# Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study

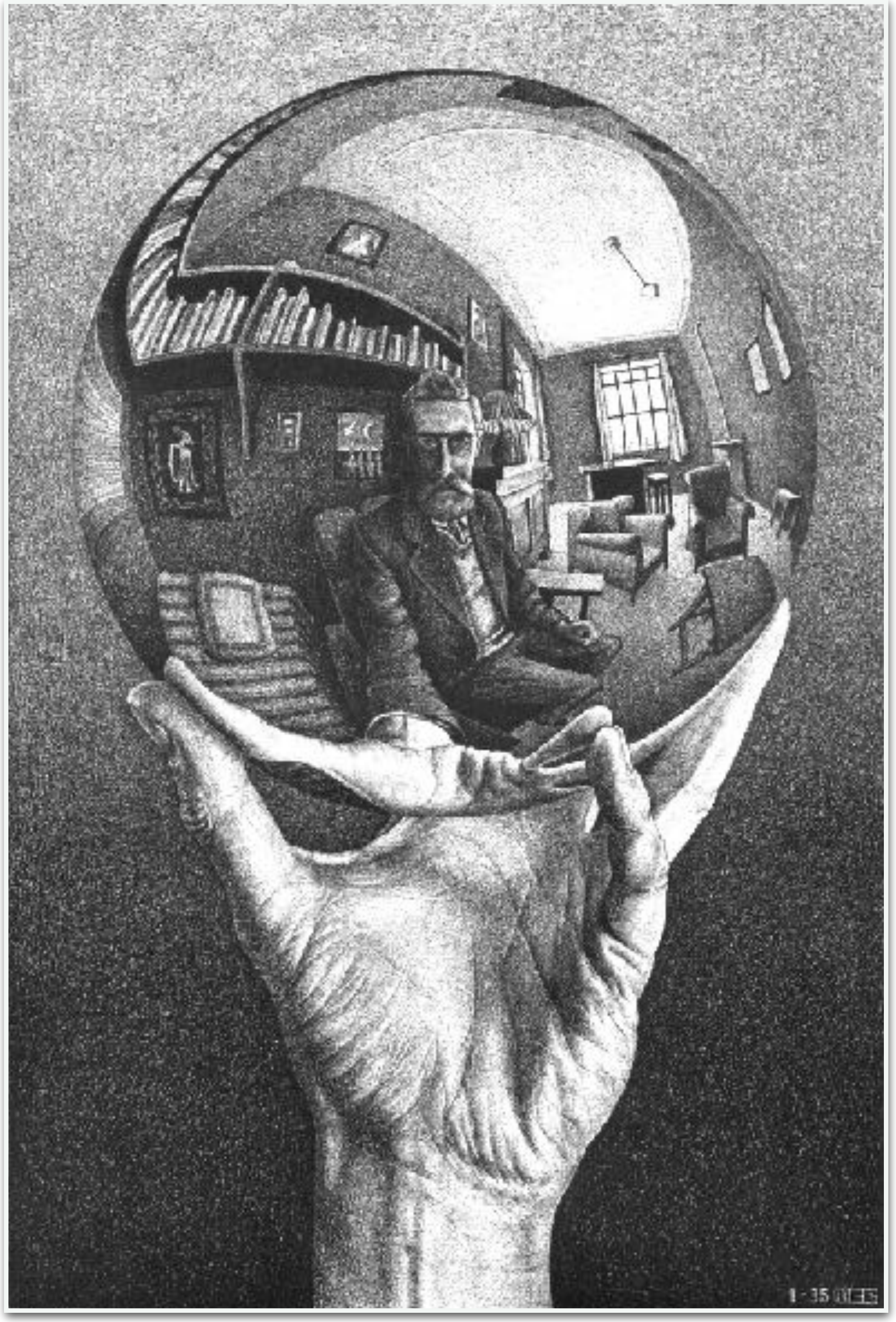
Davy Landman[*], Alexander Serebrenik[*][†], Jurgen J. Vinju[*][†]
[*] Centrum Wiskunde & Informatica, Amsterdam, The Netherlands
{Davy.Landman, Jurgen.Vinju}@cwi.nl
[†] Eindhoven University of Technology, Eindhoven, The Netherlands
{a.serebrenik,j.j.vinju}@tue.nl

CWI
Centrum Wiskunde & Informatica

TU/e Technische Universiteit
Eindhoven
University of Technology

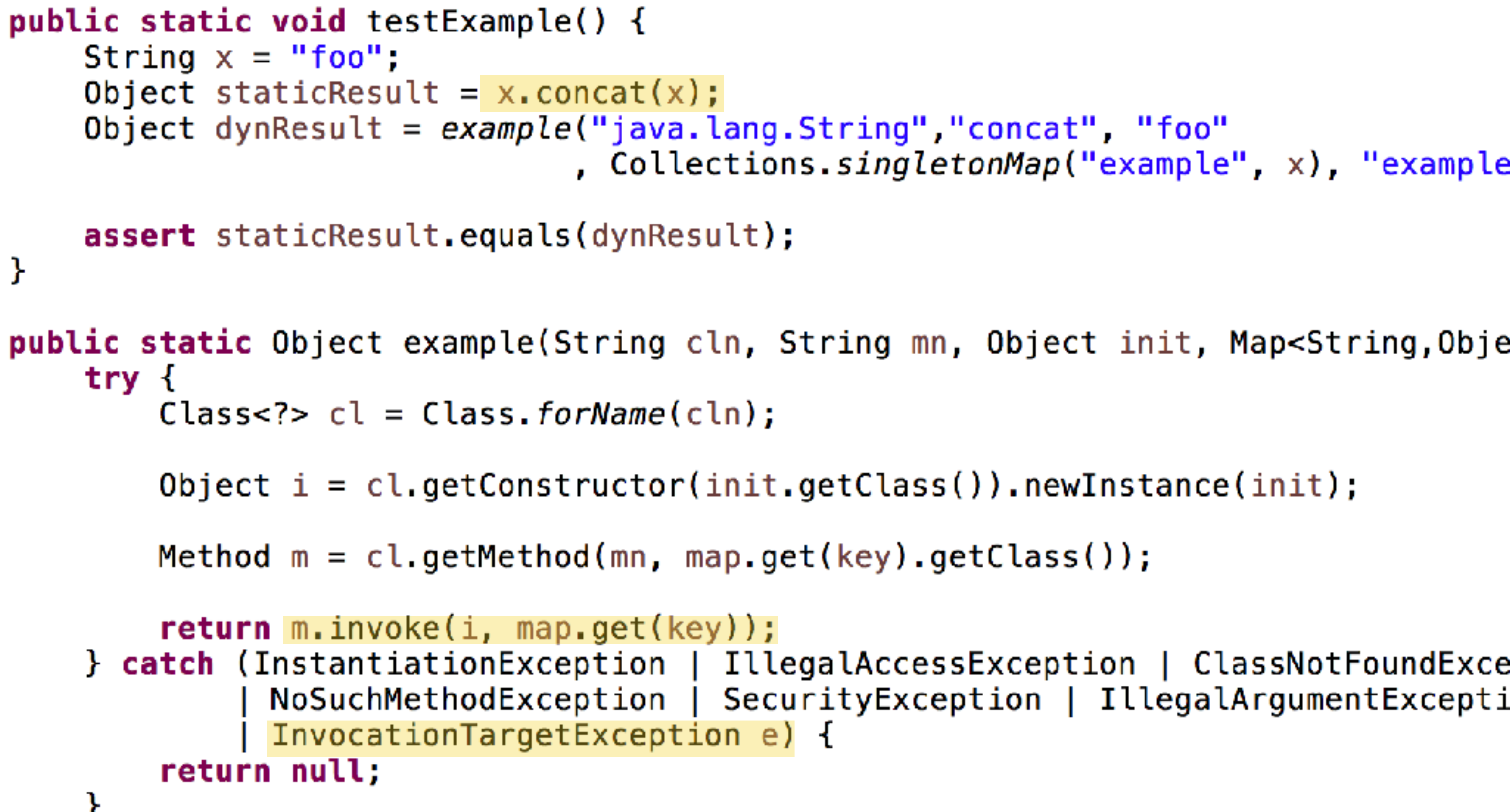
This work has been supported by the NWO TOPGO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”

