Università
della
Svizzera
italiana

**Faculty
of Informatics**

Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM

Andrea Rosà, Eduardo Rosales, Walter Binder

Università della Svizzera italiana (USI), Faculty of Informatics, Lugano, Switzerland

GPCE 2017
October 23rd, 2017
Vancouver, Canada

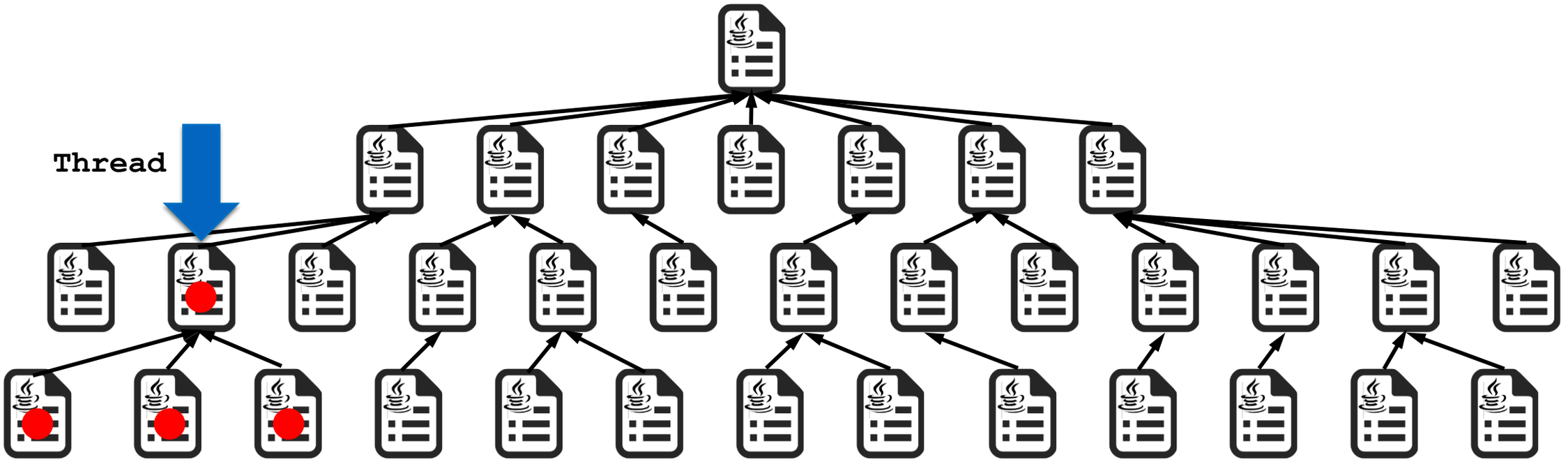
Reflective Information in the JVM

- Any information on a Java class or method available at runtime
  - Provided by the Java Reflection API
    - E.g.: `java.lang.Class`

- **Reflective supertype information (RSI)**
  - Information on all direct and indirect supertypes
    - `public Class<? super T> getSuperclass()`
    - `public Class<?>[] getInterfaces()`
    - Recursively applied

RSI during Instrumentation

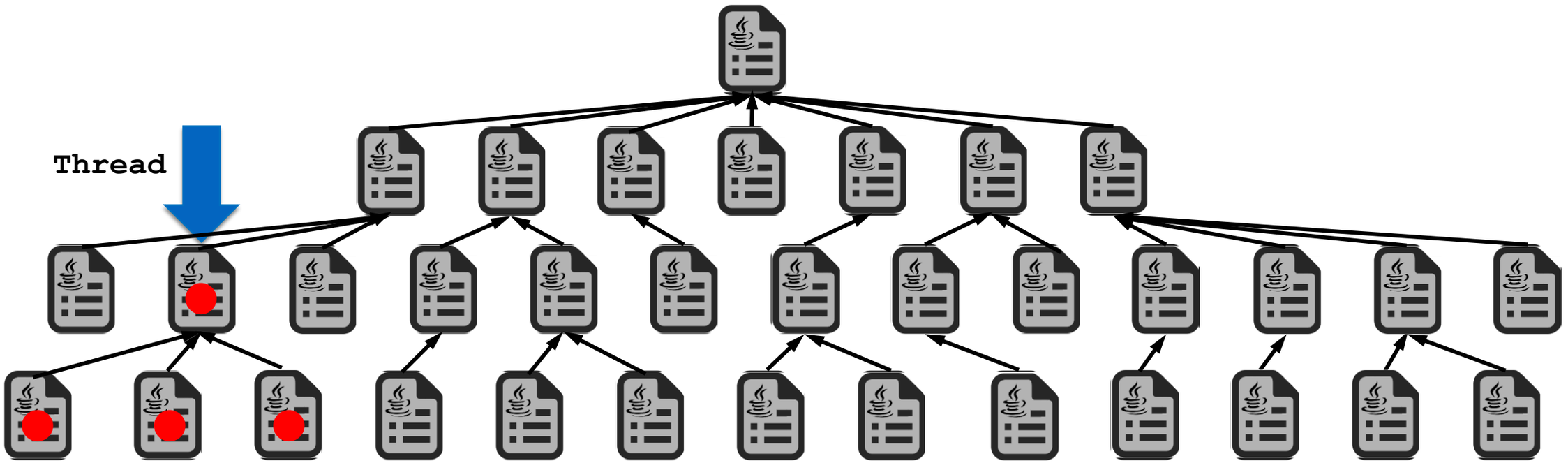
- Enables more efficient dynamic analyses
- Example: analysis targeting threads



