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

Andrea Rosà
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
andrea.rosa@usi.ch

Eduardo Rosales
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
rosale@usi.ch

Walter Binder
Faculty of Informatics
Università della Svizzera italiana
Lugano, Switzerland
walter.binder@usi.ch

Abstract

Reflective supertype information (RSI) is useful for many instrumentation-based dynamic analyses on the Java Virtual Machine (JVM). On the one hand, while such information can be obtained when performing the instrumentation within the same JVM process executing the instrumented program, in-process instrumentation severely limits the code coverage of the analysis. On the other hand, performing the instrumentation in a separate process can achieve full code coverage, but complete RSI is generally not available, often requiring expensive runtime checks in the instrumented program. Providing accurate and complete RSI in the instrumentation process is challenging because of dynamic class loading and classloader namespaces. In this paper, we present a novel technique to accurately reify complete RSI in a separate instrumentation process. We implement our technique in the dynamic analysis framework DiSL and evaluate it on a task profiler, achieving speedups of up to 45% for an analysis with full code coverage.

CCS Concepts • Software and its engineering → Dynamic analysis;

Keywords Dynamic analysis, reflective information, byte-code instrumentation, Java Virtual Machine

ACM Reference Format:

Andrea Rosà, Eduardo Rosales, and Walter Binder. 2017. Accurate Reification of Complete Supertype Information for Dynamic Analysis on the JVM. In *Proceedings of 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE'17)*. ACM, New York, NY, USA, 13 pages. <https://doi.org/10.1145/3136040.3136061>

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

GPCE'17, October 23–24, 2017, Vancouver, Canada

© 2017 Copyright held by the owner/author(s).

ACM ISBN 978-1-4503-5524-7/17/10.

<https://doi.org/10.1145/3136040.3136061>

1 Introduction

Dynamic analyses are essential tools in the development and maintenance of applications. A practical dynamic analysis should be efficient (i.e., it should not cause a significant slowdown of the application being analyzed) and guarantee full code coverage (i.e., it should profile every relevant event). Our work targets dynamic analyses on the Java Virtual Machine (JVM). Such analyses often rely on bytecode instrumentation frameworks¹ to insert the monitoring code into selected methods, to collect the desired dynamic metrics at runtime.

The availability of accurate and complete *reflective supertype information (RSI)* (i.e., information about all direct and indirect supertypes of a class) at instrumentation-time often allows more efficient dynamic analyses. Consider an analysis targeting every thread spawned by an application (e.g., all subtypes of class `java.lang.Thread`). If complete RSI is available at instrumentation-time, the weaver can instrument only `Thread` and its subtypes, leaving all other classes untouched and reducing the overhead of the instrumentation. On the contrary, if complete RSI is not available, the weaver may have to instrument many more classes loaded by the application, inserting expensive runtime checks into the instrumented code to determine whether a class represents a thread, introducing additional runtime overhead.

Unfortunately, existing approaches to expose RSI at instrumentation-time suffer from serious limitations. On the one hand, static analysis does not guarantee accurate and complete RSI, as it cannot always handle classloader namespaces correctly (e.g., it may not process classes loaded by custom classloaders), although classloaders play an important role in many frameworks such as in the dynamic module system OSGi [22]. Moreover, due to dynamic class loading, static analysis may not see all relevant classes. Nonetheless, some frameworks that perform static instrumentation (such as the AspectJ [10] compile-time weaver and abc [1]) resort to static analysis, yielding possibly inaccurate or incomplete RSI.

On the other hand, obtaining accurate and complete RSI when performing load-time instrumentation in the observed

¹In this paper, we do not distinguish between instrumentation and weaving, and use the term *weaver* to denote the component performing the instrumentation.

JVM is possible, although such an approach also has serious drawbacks. In such a setting, the weaver can access the Class instances of the supertypes of the class under instrumentation, allowing the inspection of RSI. Unfortunately, this approach can interfere with the instrumentation of the Java class library, class loading, and JVM initialization [2, 9]. To mitigate these problems, in-process load-time weavers (such as the AspectJ load-time weaver) often prevent any instrumentation of the Java class library, leading to limited code coverage.

To instrument all loaded classes, frameworks such as DiSL [11] perform instrumentation in a JVM running in a separate process.² In this setting, instrumenting the Java class library is possible, but classloader information and Class instances of the instrumented application are not available. While providing full code coverage, the instrumentation process has access only to limited RSI and cannot access classloader namespaces.