Our initial context:
Is “reflection” going to be a
*problem*
if we want to harvest some
*(domain) knowledge*
from Java source code?

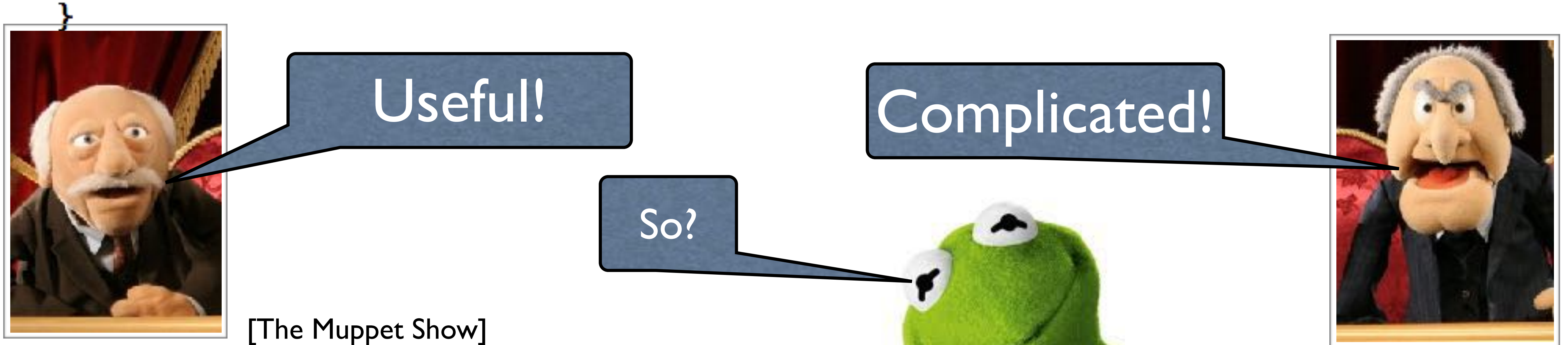


[MC Escher]

This work has been supported by the NWO TOPGO **grant # 612.001.011 “Domain-Specific Languages**: A Big Future for Small Programs”



```
public static void testExample() {
    String x = "foo";
    Object staticResult = x.concat(x);
    Object dynResult = example("java.lang.String","concat", "foo"
                              , Collections.singletonMap("example", x), "example

    assert staticResult.equals(dynResult);
}

public static Object example(String cln, String mn, Object init, Map<String,Obje
    try {
        Class<?> cl = Class.forName(cln);

        Object i = cl.getConstructor(init.getClass()).newInstance(init);

        Method m = cl.getMethod(mn, map.get(key).getClass());

        return m.invoke(i, map.get(key));
    } catch (InstantiationException | IllegalAccessException | ClassNotFoundExce
            | NoSuchMethodException | SecurityException | IllegalArgumentExcepti
            | InvocationTargetException e) {
        return null;
    }
}
```

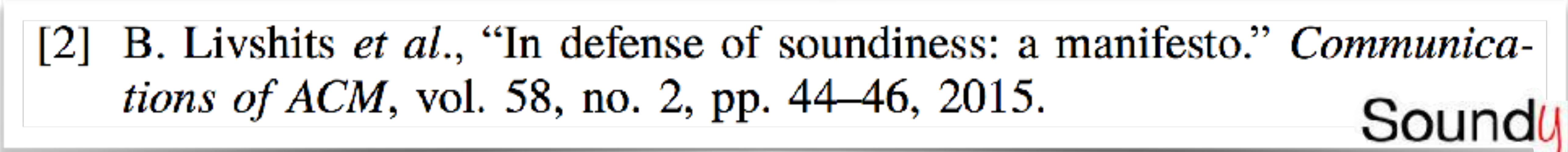


[The Muppet Show]

I **know** no general solution exists in **theory**!

You are both **biased** we could use **empirical** evidence…

But!! I **can design** tools which **work** on these categories



[2] B. Livshits *et al.*, "In defense of soundiness: a manifesto." *Communications of ACM*, vol. 58, no. 2, pp. 44–46, 2015.

**Research Question:** *What are limits of state-of-the-art static analysis tools when confronted with the Reflection API and how do these limits relate to real Java code?*

[Raphael, School of Athens]

## Actionable results

- Researchers: *high impact* suggestions
- Practisioners: adapt code for *robustness*