- Without RSI:

RSI during Instrumentation

- Enables more efficient dynamic analyses
- Example: analysis targeting threads

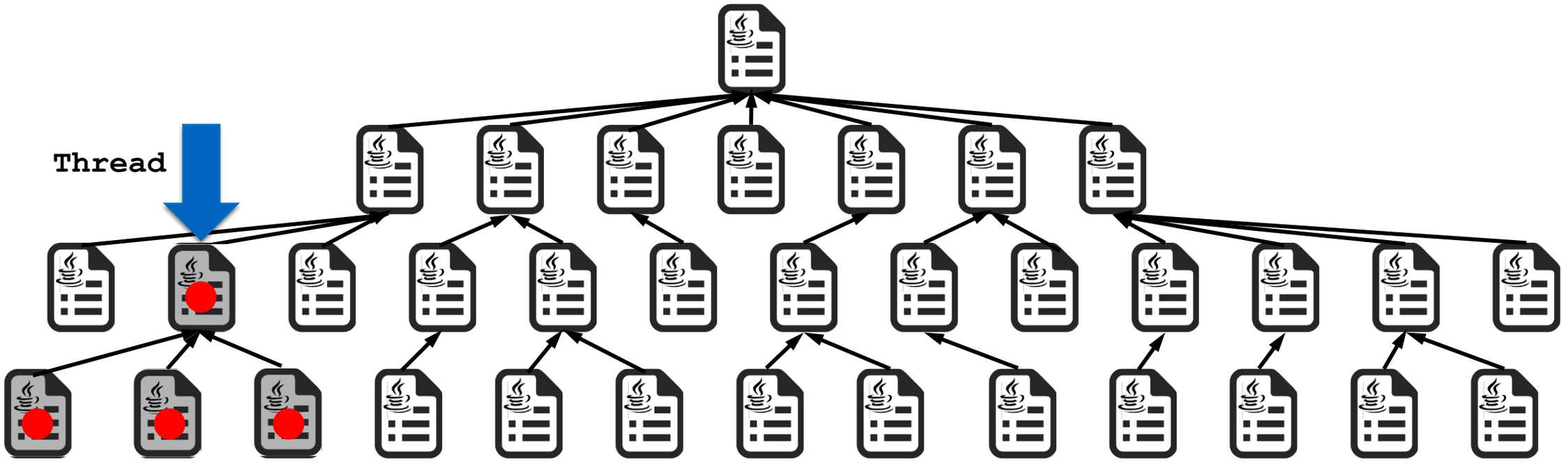


- **Without** RSI:
  - Many classes instrumented
  - Increased overhead

```
if (this instanceof Thread){
    // Do something
}
```