In this paper, we present a novel technique to accurately reify the class hierarchy (including classloader namespaces) of the instrumented application within a separate instrumentation process, such that accurate and complete RSI is available for each class to be instrumented. We implement our technique in the dynamic analysis framework DiSL [11]. Our enhanced DiSL framework ensures that the instrumentation process is aware of classloader namespaces and correctly deals with homonym classes defined by different classloaders; accurate and complete RSI is guaranteed available even if some supertypes are defined by different classloaders.

Our work makes the following contributions. We present a new API to accurately reify complete RSI within the DiSL instrumentation process (Sec. 4). Our technique ensures that supertypes are instrumented before subtypes, and exposes classloader namespaces to the instrumentation logic. It handles several special cases, such as the core Java classes loaded in the primordial phase of the JVM (Sec. 5). Moreover, we show that our approach enables significant speedups wrt. the current DiSL release (up to 45%) for a task profiler when analyzing the DaCapo benchmarks [3] with full code coverage (Sec. 6). We complement the paper with background information on the used technologies (Sec. 2), a motivating example (Sec. 3), a discussion of advanced features and limitations of our approach (Sec. 7), and a comparison with related work (Sec. 8).

2 Background

Before presenting our work, we introduce some preliminary information on classloaders, reflective information, the interfaces used to support our approach (JVM TI and JNI), and DiSL.

²We use the term *instrumentation process* to refer to the process where the JVM instance performing the instrumentation is running.

Classloaders A Java class³ is created by loading a binary representation of the class (i.e., *classfile*) using a *classloader*, which is responsible for locating the corresponding classfile, parsing it, and constructing a class representation within the JVM. A classloader can either be the *bootstrap classloader* or a *user-defined* classloader.⁴ The bootstrap classloader, supplied by the JVM, is responsible for loading the Java core classes (e.g., the `java.*` package), while a user-defined classloader is responsible for loading application classes. All user-defined classloaders are subtypes of `java.lang.ClassLoader`. The standard user-defined classloader used to load the main class of an application (i.e., the *application classloader*) can only locate classfiles in the application *classpath* (specified by the user at JVM startup). Developers can create their own user-defined classloaders to extend this behaviour. This is needed if an application fetches classfiles from other locations (such as a remote server) or loads special classes (such as encrypted or dynamically generated ones).

When a classloader CL is requested to load a class C, it can either load the class itself or delegate loading to another classloader. In the former case, CL is known as the *defining classloader* of C. Each class has a single defining classloader. The bootstrap classloader cannot delegate classloading. Normally, a class C is loaded when another previously loaded class D references it. The first classloader that attempts loading C is the defining classloader of D.⁵ A class may be loaded before some of its supertypes. Loading C triggers the subsequent loading of its direct supertypes (i.e., its superclass and all directly implemented interfaces) if they are not yet loaded. Due to delegation, it is possible that the defining classloader of a class is different from the one of its supertypes.

At runtime, a class is determined by its fully qualified name (i.e., the class name, including the package where the class is defined) and its defining classloader. Several classes with the same fully qualified name may exist at runtime, provided that their defining classloaders are different. To this end, classloaders maintain different *namespaces*: a classloader is only aware of the classes it has defined or has delegated loading. Our approach exposes classloader namespaces to an instrumentation process, ensuring that homonym classes defined by different classloaders can be handled correctly, and that RSI is always accessible even if a supertype and a subtype are defined by different classloaders.

Reflective Information *Reflective information* refers to any information related to a Java class or method available at runtime. Developers can obtain reflective information through the *Java Reflection API* [16]. Any loaded class is represented by an instance of `java.lang.Class`, which allows

³In this paper, we use the term *class* to indiscriminately refer to both Java classes and interfaces.

⁴We use the terminology of the JVM specification [15]. Note that some user-defined classloaders are part of the Java class library.

⁵The use of the Java Reflection API may cause exceptions to this behaviour. See the documentation of `java.lang.Class` for more information.

one to inspect reflective information of the corresponding class. Our work focuses on *reflective supertype information (RSI)*, i.e., complete information about all direct and indirect supertypes of a class. Such information could be retrieved by calling `Class.getSuperclass` and `Class.getInterfaces`, which return `Class` instances corresponding to the direct superclass and the directly implemented interfaces of a class, respectively. Complete RSI of a class can be obtained by repeatedly calling these methods on all the supertypes of the class. This approach only works in the process running the application.