## Empirical evidence

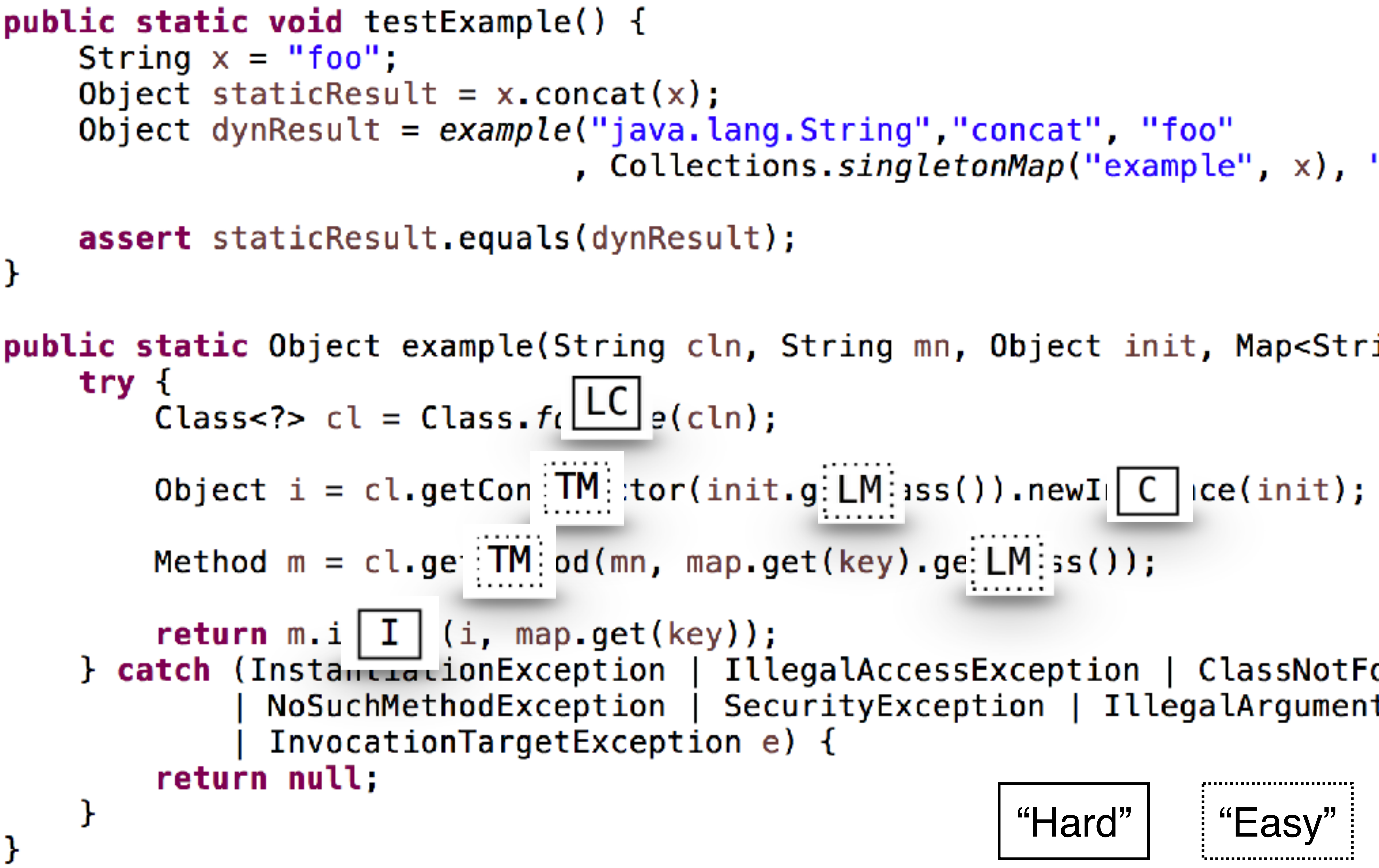
- Complex reflection is everywhere in Java
  - 462 Java projects in a ***representative* and clean corpus**
  - **78%** of Java *projects* have **hard** reflective code
- Known limitations have significant impact (**4% - 54%**)
  - Existing soundy assumptions ***validated***, more assumptions ***moti***

## Answers to research questions

1. What is Java reflection?
2. How often is Java reflection used, and how?
3. What do static analysis tools do to resolve reflection?
4. What are limitations of static analysis tools?
5. **How often does real Java code challenge limitations of static analysis?**