RSI during Instrumentation

- Enables more efficient dynamic analyses
- Example: analysis targeting threads

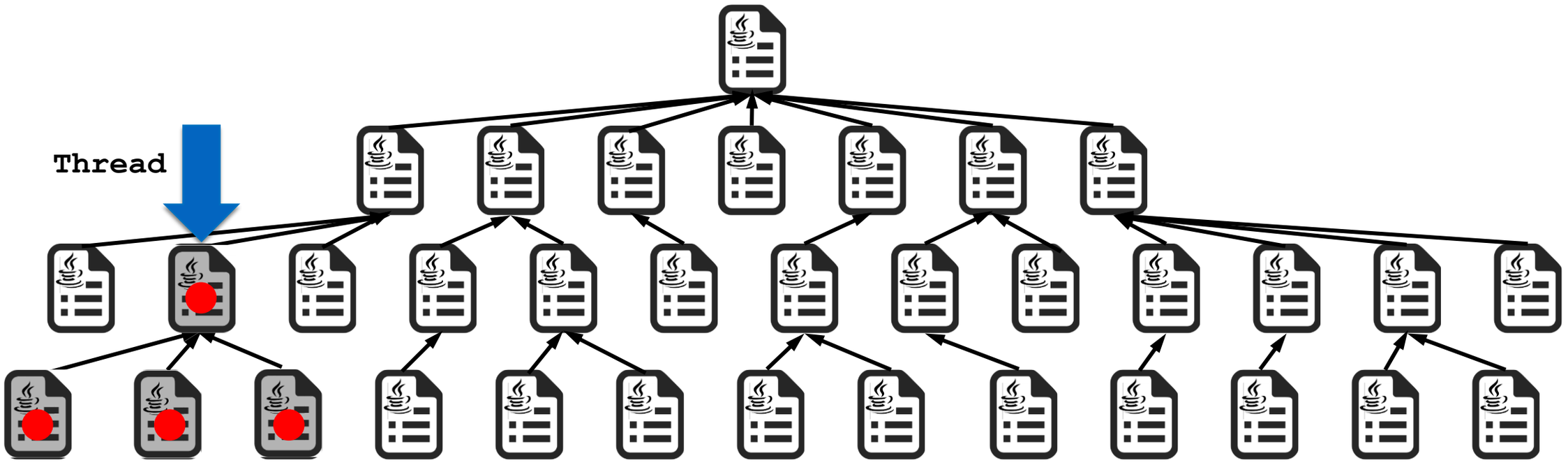


- With RSI:
  - Less classes instrumented
  - Less overhead

```
if (this instanceof Thread){
    // Do something
}
```

RSI during Instrumentation

- Enables more efficient dynamic analyses
- Example: analysis targeting threads



- With RSI:
  - Less classes instrumented
  - Less overhead

Availability of RSI
vs
Full code coverage

RSI during Instrumentation

Instrumentation	Complete RSI available?	Full code coverage?	Classloader namespaces?
Compile-time (e.g., AspectJ [1] compile-time weaver)	✗	✗	✗
Load-time in-process (e.g., AspectJ [1] load-time weaver)	✓	✗	✗
Load-time out-of-process (e.g., DiSL [2])	✗	✓	✗
Our approach (load-time out-of-process)	✓	✓	✓

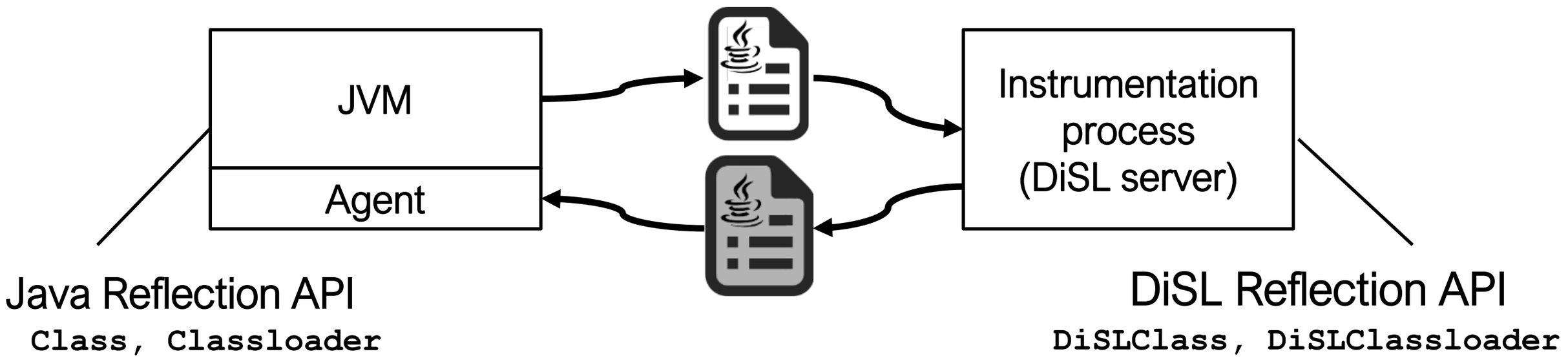
[1] Kiczales et al. *An overview of AspectJ*. ECOOP'01.

[2] Marek et al. DiSL: *A Domain-specific Language for Bytecode Instrumentation*. AOSD'12.