If complete RSI is available at instrumentation-time, the weaver can selectively instrument a class based on its position in the type hierarchy, allowing more efficient dynamic analyses. Unfortunately, frameworks performing instrumentation in a separate process cannot access the `Class` instances of the classes loaded in the target application (because such classes are not loaded in the instrumentation process). Thus, they cannot normally inspect complete RSI. Our approach fills this gap, making complete RSI available to the instrumentation process.

JVMTI and JNI JVMTI⁶ (JVM Tool Interface) is an interface that enables inspecting the state of a JVM and controlling the execution of applications running on top of it. JVMTI exposes an API to a native *agent* (written in C or C++) to be attached to a JVM. The agent runs in the same process and directly communicates with the JVM it is attached to. Among the features offered by the interface, a JVMTI agent can intercept classloading, allowing one to modify the final representation of a class before it is linked in the JVM. Moreover, JVMTI supports *heap tagging*: agents can assign a unique long value (i.e., a tag) to any object allocated on the heap, as well as retrieve or unset the tag associated to an object. Untagged objects have a tag of 0. Agents can also be notified when a tagged object is reclaimed by the garbage collector, executing custom code when this occurs.

JNI⁷ (Java Native Interface) is a standard interface for writing native Java methods. Similarly to JVMTI, developers can call the JNI methods from native agents attached to a JVM. Amongst others, JNI allows agents to call methods of Java objects from native code. JNI is not available during the early initialization phase of the JVM (i.e., the *primordial phase*). Our approach makes use of a combination of JVMTI and JNI to expose RSI and classloader namespaces to the instrumentation process.

DiSL DiSL [11] is a dynamic analysis framework based on Java bytecode instrumentation. In DiSL, developers write instrumentation code in the form of *snippets*, based on Aspect-Oriented Programming (AOP) principles that allow a concise

implementation of runtime monitoring tools. DiSL allows developers to specify where a snippet should be woven through *markers* (specifying which parts of a method to instrument, such as method bodies, basic blocks, etc.), *annotations* (specifying where a snippet must be inserted wrt. a marker, e.g., before or after method bodies), *scope* (specifying which classes or methods shall be instrumented based on a pattern matching scheme) and *guards* (predicate methods enabling the evaluation of conditionals at instrumentation-time to determine whether a snippet should be woven or not in the method being instrumented). Snippets and guards have access to *context information* provided via method arguments. Context information can be either static (i.e., static information limited to constants) or dynamic (i.e., including local variables and the operand stack). Dynamic information can be accessed only by snippets.

DiSL performs the instrumentation in a separate JVM process (i.e., the *instrumentation server*). A native JVMTI agent attached to the observed JVM intercepts classloading, sending each loaded class to the instrumentation server. The DiSL weaver guarantees full code coverage of an analysis. In particular, DiSL allows one to instrument classes in the core Java libraries, which cannot normally be instrumented with in-process instrumentation. While providing full code coverage, the latest DiSL release⁸ does not have access to complete RSI and is unaware of classloader namespaces. As a consequence, it cannot instrument classes based on their position in the type hierarchy, and treats as equal classes with the same fully qualified name but different defining classloaders. Our work tackles this issue, providing complete RSI and classloader namespaces to the weaver. This information can be inspected thanks to our new DiSL Reflection API, described in Section 4.

3 Motivating Example

In this section, we discuss the limitations of existing instrumentation techniques in offering full code coverage and accessing RSI, motivating the need for a new approach. As running example, we consider a simple dynamic analysis that profiles the execution of method `m` (with any parameters and return type) in all subtypes of `A` (including `A` itself). To correctly determine the classes to instrument, the weaver has to retrieve complete RSI for each class. We first discuss the benefits and limitations of implementing such an analysis with prevailing techniques, and highlight the importance of taking classloader namespaces into account; then, we outline how our approach overcomes these limitations.