**WAR**
on
validity
threats
=
**M**
**E**
**T**
**H**
**O**
**D**
**S**

# Q1: What is Java reflection?



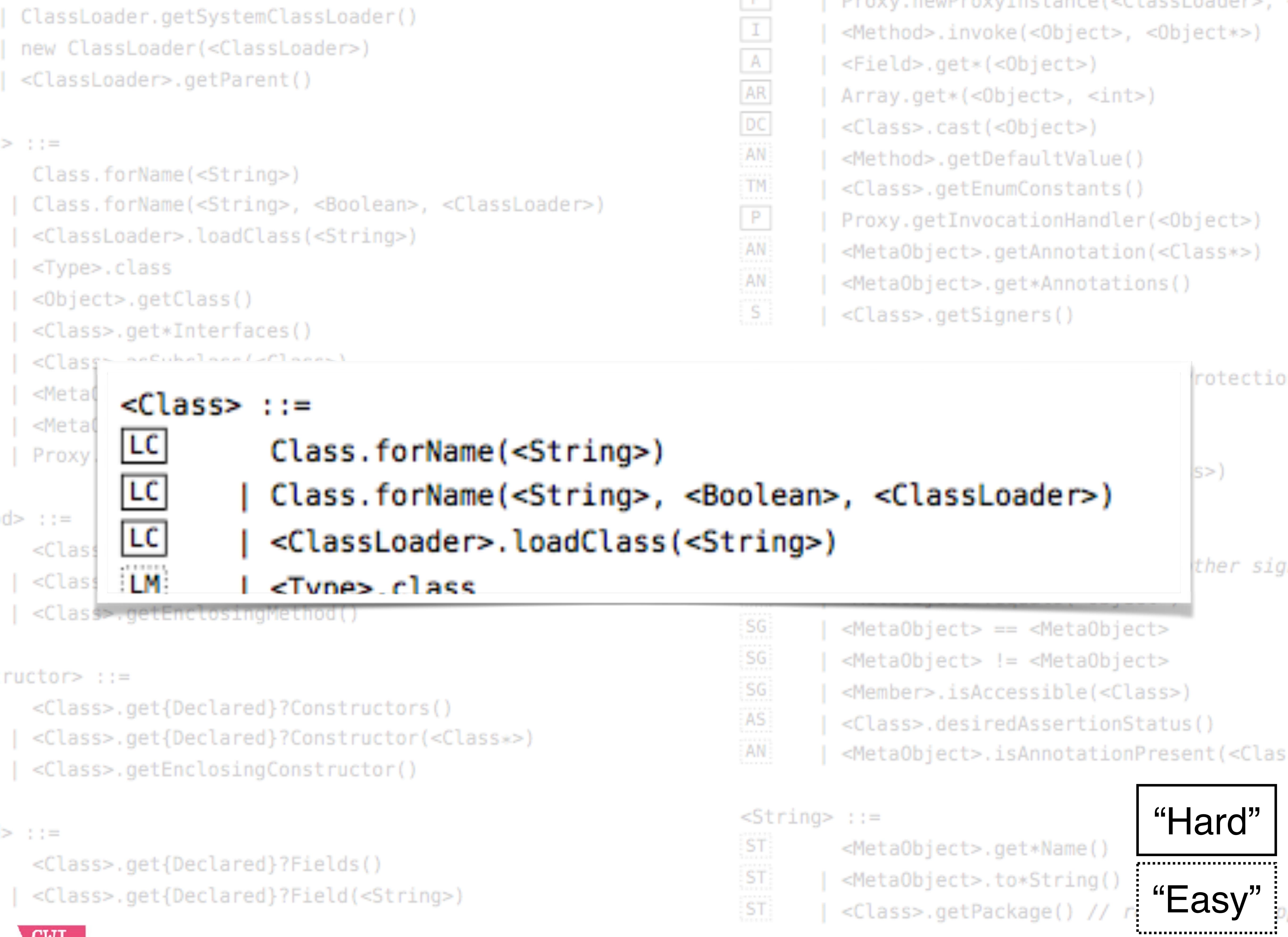
```
public static void testExample() {
    String x = "foo";
    Object staticResult = x.concat(x);
    Object dynResult = example("java.lang.String","concat", "foo"
                                , Collections.singletonMap("example", x), '

    assert staticResult.equals(dynResult);
}

public static Object example(String cln, String mn, Object init, Map<Stri
    try {
        Class<?> cl = Class.f LC e(cln);

        Object i = cl.getCon TM tor(init.g LM ass()).newI C ce(init);

        Method m = cl.ge TM od(mn, map.get(key).ge LM ss());

        return m.i I (i, map.get(key));
    } catch (Instan    tionException | IllegalAccessException | ClassNotFo
            | NoSuchMethodException | SecurityException | IllegalArgument
            | InvocationTargetException e) {
        return null;
    }
}
```

```
<MetaObject> ::= <Class> | <Method> | <Constructor> | <Field>
<Member> ::= <Method> | <Constructor> | <Field>

<ClassLoader> ::=
TM      <Class>.getClassLoader()
LM    | ClassLoader.getSystemClassLoader()
LM    | new ClassLoader(<ClassLoader>)
LM    | <ClassLoader>.getParent()

<Class> ::=
LC        Class.forName(<String>)
LC      | Class.forName(<String>, <Boolean>, <ClassLoader>)
LC      | <ClassLoader>.loadClass(<String>)
LM      | <Type>.class
LM      | <Object>.getClass()
TM      | <Class>.get*Interfaces()
TM      | <Class>.asSubclass(<Class>)
TM      | <MetaObject>.get*Class{es}?()
TM      | <MetaObject>.get*Type*()
P       | Proxy.getProxyClass(<Class*>)

<Method> ::=
TM        <Class>.get{Declared}?Methods()
TM      | <Class>.get{Declared}?Method(<String>, <Class*>)
TM      | <Class>.getEnclosingMethod()

<Constructor> ::=
TM        <Class>.get{Declared}?Constructors()
TM      | <Class>.get{Declared}?Constructor(<Class*>)
TM      | <Class>.getEnclosingConstructor()

<Field> ::=
TM        <Class>.get{Declared}?Fields()
TM      | <Class>.get{Declared}?Field(<String>)

<Void> ::=
M         <Field>.set*(<Object>, <Object>)
AR      | Array.set*(<Object>, <int>, <Object>)
MM      | <Member>.setAccessible(<Boolean>)
AS      | <ClassLoader>.{set}?{clear}?*AssertionStatus(<Boolean*>)
AS      | <ClassLoader>.set*AssertionStatus(<String>, <Boolean>)
```

```
<Object> ::=
C         <Constructor>.newInstance(<Object*>)
C       | <Class>.newInstance()
AR      | Array.newInstance*(<Class>, <int*>)
P       | Proxy.newProxyInstance(<ClassLoader>, <Class*>, <Object>)
I       | <Method>.invoke(<Object>, <Object*>)
A       | <Field>.get*(<Object>)
AR      | Array.get*(<Object>, <int>)
DC      | <Class>.cast(<Object>)
AN      | <Method>.getDefaultValue()
TM      | <Class>.getEnumConstants()
P       | Proxy.getInvocationHandler(<Object>)
AN      | <MetaObject>.getAnnotation(<Class*>)
AN      | <MetaObject>.get*Annotations()
S       | <Class>.getSigners()

<ProtectionDomain> ::= S <Class>.getProtectionDomain()

<Boolean> ::=
SG        <Class>.isAssignableFrom(<Class>)
SG      | <Class>.isInstance(<Class>)
SG      | Proxy.isProxyClass(<Class>)
SG      | <MetaObject>.is*(<Class>) // other signature checks
SG      | <MetaObject>.equals(<Object>)
SG      | <MetaObject> == <MetaObject>
SG      | <MetaObject> != <MetaObject>
SG      | <Member>.isAccessible(<Class>)
AS      | <Class>.desiredAssertionStatus()
AN      | <MetaObject>.isAnnotationPresent(<Class>)

<String> ::=
ST        <MetaObject>.get*Name()
ST      | <MetaObject>.to*String()
ST      | <Class>.getPackage() // returns a wrapper for strings

<int> ::= SG <MetaObject>.getModifiers()

<Resource> ::= <URL> | <InputStream>
RS      | <Class>.getResource*(<String>)
RS      | <ClassLoader>.get*Resource*(<String>)
```

“Hard”

“Easy”



```
| ClassLoader.getSystemClassLoader()
| new ClassLoader(<ClassLoader>)
| <ClassLoader>.getParent()

> ::=
  Class.forName(<String>)
| Class.forName(<String>, <Boolean>, <ClassLoader>)
| <ClassLoader>.loadClass(<String>)
| <Type>.class
| <Object>.getClass()
| <Class>.get*Interfaces()
| <Clas
| <Meta
| <Meta
| Proxy

d> ::=
  <Clas
| <Clas
| <Class>.getEnclosingMethod()

ructor> ::=
  <Class>.get{Declared}?Constructors()
| <Class>.get{Declared}?Constructor(<Class*>)
| <Class>.getEnclosingConstructor()

> ::=
  <Class>.get{Declared}?Fields()
| <Class>.get{Declared}?Field(<String>)
```

```
       | Proxy.newProxyInstance(<ClassLoader>,
I      | <Method>.invoke(<Object>, <Object*>)
A      | <Field>.get*(<Object>)
AR     | Array.get*(<Object>, <int>)
DC     | <Class>.cast(<Object>)
AN     | <Method>.getDefaultValue()
TM     | <Class>.getEnumConstants()
P      | Proxy.getInvocationHandler(<Object>)
AN     | <MetaObject>.getAnnotation(<Class*>)
AN     | <MetaObject>.get*Annotations()
S      | <Class>.getSigners()

         rotectio

         s>)

         ther sig
SG     | <MetaObject> == <MetaObject>
SG     | <MetaObject> != <MetaObject>
SG     | <Member>.isAccessible(<Class>)
AS     | <Class>.desiredAssertionStatus()
AN     | <MetaObject>.isAnnotationPresent(<Clas

<String> ::=
ST       <MetaObject>.get*Name()
ST     | <MetaObject>.toString()
ST     | <Class>.getPackage() // r
```