Contribution

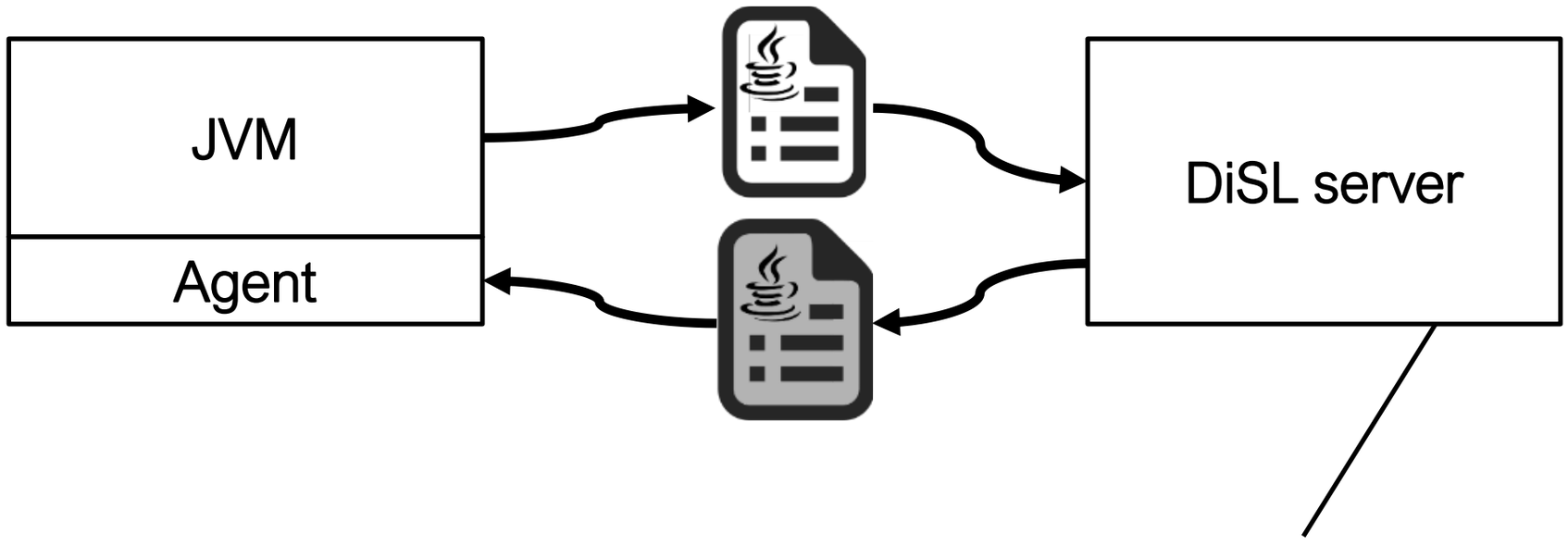
- We present a new technique to **accurately reify complete RSI** in a **separate instrumentation process**
  - Reification of classloader namespaces
  - Complete RSI available upon instrumentation of any class
  - Works with full code coverage
- Implemented in DiSL

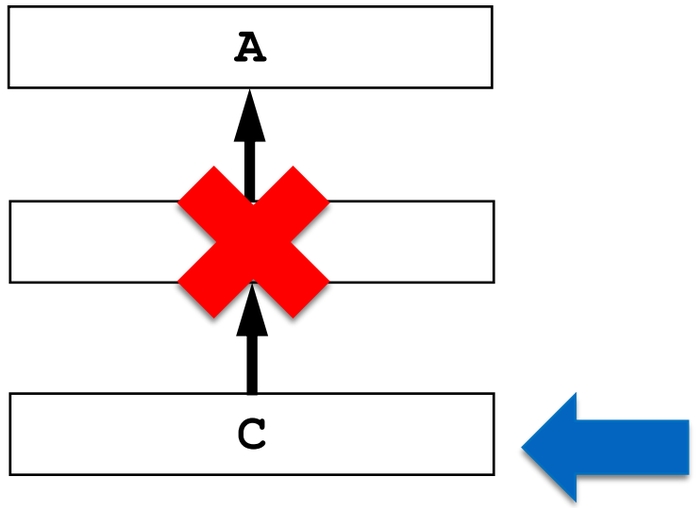
DiSL Reflection API




```
public interface DiSLClassLoader {
    long getID();
    DiSLClass forName(String fullyQualifiedClassName);
    boolean isBootstrapLoader();
    boolean isApplicationLoader();
    ...
}

public interface DiSLClass {
    String getName();
    DiSLClassLoader getClassLoader();
    DiSLClass getSuperclass();
    Stream<DiSLClass> getInterfaces();
    Stream<DiSLClass> getSupertypes();  // All direct and indirect supertypes
    ...
}
```

Forced Loading of Supertypes





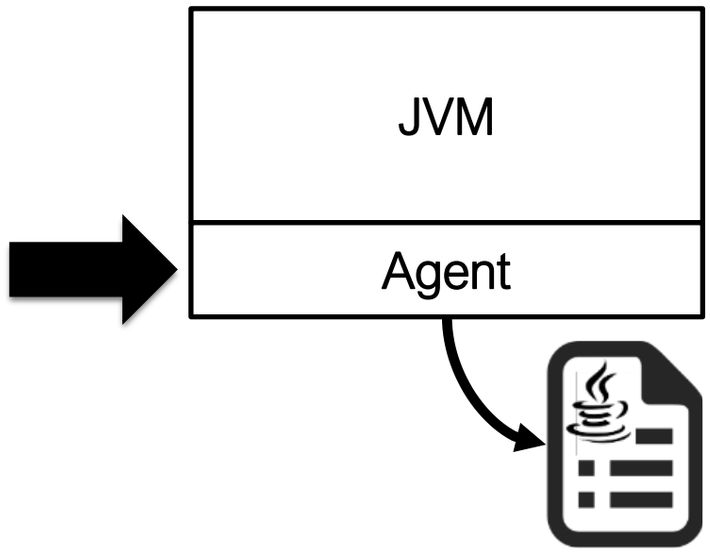
- JVM may load subtypes before supertypes

