# Lecture 23: Static Analysis to Find Bugs

David Hovemeyer

November 30, 2022

601.428/628 Compilers and Interpreters

# Bugs in Software

- Bugs in software are a significant problem
- 2018 estimate of annual cost to US economy: \$2.84 trillion[1]
- Ways to find bugs before they enter production systems are needed
- What can we do?

[1] https://www.it-cisq.org/the-cost-of-poor-quality-software-in-the-us-a-2018-report/The-Cost-of-Poor-Quality-Software-in-the-US-2018-Report.pdf

# Testing

- Run the program, see if it behaves correctly
- Limitations:
  - Error handling code is difficult to test
  - Threading bugs can be very hard to reproduce
  - Test scaffolding is time-consuming to create

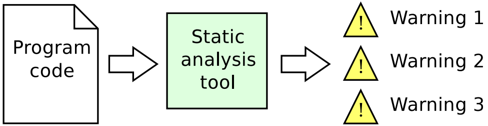
# Code inspection

- Manually examine source code, look for bugs
- Limitations:
  - Labor intensive
  - Subjective: source code might appear to be correct when it is is not
    - Can you spot the typo in this slide?
    - People have similar blind spots reading source code

# Code inspection

- Manually examine source code, look for bugs
- Limitations:
  - Labor intensive
  - Subjective: source code might appear to be correct when it is is not
    - Can you spot the typo in this slide?
    - People have similar blind spots reading source code

# Static analysis




- Idea: *automated code inspection*
- Use a program to analyze your program for bugs
  - Analyze statements, control flow, method calls
- Advantages over testing and manual code inspection:
  - Can analyze many potential program behaviors
  - Doesn't get bored
  - Relatively objective

# Limits of static analysis

- Nontrivial properties of programs are *undecidable*

$$\begin{aligned} & \text{“Does program } P \text{ have bug } X\text{?”} \\ \equiv\ & \text{“Can program } P \text{ reach state } X\text{?”} \\ \equiv\ & \text{Halting problem} \end{aligned}$$

- Static analysis can (in general) never be fully precise, so it must *approximate* the behavior of the program

# Approximating towards completeness

- We could design a bug-finding analysis so that it always overestimates possible program behaviors
  - *Never* misses a bug, but might report some false warnings
- Problem: the analysis may report so many false warnings that the real bugs cannot be found!
  - Trivial version: report a bug at every point in the program

# Approximating towards soundness

- We could design a bug-finding analysis so that it always underestimates possible program behaviors
  - Never reports a false warning, but might miss some real bugs
- Problem: analysis may not find as many bugs as we would like
  - Trivial version: never report any warnings

# Heuristic analysis

- A static analysis to find bugs does not need to be *consistent* in its approximations
  - Neither complete nor sound: miss some real bugs, and report some false warnings
- This gives the analysis the flexibility to estimate *likely* program behaviors
- May allow the analysis to be more precise in general

# Practical issues

- Say your program has 100 real bugs
- Would you rather use
  - A tool that finds all 100 bugs, but reports 1,000,000 warnings
  - A tool that finds only 25 bugs, but reports 50 warnings
- Using a bug-finding tool must be a productive use of the developer's time
- In general, no useful tool will find *every* bug

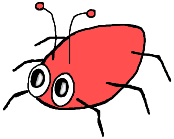
# Bug patterns

# Bug patterns

- Not all bugs are subtle and unique
- Many bugs share common characteristics
- A *bug pattern* is a code idiom that is usually a bug
  - Detection of many bug patterns can be automated using simple analysis techniques

# The FindBugs tool

- FindBugs:
  - Open source
  - https://findbugs.sourceforge.net
  - Implements detectors for 50+ bug patterns
  - No longer maintained: successor project is SpotBugs https://spotbugs.github.io/
- Analyzes Java bytecode
  - Bytecode is the machine language for the Java Virtual Machine
  - Easier to analyze than source code



# Null pointer bugs

# Null pointer bugs

- In Java, a reference value can be `null`
- If such a reference is *dereferenced*, a `NullPointerException` is thrown
  - Default behavior: the thread performing the operation is abruptly terminated
- Examples of dereferences:
  - Call an instance method (`x.foo()`)
  - Load a value from a field (`sum += x.count`)
  - Store a value to a field (`x.count = 42`)
  - Load a value from an array element (`sum += x[i]`)
  - Store a value to an array element (`x[i] = 17`)
  - Check the length of an array (`i < x.length`)