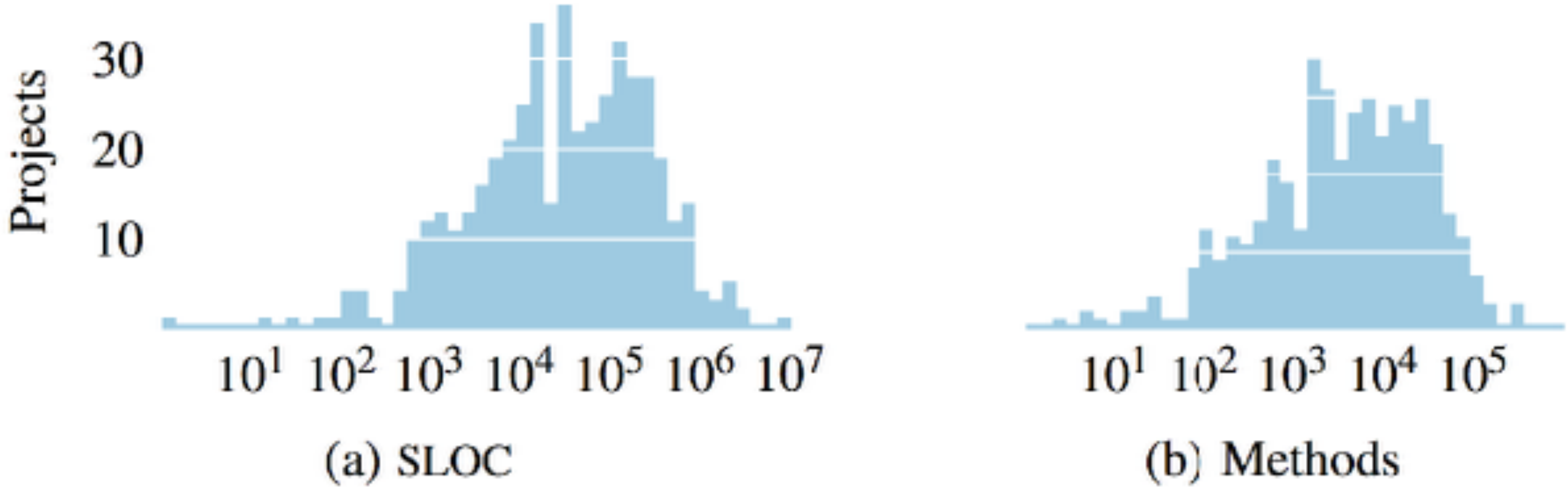
```
<Class> ::=
LC        Class.forName(<String>)
LC      | Class.forName(<String>, <Boolean>, <ClassLoader>)
LC      | <ClassLoader>.loadClass(<String>)
LM      | <Type>.class
```

“Hard”

“Easy”

# Q2: How often is reflection used?

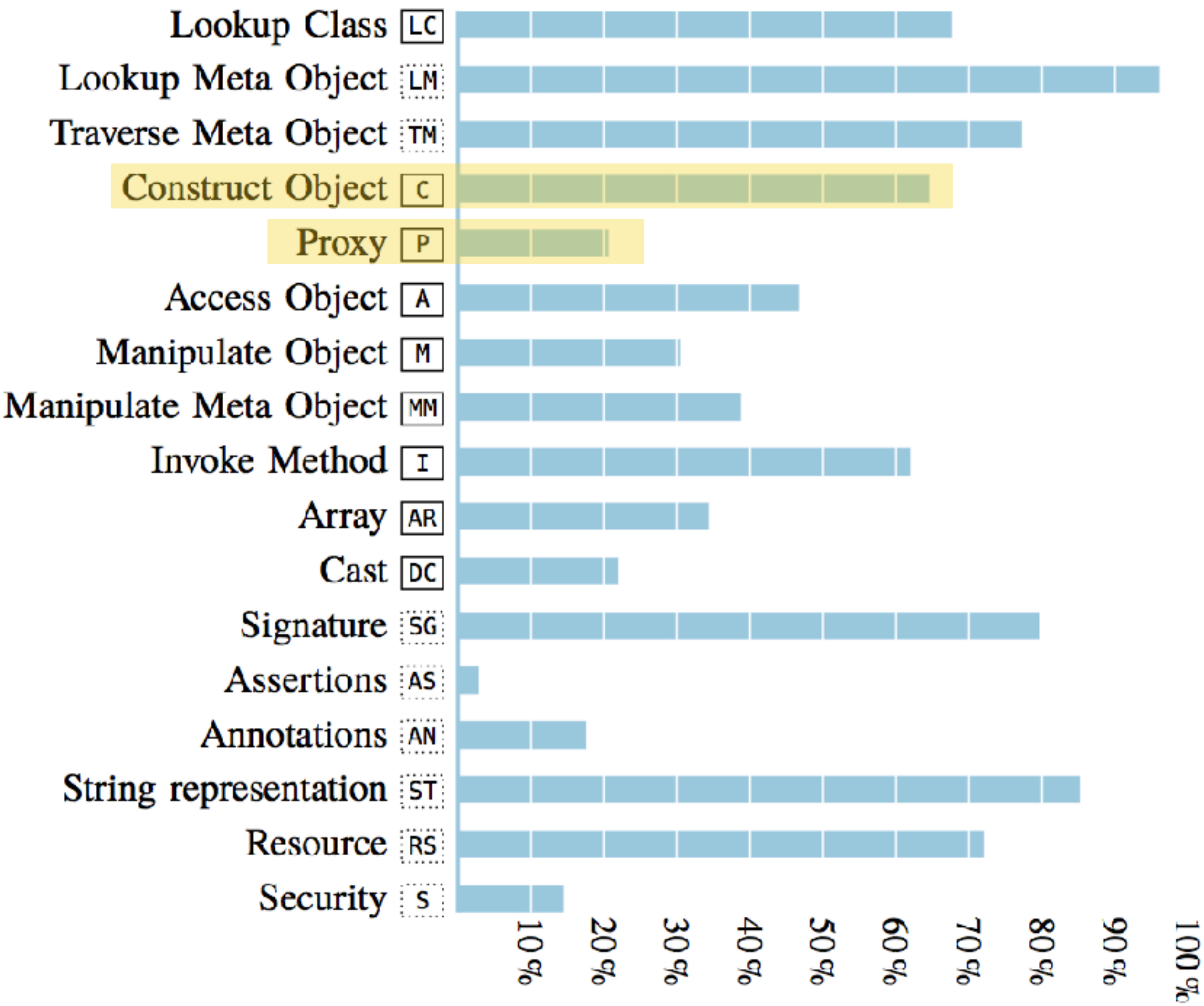
- Corpus of 461 (out of 3000) OSS Java projects:
  - Maximize representativeness [55]
  - Clean [*clone detection*]
  - Parse & resolve [*Rascal, Eclipse JDT*]
  - Categorize [see *Q1*]



(a) SLOC

(b) Methods

[55] M. Nagappan, T. Zimmermann, and C. Bird, “Diversity in software engineering research,” in *ESEC/FSE*. ACM, 2013, pp. 466–476.



of projects using reflection

# Q3: What do analysis tools do?

- Extended structured literature review
  - 4K pdf’s
  - Semi-automatic full text analysis
- Filtering from 4k via <u>514</u>, to 50 to <u>33</u> pdf’s
- Annotating
- Categorizing