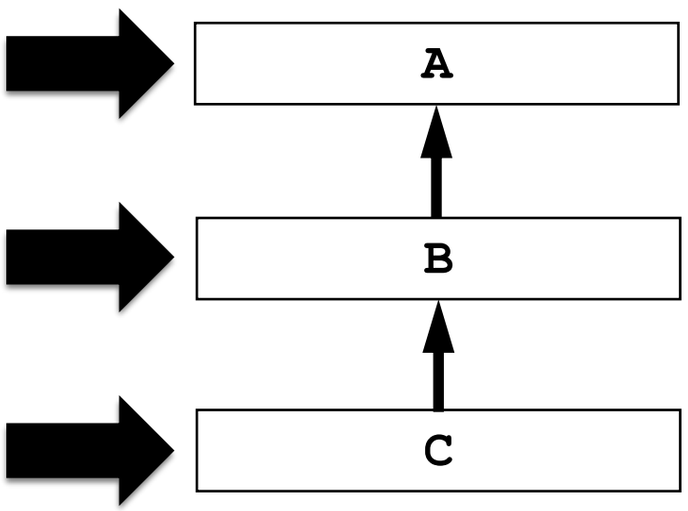
```
public interface DiSLClass {
    String getName();
    DiSLClassLoader getClassLoader();
    DiSLClass getSuperclass();
    Stream<DiSLClass> getInterfaces();
    Stream<DiSLClass> getSupertypes();
    ...
}
```

- Solution: force loading of supertypes

Forced Loading of Supertypes

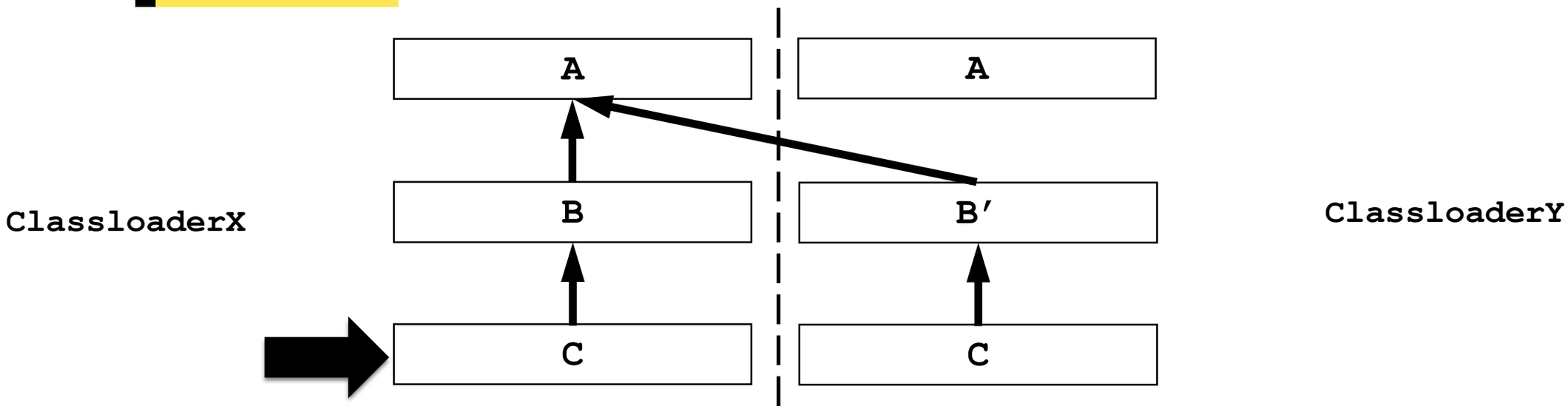
```
input  : b  original byte array of the loaded class
         l  defining classloader of the loaded class
output : B_c  instrumented byte array of the loaded class

1  callback onClassLoad(B_c b, L l) begin
2      STRING s_s ← superclass(b)
3      P p_s ← < s_s, loadAndGetClassloaderID(s_s, l) >
4      𝒫 p_i ← ()
5      𝒮 s_i ← interfaces(b)
6      foreach s_j ∈ s_i do
7          P p_j ← < s_j, loadAndGetClassloaderID(s_j, l) >
8          p_i ← p_i ‖ (p_j)
9      end
10     return instrumentRemotely(b, id(l), p_s, p_i)
11 end

12 function loadAndGetClassloaderID(STRING s, L l) begin
13     return id(classloader(loadClass(s, l)))
14 end
```