# Example null pointer bug

- Apache Ant 1.6.2, org.apache.tools.ant.taskdefs.optional.metamata.MAudit

```
if (out == null) {
    try {
        out.close();
    } catch (IOException e) {
    }
}
```

# Example null pointer bug

- Eclipse 3.0.1, org.eclipse.update.internal.core.ConfiguredSite

```
if (in == null)
    try {
        in.close();
    } catch (IOException e1) {
    }
```

# Example null pointer bug

- Eclipse 3.0.1, org.eclipse.jdt.internal.debug.ui.JDIModelPresentation

```
if (sig != null || sig.length() == 1) {
  return sig;
}
```

# Example null pointer bug

- Eclipse 3.0.1, org.eclipse.jdt.internal.ui.compare.JavaStructureDiffViewer

```
Control c= getControl();
if (c == null && c.isDisposed())
  return;
```

# Example null pointer bug

- From JBoss 4.0.0RC1

```
public String getContentId()
    String[] header = getMimeHeader("Content-Id");
    String id = null;
    if( header != null || header.length > 0 )
        id = header[0];
    return id;
}
```

# Null pointer dereferences

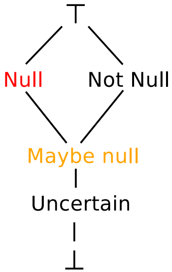
- Some null pointer deferences require sophisticated analysis to find
  - Analyzing across method calls, modeling the contents of heap objects
- We have seen many examples of *obvious* null pointer dereferences
  - Often arising from simple mistakes, such as using the wrong boolean operator
- How can we construct an analysis to find obvious null pointer dereferences?
  - Values which are always null
  - Values which were null on some control path

# Dataflow analysis

- At each point in a method, keep track of *dataflow facts*
  - E.g., which local variables and stack locations might contain null
- Symbolically execute the method:
  - Model instructions
  - Model control flow
  - Iterate until a fixed point solution is reached

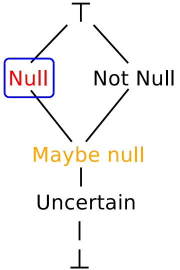
# Dataflow values

- Model values of local variables and stack operands using lattice of symbolic values
- When to control paths merge, use *meet* operator to combine values
  - This is the greatest lower bound of the values



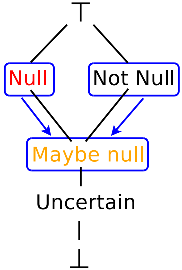
# Meet example

$$\text{Null} \diamond \text{Null} = \text{Null}$$



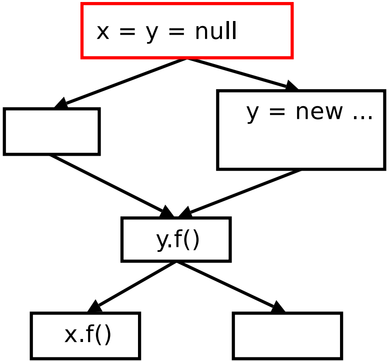
# Meet example

Null $\diamond$ Not null $=$ Maybe null



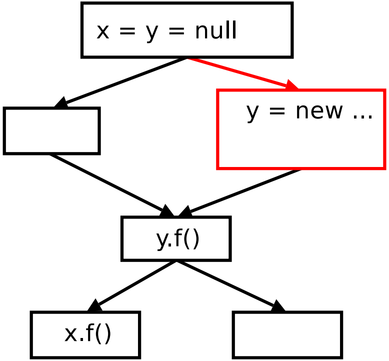
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



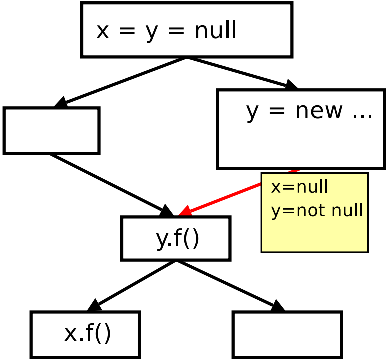
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



