Reflection Analysis for Java

Benjamin Livshits, John Whaley, Monica S. Lam

Stanford University
Background: Bug Detection

- Our focus: bug detection tools
- Troubling observation: large portions of the program are not analyzed

Race conditions  Memory leaks  SQL injections
null dereferences  Resource usage errors  Cross-site scripting

Missing portions of the callgraph (# methods)
Reflection is to Blame

- Reflection is at the core of the problem
- Most analyses for Java ignore reflection
  - Fine approach for a while
  - SpecJVM hardly uses reflection at all
- Call graph is incomplete
  - Code not analyzed => bugs are missing
- Can no longer get away with this
  - Reflection is very common in Java: JBoss, Tomcat, Eclipse, etc. are reflection-based
  - Ignoring reflection misses ½ application & more
- Reflection is the proverbial white elephant: neglected issues nobody is talking about
Introduction to Reflection

- Reflection is a dynamic language feature
- Used to query object and class information
  - `static Class Class.forName(String className)`
    - Obtain a `java.lang.Class` object
    - I.e. `Class.forName("java.lang.String")` gets an object corresponding to class `String`
  - `Object Class.newInstance()`
    - Object constructor in disguise
    - Create a new object of a given class

```
Class c = Class.forName("java.lang.String");
Object o = c.newInstance();
```

- This makes a new empty string `o`
Running Example

- Most typical use of reflection:
  - Take a class name, make a `Class` object
  - Create object of that class, cast and use it

```
1. String className = ...;
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;        new T1();
      new T2();
...```

- Statically convert

```
Class.newInstance => new T()```
Other Reflective Constructs

- Object creation – most common idiom
- But there is more:
  - Access methods
  - Access fields
  - Constructor objects
- Please refer to the paper for more...
public void addHandlers(String path) {
    
    while (it.hasNext()) {
        XmlElement child = (XmlElement) it.next();
        String id = child.getAttribute("id");
        String clazz = child.getAttribute("class");

        AbstractPluginHandler handler = null;
        try {
            Class c = Class.forName(clazz);
            handler = (AbstractPluginHandler) c.newInstance();
            registerHandler(handler);
        } catch (ClassNotFoundException e) {
            
            registerHandler(handler);
        }
    }
}
Real-life Reflection Scenarios

- Real-life scenarios:
  - Specifying application extensions
    - Read names of extension classes from a file
  - Custom object serialization
    - Serialized objects are converted into runtime data structures using reflection
  - Code may be unavailable on a given platform
    - Check before calling a method or creating an object
    - Can be used to get around JDK incompatibilities

- Our 60-page TR has detailed case studies
Talk Outline

- Introduction to Reflection
- Reflection analysis framework
  - Possible analysis approaches to constructing a call graph in the presence of reflection
  - Pointer analysis-based approximation
  - Deciding when to ask for user input
  - Cast-based approximation
  - Overall analysis framework architecture
- Experimental results
- Conclusions
What to Do About Reflection?

1. String className = ...;
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;

1. Anything goes
   + Obviously conservative
   - Call graph extremely big and imprecise

2. Ask the user
   + Good results
   - A lot of work for user, difficult to find answers

3. Subtypes of T
   + More precise
   - T may have many subtypes

4. Analyze className
   + Better still
   - Need to know where className comes from
Analyzing Class Names

- Looking at `className` seems promising

```java
String stringClass = "java.lang.String";
foo(stringClass);
...

void foo(String clazz){
    bar(clazz);
}

void bar(String className){
    Class c = Class.forName(className);
}
```

- This is interprocedural const+copy prop on strings
StringClass
clazz
className

Stack variables

Heap objects

java.lang.String
Reflection Resolution Using Points-to

1. String className = ...;
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;

- Need to know what `className` is
  - Could be a local string constant like `java.lang.String`
  - But could be a variable passed through many layers of calls
- Points-to analysis says what `className` refers to
  - className --> concrete heap object
Reflection Resolution

Constants

Specification points

Class.forName(className)
Resolution May Fail!