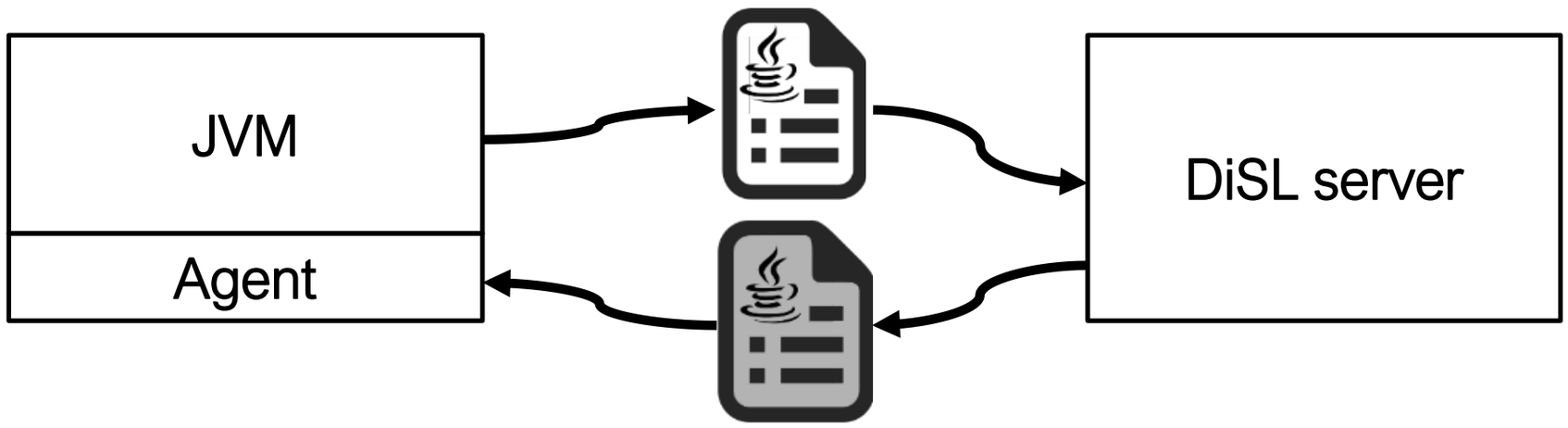


Triggers instrumentation of *s*

Classloader Namespaces

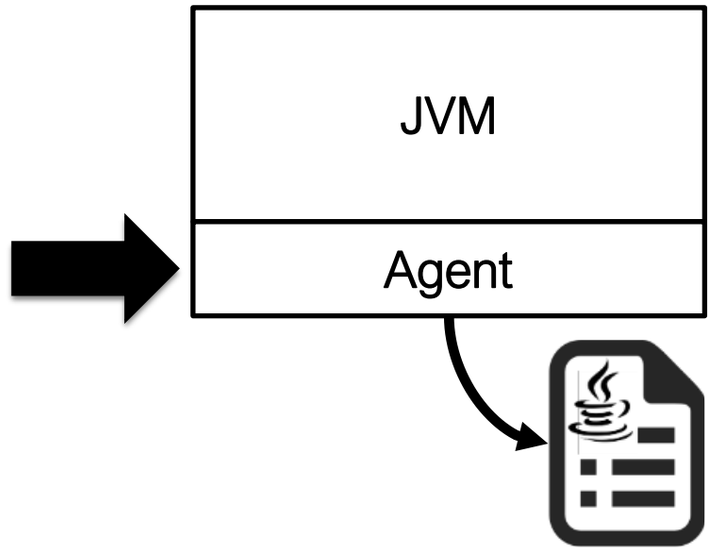


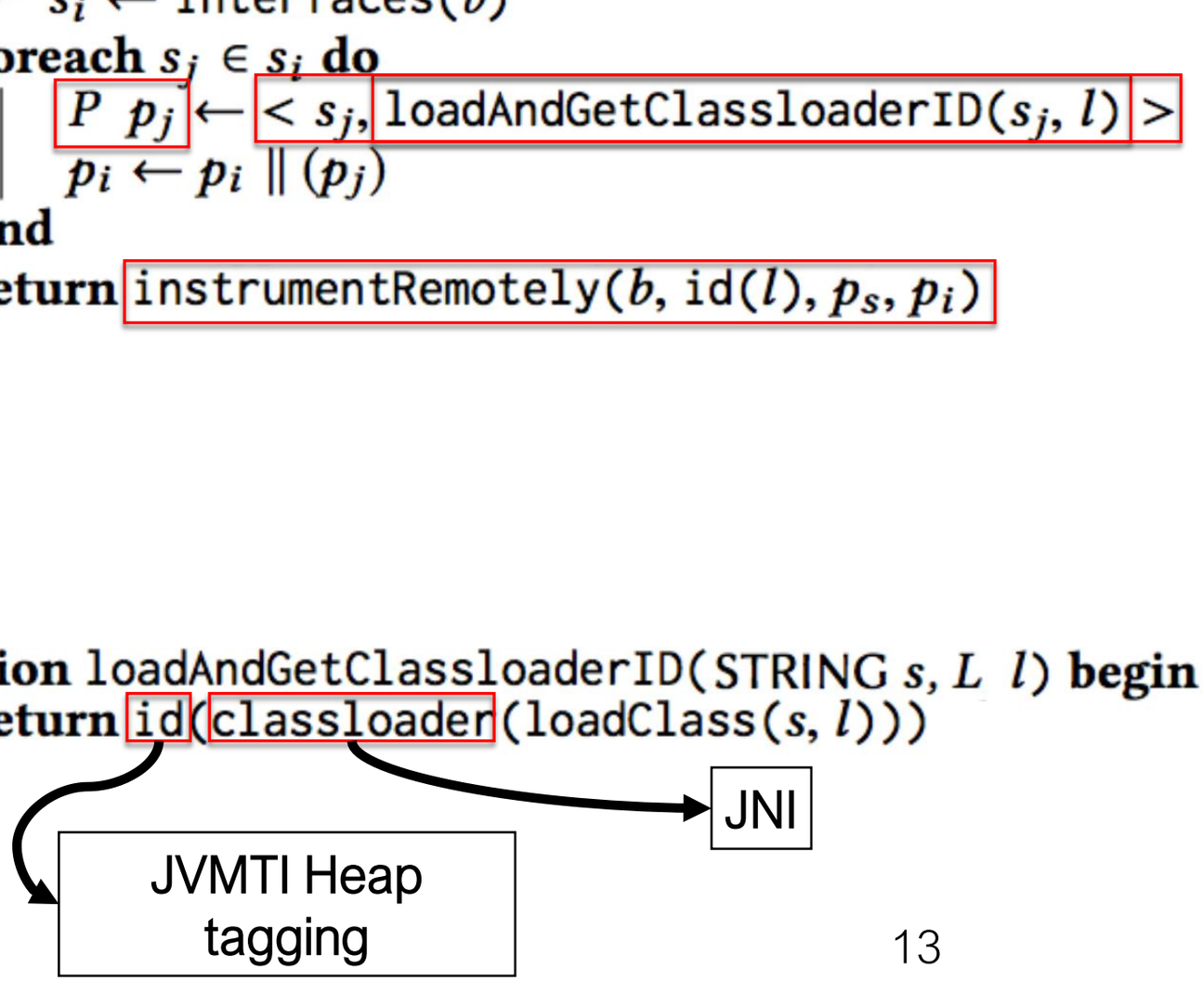

```
<C, ClassloaderX>
<B, ClassloaderX>
```



Classloader Namespaces

```
input  : b  original byte array of the loaded class
         l  defining classloader of the loaded class
output : B_c  instrumented byte array of the loaded class

1  callback onClassLoad(B_c b, L l) begin
2      STRING s_s ← superclass(b)
3      P p_s ← < s_s, loadAndGetClassloaderID(s_s, l) >
4      𝒫 p_i ← ()
5      𝒮 s_i ← interfaces(b)
6      foreach s_j ∈ s_i do
7          P p_j ← < s_j, loadAndGetClassloaderID(s_j, l) >
8          p_i ← p_i ‖ (p_j)
9      end
10     return instrumentRemotely(b, id(l), p_s, p_i)
11 end

12 function loadAndGetClassloaderID(STRING s, L l) begin
13     return id(classloader(loadClass(s, l)))
14 end
```