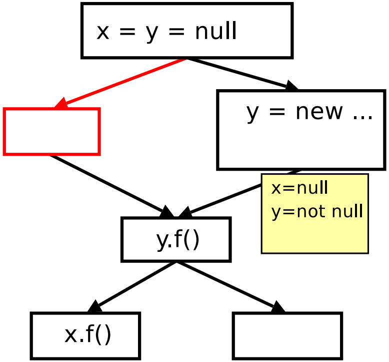
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



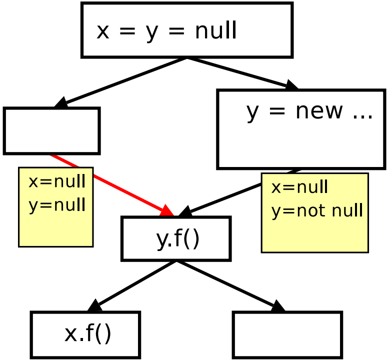
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



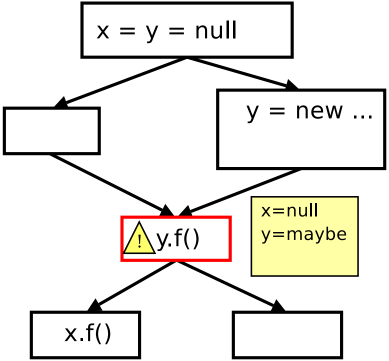
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



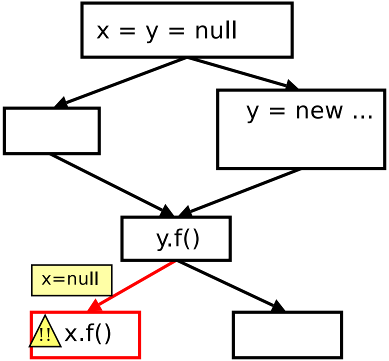
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



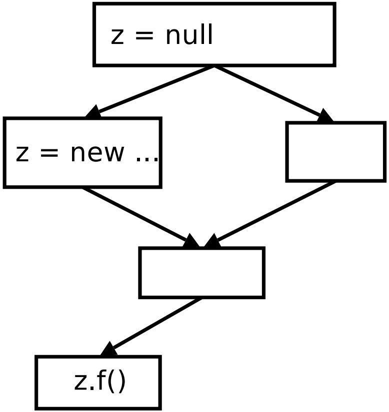
# Null-pointer dataflow example

```
x = y = null;
if (!cond) {
  y = new ...
}
y.f();
if (cond2)
  x.f();
else
  ;
```



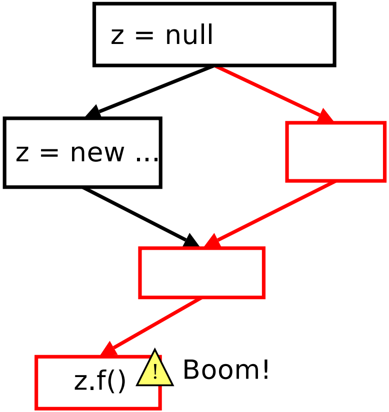
# Null-pointer dataflow example

```
z = null;
if (cond) {
   z = new ...
}

if (cond2)
   z.f();
```



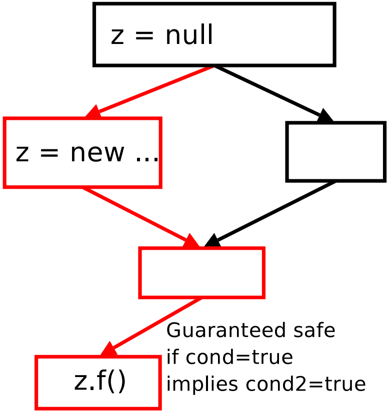
# Null-pointer dataflow example

```
z = null;
if (cond) {
  z = new ...
}

if (cond2)
  z.f();
```



# Null-pointer dataflow example

```
z = null;
if (cond) {
   z = new ...
}

if (cond2)
   z.f();
```



# Issue: Correlated Conditionals

- Not every path through a control flow graph is necessarily feasible
  - The outcome of an earlier conditional may determine the outcome of a later conditional
- This can cause lots of false positives!
- Our approach:
  - Only report all NPEs that would occur given full statement coverage or full branch coverage
  - “Maybe” values changed to “Uncertain” on conditional branches