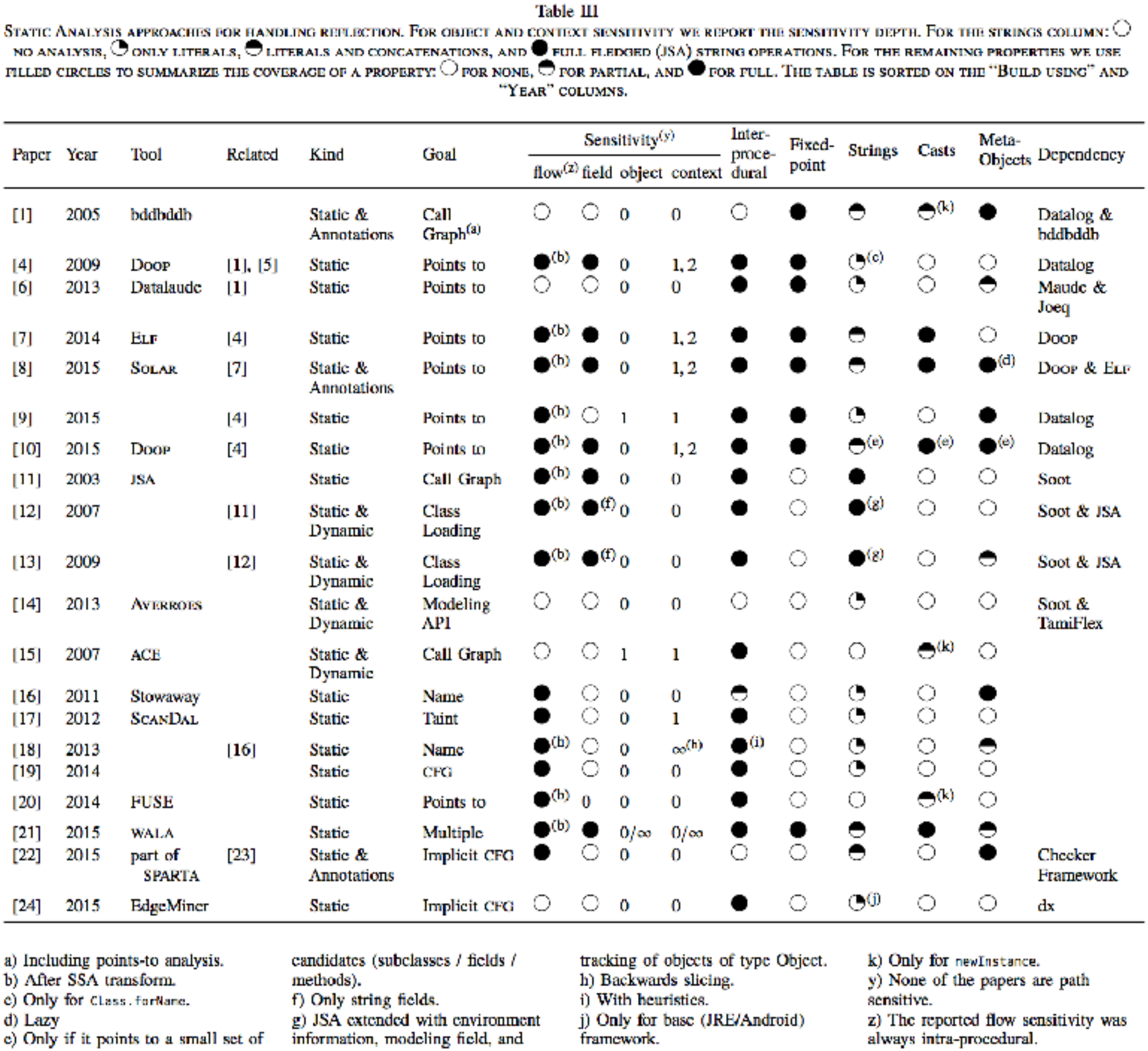


Table III

STATIC ANALYSIS APPROACHES FOR HANDLING REFLECTION. FOR OBJECT AND CONTEXT SENSITIVITY WE REPORT THE SENSITIVITY DEPTH. FOR THE STRINGS COLUMN: ○ NO ANALYSIS, ◔ ONLY LITERALS, ◓ LITERALS AND CONCATENATIONS, AND ● FULL FLEDGED (JSA) STRING OPERATIONS. FOR THE REMAINING PROPERTIES WE USE FILLED CIRCLES TO SUMMARIZE THE COVERAGE OF A PROPERTY: ○ FOR NONE, ◓ FOR PARTIAL, AND ● FOR FULL. THE TABLE IS SORTED ON THE "BUILD USING" AND "YEAR" COLUMNS.

| Paper | Year | Tool | Related | Kind | Goal | Sensitivity[y] | | | | Inter-procedural | Fixed-point | Strings | Casts | Meta-Objects | Dependency |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | | flow[z] | field | object | context | | | | | | |
| [1] | 2005 | bddbddb | | Static & Annotations | Call Graph[a] | ○ | ○ | 0 | 0 | ○ | ● | ◓ | ◓[k] | ● | Datalog & bddbddb |
| [4] | 2009 | Doop | [1], [5] | Static | Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◔[c] | ○ | ○ | Datalog |
| [6] | 2013 | Datalaude | [1] | Static | Points to | ○ | ○ | 0 | 0 | ● | ● | ◔ | ○ | ◓ | Maude & Joeq |
| [7] | 2014 | Elf | [4] | Static | Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◓ | ● | ○ | Doop |
| [8] | 2015 | Solar | [7] | Static & Annotations | Points to | ●[h] | ● | 0 | 1, 2 | ● | ● | ◓ | ● | ●[d] | Doop & Elf |
| [9] | 2015 | | [4] | Static | Points to | ●[h] | ○ | 1 | 1 | ● | ● | ◔ | ○ | ● | Datalog |
| [10] | 2015 | Doop | [4] | Static | Points to | ●[h] | ● | 0 | 1, 2 | ● | ● | ◓[e] | ●[e] | ●[e] | Datalog |
| [11] | 2003 | JSA | | Static | Call Graph | ●[h] | ● | 0 | 0 | ● | ○ | ● | ○ | ○ | Soot |
| [12] | 2007 | | [11] | Static & Dynamic | Class Loading | ●[b] | ●[f] | 0 | 0 | ● | ○ | ●[g] | ○ | ○ | Soot & JSA |
| [13] | 2009 | | [12] | Static & Dynamic | Class Loading | ●[b] | ●[f] | 0 | 0 | ● | ○ | ●[g] | ○ | ◓ | Soot & JSA |
| [14] | 2013 | Averroes | | Static & Dynamic | Modeling API | ○ | ○ | 0 | 0 | ○ | ○ | ◔ | ○ | ○ | Soot & TamiFlex |
| [15] | 2007 | ACE | | Static & Dynamic | Call Graph | ○ | ○ | 1 | 1 | ● | ○ | ○ | ◓[k] | ○ | |
| [16] | 2011 | Stowaway | | Static | Name | ● | ○ | 0 | 0 | ◓ | ○ | ◔ | ○ | ● | |
| [17] | 2012 | ScanDal | | Static | Taint | ● | ○ | 0 | 1 | ● | ○ | ◔ | ○ | ○ | |
| [18] | 2013 | | [16] | Static | Name | ●[b] | ○ | 0 | ∞[h] | ●[i] | ○ | ◔ | ○ | ◓ | |
| [19] | 2014 | | | Static | CFG | ● | ○ | 0 | 0 | ● | ○ | ◔ | ○ | ○ | |
| [20] | 2014 | FUSE | | Static | Points to | ●[b] | 0 | 0 | 0 | ● | ○ | ○ | ◓[k] | ○ | |
| [21] | 2015 | WALA | | Static | Multiple | ●[b] | ● | 0/∞ | 0/∞ | ● | ● | ◓ | ● | ◓ | |
| [22] | 2015 | part of SPARTA | [23] | Static & Annotations | Implicit CFG | ● | ○ | 0 | 0 | ○ | ○ | ◓ | ○ | ● | Checker Framework |
| [24] | 2015 | EdgeMiner | | Static | Implicit CFG | ○ | ○ | 0 | 0 | ● | ○ | ◔[j] | ○ | ○ | dx |