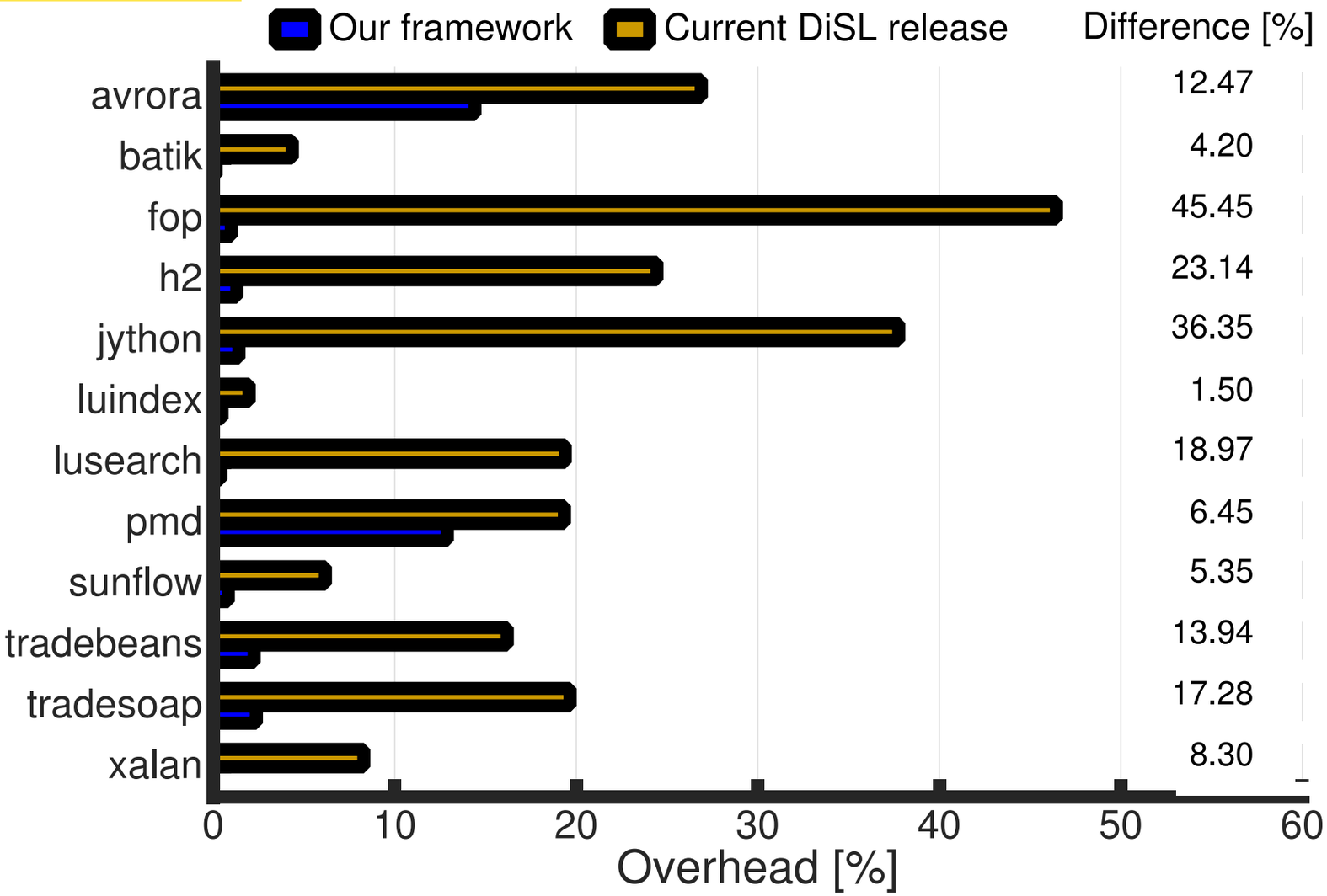


Evaluation

- Task profiling on DaCapo benchmarks [3]
- Target: subtypes of `Runnable` and `Callable`
- Comparison between:
  - AspectJ (load-time in-process instrumentation)
  - DiSL without Reflection API
  - DiSL with Reflection API
- Metrics:
  - Overhead
  - Code coverage

[3] Blackburn et al. *The DaCapo Benchmarks: Java Benchmarking Development and Analysis*. OOPSLA'06.

Evaluation - Overhead



- High overhead without DiSL Reflection API
  - Average overhead: 19%

- Reduced overhead with DiSL Reflection API
  - Average overhead: 3%
  - Difference: 16% (average) – 45% (max)

Evaluation – Code Coverage

Benchmark	# tasks AspectJ	# tasks DiSL	difference (DiSL - AspectJ)	% coverage AspectJ
avrora	8	9	1	88.89
batik	1	3	2	33.33
eclipse	139	368	229	37.77
fop	0	3	3	0
h2	34	35	1	97.14
jython	2	4	2	50.00
luindex	2	2	0	100
lusearch	33	33	0	100
pmd	253	507	254	49.90
sunflow	66	101	35	65.35
tomcat	2737	3083	346	88.78
tradebeans	34	37	3	91.89
tradesoap	34	37	3	91.89
xalan	1733	1733	0	100

- Tasks in Java class library not detected by AspectJ
  - Only 71% of tasks profiled by AspectJ

Limitations

- Incompatible with analyses making use of JVMTI heap tagging
  - Possible solution:
    weak-key identity-based hash tables
    instead of heap tagging
- RSI currently not available for fields, method receivers, and arguments
  - Huge memory footprint and slow startup

Conclusions

- New technique to reify RSI in a separate instrumentation process
  - Instrumentation process aware of classloader namespaces
  - Accurate and complete RSI available for each loaded class
  - Reconciling complete RSI and full code coverage
- Integrated into DiSL
- Significant overhead reductions in type-specific instrumentations

Thank you for the attention

- DiSL: **`https://disl.ow2.org`**

- Andrea Rosà

  `andrea.rosa@usi.ch`
  `http://www.inf.usi.ch/phd/rosaa`