Compile-time Instrumentation A possible approach is to implement the analysis with compile-time instrumentation (i.e., replacing existing classfiles with instrumented ones), for example resorting to the compile-time weaver of

⁶<https://docs.oracle.com/javase/8/docs/technotes/guides/jvmti/>

⁷<http://docs.oracle.com/javase/8/docs/technotes/guides/jni>

⁸DiSL 2.1 at the time of writing.

the AspectJ AOP framework [10]. Figure 1(a) shows the code for implementing the analysis in AspectJ. Compile-time instrumentation typically resorts to static analysis to derive the type hierarchy of an application, so as to insert instrumentation code (line 4) only in *A* and its subtypes. Unfortunately, static analysis cannot guarantee accurate and complete RSI, as it lacks information about which classes will be loaded at runtime (due to dynamic class loading). Moreover, it may not handle classloader namespaces correctly (e.g., it may not process classes loaded by custom classloaders). Such limited RSI may lead to failures in instrumenting all relevant classes (e.g., every subtype of *A*), resulting in an analysis with limited code coverage.

Load-time In-process Instrumentation Another approach is load-time instrumentation, performing the instrumentation within the observed JVM process (an approach known as *in-process instrumentation*). In this setting, instrumentation occurs right after a class has been loaded (before it is linked by the JVM). Interfaces such as those of the *Java Instrumentation API* [13] enable intercepting and instrumenting each loaded class, including dynamically loaded classes or those defined by custom classloaders. For example, the AspectJ load-time weaver operates in such a setting.

While obtaining complete RSI using load-time instrumentation is possible, such an approach has serious drawbacks. On the one hand, the weaver has access to the classloader used to load a class (provided as argument to a *ClassFileTransformer*), and can call *Class.forName* (passing the classloader as an argument) to access the *Class* instances of the direct supertypes; then, it can use this approach recursively to get the *Class* instances of all indirect supertypes. On the other hand, this approach interferes with the instrumentation of the Java Reflection API and can cause infinite recursions if the instrumentation is not handled with care. Moreover, agents making use of the Java class library can interfere with the instrumentation of the Java class library itself (in practice this is often the case, unless the whole instrumentation logic is implemented as a native-code agent). To prevent interferences with class loading and JVM initialization [2, 9], in-process load-time weavers often prevent instrumentation of the Java class library altogether.⁹ Similar to the previous case, the resulting analyses usually suffer from limited code coverage. For example, subtypes of *A* inside *java.** cannot be instrumented.

Load-time Out-of-process Instrumentation Another alternative is performing the instrumentation at load-time in a JVM instance running in a different process (an approach

```
1 public aspect Instrumentation {
2   pointcut exec(): execution(void A+.m(..));
3   before(): exec() {
4     // Inserted instrumentation code - omitted here
5   }
6 }
```

(a) AspectJ [10]

```
1 @Before(marker = BodyMarker.class, scope = "m")
2 static void beforeMethod(DynamicContext dc) {
3   if (dc.getThis() instanceof A) { // Inserted
4     // Inserted instrumentation code - omitted here
5   }
6 }
```

(b) DiSL [11]

Figure 1. AspectJ and DiSL code for instrumenting method *m* in every subtype of *A* (including *A* itself).

known as *out-of-process instrumentation*). This setting prevents any interference with class loading or JVM initialization and enables full code coverage. For example, DiSL [11] employs load-time out-of-process instrumentation and can instrument every loaded class, including Java core classes.

Unfortunately, the instrumentation process has no access to complete RSI, because it cannot access the *Class* instances of the target application. In particular, the weaver can only inspect the *names* of the *direct* supertypes of a class (as they are encoded in the classfile), while any other information on the direct and indirect supertypes may be missing. To implement the target analysis in such a setting (while ensuring full code coverage), the weaver must resort to expensive runtime checks (i.e., using the *instanceof* operator) to detect subtypes of *A*. Figure 1(b) shows a DiSL snippet implementing the analysis. The snippet instructs the weaver to instrument all methods named *m* of *every* loaded class (line 1), executing the profiling logic (line 4) only if the current object is an instance of *A* (line 3).¹⁰ The performance penalty introduced by such checks may be significant, especially if *m* is a method defined in many classes or a constructor. While guaranteeing full code coverage, the resulting analysis can be quite inefficient.

Classloader Namespaces Classloaders maintain their own namespaces and are unaware of the classes defined by other classloaders. As a result, two classloaders can define different classes with the same fully qualified name. This situation can lead to incomplete instrumentation if not carefully handled by the weaver. To better understand this issue, consider the classes shown in Figure 2, and suppose that all of them are loaded by the application. Classes *B* and *C* (lines 2–3) are located in the classpath and are defined by classloader *CL1* (i.e., the application classloader), while those at lines 6–7 are

⁹For example, AspectJ prevents the instrumentation of classes in the packages *java.**, *javax.**, and *sun.reflect.**. More details are available at <https://eclipse.org/aspectj/doc/released/devguide/ltw-specialcases.html>.

