>
<td>19</td>
<td>1</td>
<td>45</td>
<td>855</td>
</tr>
<tr>
<td>jack-1</td>
<td>18</td>
<td>79</td>
<td>4.39</td>
<td>0</td>
<td>303</td>
<td>758</td>
</tr>
<tr>
<td>jack-2</td>
<td>57</td>
<td>151</td>
<td>2.65</td>
<td>0</td>
<td>339</td>
<td>647</td>
</tr>
<tr>
<td>jack-3</td>
<td>18</td>
<td>69</td>
<td>3.83</td>
<td>0</td>
<td>304</td>
<td>603</td>
</tr>
<tr>
<td>jack-4</td>
<td>18</td>
<td>79</td>
<td>4.39</td>
<td>0</td>
<td>304</td>
<td>759</td>
</tr>
<tr>
<td>jack-5</td>
<td>57</td>
<td>151</td>
<td>2.65</td>
<td>0</td>
<td>339</td>
<td>647</td>
</tr>
<tr>
<td>jack-6</td>
<td>35</td>
<td>132</td>
<td>3.77</td>
<td>0</td>
<td>338</td>
<td>802</td>
</tr>
<tr>
<td>jack-7</td>
<td>35</td>
<td>132</td>
<td>3.77</td>
<td>0</td>
<td>338</td>
<td>802</td>
</tr>
<tr>
<td>jack-8</td>
<td>35</td>
<td>132</td>
<td>3.77</td>
<td>0</td>
<td>338</td>
<td>802</td>
</tr>
<tr>
<td>jack-9</td>
<td>30</td>
<td>79</td>
<td>2.63</td>
<td>0</td>
<td>304</td>
<td>759</td>
</tr>
<tr>
<td>jack-10</td>
<td>57</td>
<td>151</td>
<td>2.65</td>
<td>0</td>
<td>339</td>
<td>647</td>
</tr>
</tbody>
</table>
Threats to validity

- Uses injected bugs from the SIR suite
- Use BFS traversal to simulate user process
- Use of whole-program pointer analysis and call graph construction for the thin slicer may not scale to larger benchmarks
Conclusions

- Thin slices lead the user to desired statements
- Thin slices focus better on desired statements than traditional slices
- A precise pointer analysis is key to effective thin slicing
- Thin slices can be computed efficiently: context-insensitive thin slicing algorithm scaled well to large programs
Questions?

Thanks!