1. String className = r.readLine();
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;

- Need help figuring out what className is
- Two options
  1. Can ask user for help
     - Call to r.readLine on line 1 is a specification point
     - User needs to specify what can be read from a file
     - Analysis helps the user by listing all specification points
  2. Can use cast information
     - Constrain possible types instantiated on line 3 to subclasses of T
     - Need additional assumptions
1. Specification Files

- Format: invocation site => class

loadImpl() @ 43 InetAddress.java:1231 => java.net.Inet4AddressImpl

loadImpl() @ 43 InetAddress.java:1231 => java.net.Inet6AddressImpl

lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.ISO_8859_15

lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.MS1251

tryToLoadClass() @ 29 DataFlavor.java:64 => java.io.InputStream
2. Using Cast Information

1. String className = ...;
2. Class c = Class.forName(className);
3. Object o = c.newInstance();
4. T t = (T) o;

- Providing specification files is tedious, time-consuming, error-prone
- Leverage cast data instead
  - o instanceof T
  - Can constrain type of o if
    1. Cast succeeds
    2. We know all subclasses of T
Analysis Assumptions

1. Assumption: Correct casts.
   Type cast operations that always operate on the result of a call to Class.newInstance are correct; they will always succeed without throwing a ClassCastException.

2. Assumption: Closed world.
   We assume that only classes reachable from the class path at analysis time can be used by the application at runtime.
Casts Aren’t Always Present

- Can’t do anything if no cast post-dominating a Class.newInstance call

```java
Object factory(String className) {
    Class c = Class.forName(className);
    return c.newInstance();
}
...

SunEncoder t = (SunEncoder)
    factory("sun.io.encoder." + enc);
SomethingElse e = (SomethingElse)
    factory("SomethingElse");
```
Call Graph Discovery Process

1. Program IR
2. Call graph construction
3. Reflection resolution using points-to
4. Resolved calls
5. Final call graph
6. User-provided spec
7. Specification points
8. Cast-based approximation
Juicy Implementation Details

- Call graph construction algorithm in the presence of reflection is integrated with pointer analysis
  - Pointer analysis already has to deal with virtual calls: new methods are discovered, points-to relations for them are created
  - Reflection analysis is another level of complexity
- Uses bddbddb, an efficient program analysis tool
  - Come to talk tomorrow
  - Rules are expressed in Datalog, see the paper
  - Rules that have to do with resolving method calls, etc. can get quite involved
  - Datalog makes experimentation easy
Talk Outline

- Introduction to Reflection
- Reflection analysis framework
- Experimental results
  - Benchmark information
  - Setup: 5 flavors of reflection analysis
  - Comparing...
    - Effectiveness of `Class.forName` resolution
    - Specification effort involved
    - Call graph sizes
- Conclusions
Experimental Summary

- Ran experiments on 6 very large applications in common use
- Compare the following analysis strategies:
  1. None -- no reflection resolution at all
  2. Local -- intraprocedural analysis
  3. Points-to -- relies on pointer analysis
  4. Casts -- points-to + casts
  5. Sound -- points-to + user spec
- Only version “Sound” is conservative
## Benchmark Information

- Among top Java apps on SourceForge
- Large, modern apps, not Spec JVM

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Description</th>
<th>Line count</th>
<th>File count</th>
<th>App Jars</th>
<th>Available classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>jgap</td>
<td>genetic algorithms package</td>
<td>32,961</td>
<td>172</td>
<td>9</td>
<td>62,727</td>
</tr>
<tr>
<td>freetts</td>
<td>speech synthesis system</td>
<td>42,993</td>
<td>167</td>
<td>19</td>
<td>62,821</td>
</tr>
<tr>
<td>gruntspud</td>
<td>graphical CVS client</td>
<td>80,138</td>
<td>378</td>
<td>10</td>
<td>63,847</td>
</tr>
<tr>
<td>jedit</td>
<td>graphical text editor</td>
<td>144,496</td>
<td>427</td>
<td>1</td>
<td>62,910</td>
</tr>
<tr>
<td>columba</td>
<td>graphical email client</td>
<td>149,044</td>
<td>1,170</td>
<td>35</td>
<td>53,689</td>
</tr>
<tr>
<td>jfreechart</td>
<td>chart drawing library</td>
<td>193,396</td>
<td>707</td>
<td>6</td>
<td>62,885</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>643,028</strong></td>
<td><strong>3,021</strong></td>
<td><strong>80</strong></td>
<td><strong>368,879</strong></td>
</tr>
</tbody>
</table>
Classification of Calls

