Finding Bugs is Easy [2]

Overview

• Motivation
• Problem
• Approach
• Experiments
Motivation

- Bugs are a serious problem
- Many techniques developed to automatically find bugs
  - Formal methods
  - Sophisticated program analysis
- Existing techniques are difficult to apply, and aren't always effective in finding real bugs

Problem

- How to detect bugs using simple and broad techniques, rather than focused and narrow techniques?
Pattern-based Bug Detection

- Bug patterns are code idioms that are often errors
- Start by looking at actual bugs in real code, extract the bug patterns, and then develop bug pattern detectors to find similar bugs

Bug Pattern Detectors

- 45 bug pattern detectors are implemented using BCEL
- Four categories of detectors:
  - Single-threaded correctness issue
  - Thread/synchronization correctness issue
  - Performance issue
  - Security and vulnerability to malicious untrusted code
Four categories of implementation strategies

• Class structure and inheritance hierarchy only
  – Some of the detectors simply look at the structure of analyzed classes without looking at the code
  – E.g., equals() and hashcode() should be defined together

• Linear code scan
  – No control flow analysis
  – E.g., bad covariant definition of equals:
    public boolean equals(Foo obj) {...}

• Control sensitive
  – Control flow analysis
  – E.g., WaitNotInLoop:
    • Object.wait() method waits on a monitor for another thread to call notify() or notifyAll()
    • Usually, wait() is waiting for a particular condition to become true
    • The most robust way is to put it in a loop, where the waited-for condition is checked each time the thread wakes up
Four categories of implementation strategies

• Data flow
  – Control and data flow analysis
  – E.g., null pointer dereference
    
    ```java
    if (foo == null)
    {
        ...
        foo.f ...
    }
    ```

Selected Bug Pattern Detectors
Inconsistent Synchronization

• Detect fields which are sometimes accessed with a self lock held and sometimes without are candidate instances
• Several heuristics are used to reduce the number of false positives
  – Public or volatile fields are ignored
  – Fields that are never read without a lock are ignored (?)

Inconsistent Synchronization

• Heuristics to reduce false positives (cont’d)
  – Accesses in object lifecycle methods (such as constructors and finalizers), or in nonpublic methods reachable only from these lifecycle methods, are ignored
  – If there is a high proportion of unlocked accesses (>=1/3), ignore it
    • $2(RU + 2WU) > (RL + 2WL)$
Static Field Modifiable by Untrusted Code

- Untrusted code is allowed to modify static fields, thereby modifying the behavior of the library for all uses
  - A static non-final field has public or protected access
  - A static final field has public or protected access, and references a mutable structure such as an array or Hashtable
  - A method returns a reference to a static mutable structure such as an array or Hashtable

Where are bug patterns from?

- Many of the bug patterns are suggested by Java semantics
- A number of books describe potential Java coding pitfalls
- Several bug patterns are observed in student projects and later implemented
- Several bug patterns are suggested by FindBugs users
Evaluation

• Run FindBugs on six applications
  – GNU Classpath, version 0.08
  – rt.jar from Sun JDK 1.5.0, build 59
  – Eclipse, version 3.0
  – DrJava, version stable-20040326
  – JBoss, version 4.0.0RC1
  – jEdit, version 4.2pre15

<table>
<thead>
<tr>
<th></th>
<th>classpath-0.06</th>
<th>rt.jar 1.5.0 build 18</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>warnings</td>
<td>serious</td>
</tr>
<tr>
<td>DC</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>IS2</td>
<td>18</td>
<td>72%</td>
</tr>
<tr>
<td>NP</td>
<td>7</td>
<td>85%</td>
</tr>
<tr>
<td>OS</td>
<td>9</td>
<td>22%</td>
</tr>
<tr>
<td>RR</td>
<td>7</td>
<td>100%</td>
</tr>
<tr>
<td>RV</td>
<td>11</td>
<td>45%</td>
</tr>
<tr>
<td>UIR</td>
<td>3</td>
<td>100%</td>
</tr>
<tr>
<td>UW</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>Wa</td>
<td>2</td>
<td>0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>eclipse-2.1.0</th>
<th>drjava-stable-20030822</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>warnings</td>
<td>serious</td>
</tr>
<tr>
<td>NP</td>
<td>43</td>
<td>93%</td>
</tr>
<tr>
<td>OS</td>
<td>16</td>
<td>6%</td>
</tr>
<tr>
<td>RR</td>
<td>22</td>
<td>4%</td>
</tr>
<tr>
<td>RV</td>
<td>9</td>
<td>100%</td>
</tr>
<tr>
<td>UIR</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>UW</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>jboss-3.2.2RC3</th>
<th>jedit-4.1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>warnings</td>
<td>serious</td>
</tr>
<tr>
<td>IS2</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>NP</td>
<td>10</td>
<td>100%</td>
</tr>
<tr>
<td>OS</td>
<td>2</td>
<td>100%</td>
</tr>
<tr>
<td>RR</td>
<td>0</td>
<td>0%</td>
</tr>
<tr>
<td>RV</td>
<td>2</td>
<td>0%</td>
</tr>
<tr>
<td>UIR</td>
<td>2</td>
<td>50%</td>
</tr>
<tr>
<td>UW</td>
<td>1</td>
<td>100%</td>
</tr>
<tr>
<td>Wa</td>
<td>0</td>
<td>0%</td>
</tr>
</tbody>
</table>
Interesting Observations

• No type of bug has been so “dumb” or “obvious” that we have failed to find examples of it in real code
• The potential for misuse of language features and APIs is enormous
• FindBugs can effectively raise the awareness of developers about subtle correctness issues

Reference