a) Including points-to analysis.
b) After SSA transform.
c) Only for `Class.forName`.
d) Lazy
e) Only if it points to a small set of candidates (subclasses / fields / methods).
f) Only string fields.
g) JSA extended with environment information, modeling field, and tracking of objects of type Object.
h) Backwards slicing.
i) With heuristics.
j) Only for base (JRE/Android) framework.
k) Only for `newInstance`.
y) None of the papers are path sensitive.
z) The reported flow sensitivity was always intra-procedural.

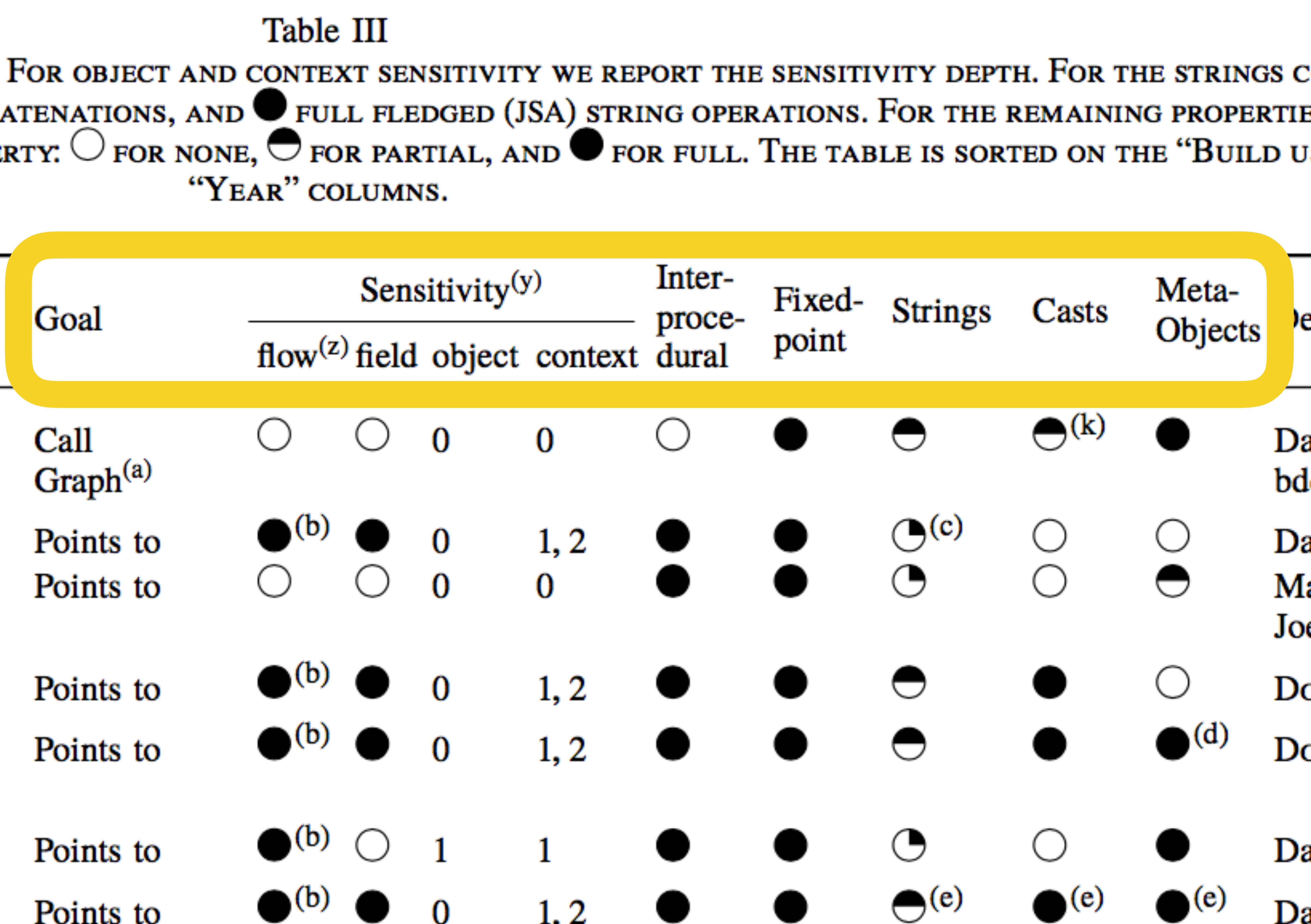


Table III

For object and context sensitivity we report the sensitivity depth. For the strings c
atenations, and ● full fledged (JSA) string operations. For the remaining propertie
rty: ○ for none, ◓ for partial, and ● for full. The table is sorted on the "Build u
"Year" columns.

| Goal | Sensitivity[y] | | | | Inter-procedural | Fixed-point | Strings | Casts | Meta-Objects | De |
|---|---|---|---|---|---|---|---|---|---|---|
| | flow[z] | field | object | context | | | | | | |
| Call Graph[a] | ○ | ○ | 0 | 0 | ○ | ● | ◓ | ◓[k] | ● | Da<br>bd |
| Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◔[c] | ○ | ○ | Da |
| Points to | ○ | ○ | 0 | 0 | ● | ● | ◔ | ○ | ◓ | Ma<br>Jo |
| Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◓ | ● | ○ | Do |
| Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◓ | ● | ●[d] | Do |
| Points to | ●[b] | ○ | 1 | 1 | ● | ● | ◔ | ○ | ● | Da |
| Points to | ●[b] | ● | 0 | 1, 2 | ● | ● | ◓[e] | ●[e] | ●[e] | Da |

# Q4: What are the limitations? and Q5: how do these relate to real code?

- Collect and categorize analysis papers self-reported:
  - Optimistic ‘soundy’ *assumptions* about code
  - Known *limitations* of the algorithms
  - What is their ***damage*** in the corpus?
- Method:
  - Recognize and count *counter examples*
  - Applying *AST patterns* to the entire corpus
  - Rascal metaprogramming language

[63] D. Landman, “cwi-swat/static-analysis-reflection,” https://doi.org/10.5281/zenodo.163326, Oct. 2016.