# More sophisticated approach

- The issues found by the approach just described are highly likely to be real issues
- But, the loss of precision when there are paths with multiple conditional branches means that some real bugs are missed

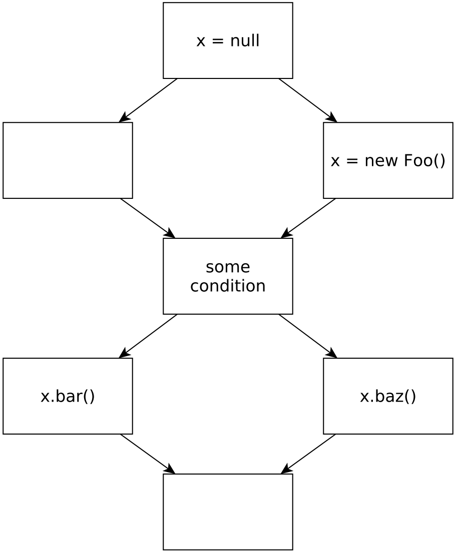
# A missed null pointer bug

```
// In Apache Tomcat 4.1.24

HttpServletRequest hreq = null;
if (req instanceof HttpServletRequest)
  hreq = (HttpServletRequest) req;

if (isResolveHosts())
  result.append(req.getRemoteHost());
else
  result.append(req.getRemoteAddr());
...
result.append(hreq.getMethod());
```

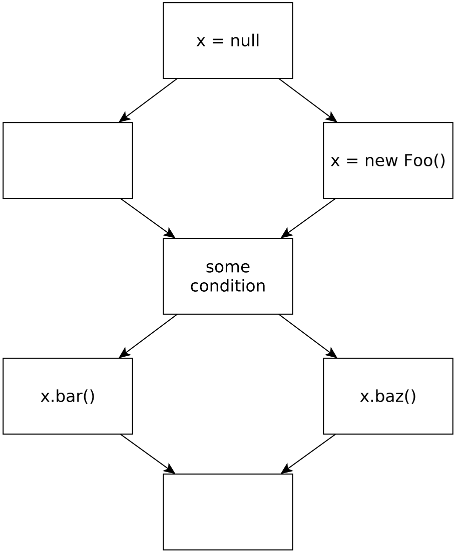
# More sophisticated analysis

- Idea: add a backwards analysis to determine where in a method reference values are guaranteed to be dereferenced
  - Very similar to liveness analysis! Main difference is that we only consider dereferences, which are a subset of uses
- Compare the results of the guaranteed dereference analysis with the results of the nullness analysis
- If we find a location where a value which is definitely null or “null on a simple path” is guaranteed to be dereferenced, report a warning

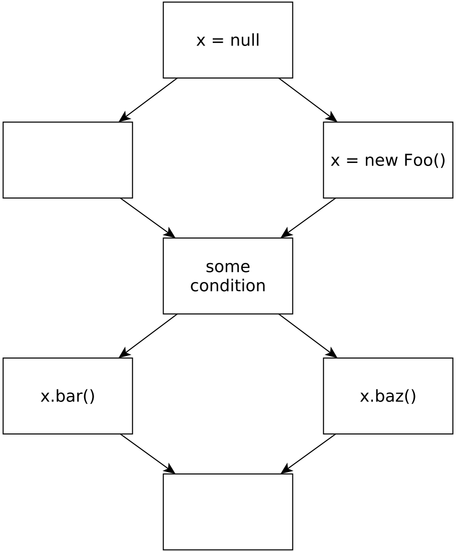
# Guaranteed dereference example



# Guaranteed dereference example (nullness analysis)



# Guaranteed dereference example (guaranteed deref analysis)



# Conclusions

# Program analysis is useful!

- Program analysis techniques, such as dataflow analysis, are useful for more than just compiler optimization
- Many useful tools have been built using this approach
  - Clang static analyzer
  - Coverity Scan
  - Many others
- Static analysis is not a silver bullet
  - But, can be a useful complement to other techniques for finding software defects