Fully resolved

forName(className)

Partially resolved

forName(className)

Fully unresolved

forName(className)
Consider **Class.forName** resolution in jedit

- Some reflective calls don’t have targets on a given analysis platform
Reflective Calls with No Targets

// Class javax.sound.sampled.AudioSystem

private static final String defaultServicesClassName = "com.sun.media.sound.DefaultServices";

Vector getDefaultServices(String serviceName) {
    Vector v = null;
    try {
        Class defaultServices = Class.forName(defaultServicesClassName);
        Method m = defaultServices.getMethod(servicesMethodName, servicesParamTypes);
        Object[] arguments = new Object[] { serviceName };
        v = (Vector) m.invoke(defaultServices, arguments);
    } catch (InvocationTargetException e) {
        ...
    }
    return v;
}
Specification Effort

- Significantly less specification effort when starting from **Casts** compared to starting with **Points-to**

**Number of Class.forName calls requiring specification**

<table>
<thead>
<tr>
<th>Library</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
</tr>
</thead>
<tbody>
<tr>
<td>jgap</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>20</td>
<td>25</td>
<td></td>
</tr>
<tr>
<td>freetts</td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>gruntspud</td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jedit</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>columba</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
<td></td>
</tr>
<tr>
<td>jfreechart</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>15</td>
</tr>
</tbody>
</table>

Legend:
- **Libs**
- **App**

- points-to casts
- points-to casts
- points-to casts
- points-to casts
- points-to casts
- points-to casts
- points-to casts
- points-to casts
Specification is Hard

- Took us about 15 hours to provide specification for all benchmarks
- In many cases 2-3 iterations are necessary
  - More reflective calls are gradually discovered
  - More specification may be needed
- Fortunately, most unresolved calls are in library code
  - JDK, Apache, Swing, etc. have unresolved calls
  - Specifications can be shared among libraries
Call Graph Sizes

Call graph size (number of methods) compared

- None
- Local
- Points-to
- Casts
- Sound

jedit: 5,000 methods

Methods

jgap  freetts  gruntspud  jedit  columba  jfreechart
## Callgraph Sizes Compared: Sound vs None

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Classes</th>
<th>Methods</th>
</tr>
</thead>
<tbody>
<tr>
<td>jgap</td>
<td>(6x) 5.94</td>
<td>(7x) 6.58</td>
</tr>
<tr>
<td>freetts</td>
<td>4.58</td>
<td>4.05</td>
</tr>
<tr>
<td>gruntspud</td>
<td>2.21</td>
<td>1.96</td>
</tr>
<tr>
<td>jedit</td>
<td>(50% more) 1.66</td>
<td>(50% more) 1.43</td>
</tr>
<tr>
<td>columba</td>
<td>2.43</td>
<td>2.13</td>
</tr>
<tr>
<td>jfreechart</td>
<td>2.65</td>
<td>2.25</td>
</tr>
</tbody>
</table>
Related Work

- Call graph construction algorithms:
  - Function pointers in C [EGH94,Zha98,MRR01,MRR04]
  - Virtual functions in C++ [BS96,Bac98,AH96]
  - Methods in Java [GC01,GDDC97,TP00,SHR+00,ALS02,RRHK00]

- Reflection is a relatively unexplored research area
  - Partial evaluation [BN99,Ruf93,MY98]
    - “Compile reflection away”
    - Type constrains are provided by hand
  - Compiler frameworks accepting specification [TLSS99,LH03]
    - Can add user-provided edges to the call graph
  - Dynamic analysis [HDH2004]
    - Dynamic online pointer analysis that addresses dynamic class loading
Conclusions

- First call graph construction algorithm to explicitly deal with the issue of reflection
  - Uses points-to analysis for call graph discovery
  - Finds specification points
  - Casts are used to reduce specification effort

- Applied to 6 large apps, 190,000 LOC combined
  - About 95% of calls to `Class.forName` are resolved at least partially without any specs
  - There are some “stubborn” calls that require user-provided specification or cast-based approximation
  - Cast-based approach reduces the specification burden
  - Reflection resolution significantly increases call graph size: as much as 7X more methods, 7,000+ new methods