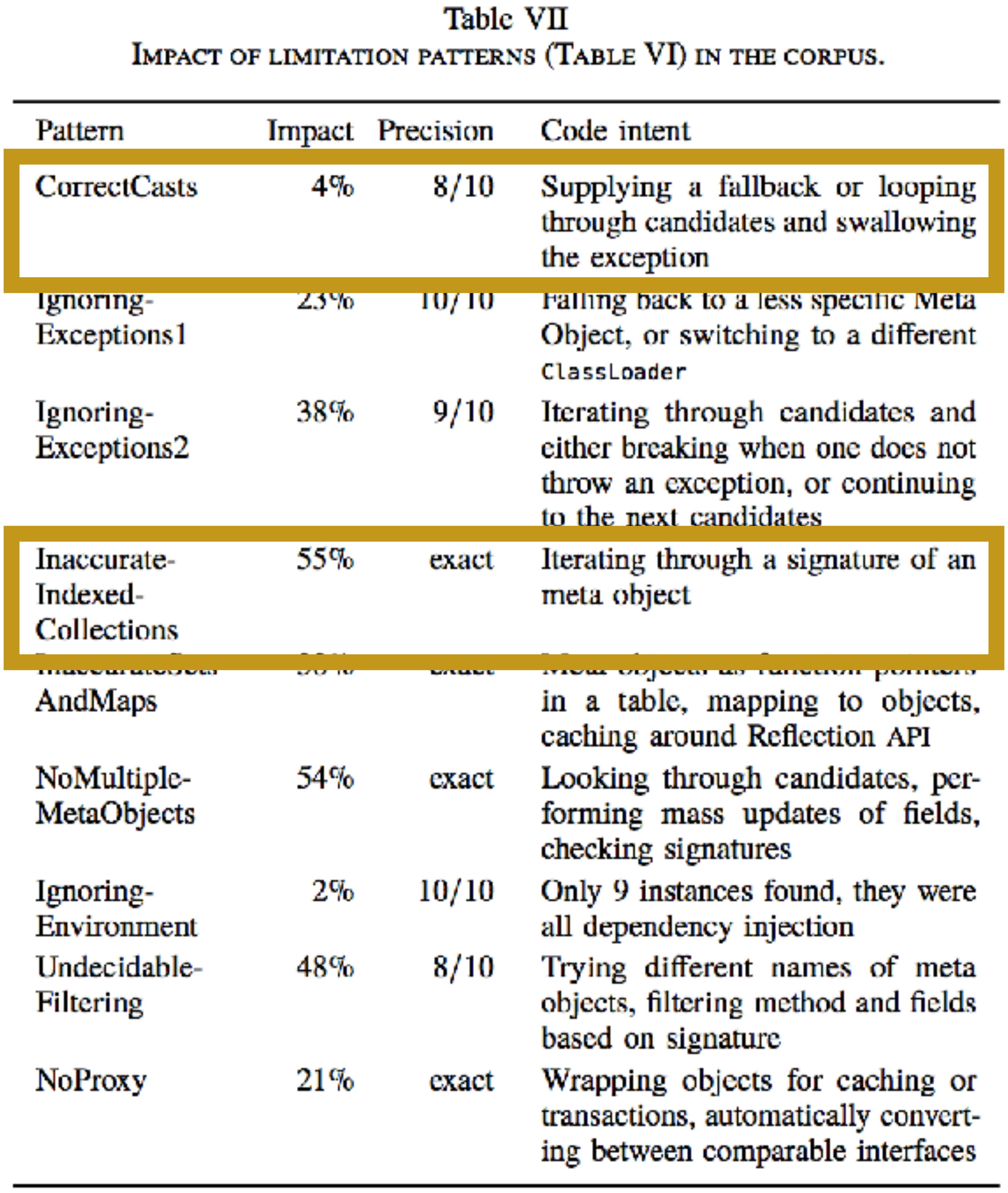


Table VII
IMPACT OF LIMITATION PATTERNS (TABLE VI) IN THE CORPUS.

| Pattern | Impact | Precision | Code intent |
|---|---|---|---|
| CorrectCasts | 4% | 8/10 | Supplying a fallback or looping through candidates and swallowing the exception |
| Ignoring-Exceptions1 | 23% | 10/10 | Falling back to a less specific Meta Object, or switching to a different `ClassLoader` |
| Ignoring-Exceptions2 | 38% | 9/10 | Iterating through candidates and either breaking when one does not throw an exception, or continuing to the next candidates |
| Inaccurate-Indexed-Collections | 55% | exact | Iterating through a signature of an meta object |
| InaccurateSets-AndMaps | 30% | exact | Meta objects as function pointers in a table, mapping to objects, caching around Reflection API |
| NoMultiple-MetaObjects | 54% | exact | Looking through candidates, performing mass updates of fields, checking signatures |
| Ignoring-Environment | 2% | 10/10 | Only 9 instances found, they were all dependency injection |
| Undecidable-Filtering | 48% | 8/10 | Trying different names of meta objects, filtering method and fields based on signature |
| NoProxy | 21% | exact | Wrapping objects for caching or transactions, automatically converting between comparable interfaces |

## Suggestions for static analysis researchers and Java language designers

1. Reflection API improvements to restrict arbitrary interactions (i.e. using lambdas)
2. Infer information from downcasts more aggressively
3. Make soundy assumptions about dynamic proxies: *the “oblivious wrapper proxy”*
4. Model common “goto patterns” with exceptions around reflection
5. Soundily assume boundedness and unorderedness of meta object collections
6. Apply dynamic language analysis techniques to methods which have reflection

## Advice for software engineers; make your code more robust *now*

1. Do not factor reflection into type polymorphic methods
2. Never use dynamic proxies
3. Use local variables/fields for meta object storage
4. Avoid loops over collections of meta objects
5. Test for preconditions instead of waiting for exceptions

# Challenges for Static Analysis of Java Reflection – Literature Review and Empirical Study

@davylandman @aserebrenik @jurgenvinju

Please use these artefacts for yourselves, or contact us for discussion about:

- the new soundy assumptions are a *prioritized work list* (*)
- the corpus is a way to **validate relevance** for new ideas in static analysis [3]
- tell us why we were wrong (replicate it) [63]

[63] D. Landman, “cwi-swat/static-analysis-reflection,” https://doi.org/10.5281/zenodo.163326, Oct. 2016.

[3] D. Landman, “A corpus of java projects representing the 2012 ohloh universe,” https://doi.org/10.5281/zenodo.162926, Mar 2016.

*To the authors of the static analysis papers, to the anonymous reviewers and to the members of IFIP WG 2.4 Software Implementation Technology, including Anders Møller*

This work has been supported by the NWO TOPGO grant #612.001.011 “Domain-Specific Languages: A Big Future for Small Programs”