¹⁰The snippet shown in Figure 1(b) could be optimized if *A* is a direct supertype of the class under instrumentation, or if the latter is a direct subtype of *Object* and does not implement any interface.


```

1 // Located in the classpath - defined by CL1
2 public class B { void m() { /*...*/ } }
3 public class C extends B { void m() { /*...*/ } }
4
5 // Located outside the classpath - defined by CL2
6 public class B extends A { void m() { /*...*/ } }
7 public class C extends B { void m() { /*...*/ } }

```

Figure 2. Example of homonym classes defined by different classloaders.

located outside the classpath and are defined by classloader CL2 (i.e., a custom classloader). To guarantee a correct analysis, the weaver must ensure that only classes B and C defined by CL2 are instrumented (as they are subtypes of A), leaving untouched those defined by CL1 (which are not subtypes of A). Unfortunately, compile-time instrumentation may not process classes loaded by custom classloaders such as CL2. As a result, it may miss the instrumentation of B and C loaded by CL2.

Moreover, suppose now that our analysis targets subtypes of B rather than A. A flexible instrumentation framework should provide users a way to define at instrumentation-time which of the two B classes fall within the scope of the analysis (e.g., instrument only subtypes of B defined by CL2), such that the weaver may instrument classes based on their defining classloaders. Both AspectJ and DiSL does not offer such functionality, requiring the insertion of runtime checks to retrieve the defining classloader of a class and, if the classloader matches the one targeted by the analysis, executing instrumentation code.

Our Solution Our approach reconciles the benefit offered by out-of-process instrumentation (i.e., full code coverage) with the provisioning of accurate and complete RSI and classloader namespaces. We implement our technique in an extended version of DiSL, where the weaver reifies classloader namespaces. Our solution avoids the expensive runtime checks required by the current DiSL release, enabling efficient analyses that guarantee full code coverage. Moreover, our framework correctly deals with homonym classes defined by different classloaders and allows the developer to identify classloader namespaces (if desired) when writing dynamic analyses. The weaver can inspect RSI and classloader namespaces thanks to a new API, described in the next section.

4 The DiSL Reflection API

A key feature of our approach is the new *DiSL Reflection API*, which provides an interface to access RSI within the instrumentation server, as if the server could access the (future) Class instance corresponding to the class being instrumented. Moreover, our API allows inspecting classloader namespaces.

```

1 public interface DiSLClassLoader {
2     long getID();
3     DiSLClass forName(String fullyQualifiedClassName);
4     boolean isBootstrapLoader();
5     boolean isApplicationLoader();
6     ...
7 }
8
9 public interface DiSLClass {
10     String getName();
11     DiSLClassLoader getClassLoader();
12     DiSLClass getSuperclass();
13     Stream<DiSLClass> getInterfaces();
14     Stream<DiSLClass> getSupertypes();
15     ...
16 }
17
18 public class ReflectionContext extends
19     AbstractStaticContext {
20     public DiSLClass thisClass() { ... };
21     ...
22 }

```

Figure 3. Classes and methods of the new DiSL Reflection API. Methods not relevant for this paper are omitted.

Figure 3 summarises the key classes of the DiSL Reflection API. The purpose of the classes `DiSLClassLoader` and `DiSLClass` is to mirror their counterparts in the target application (i.e., `ClassLoader` and `Class`, respectively), such that RSI and classloader namespaces can be inspected from the instrumentation server. Each time a new classloader is used to load a class, the corresponding instance of `DiSLClassLoader` is created in the instrumentation server. Similarly, when a class is loaded by the target application, the server creates the corresponding `DiSLClass`.

`DiSLClassLoader` represents the namespace of the corresponding classloader within the instrumentation server; in particular, it maintains the mapping from fully qualified class names to the corresponding `DiSLClass` instances. Thanks to our API, an instrumentation is able to correctly handle homonym classes defined by different classloaders. Internally, `DiSLClassLoader` instances are identified by a unique ID (line 2). ID 0 corresponds to the bootstrap classloader, ID 1 corresponds to the application classloader, while subsequent IDs identify other user-defined classloaders.

A `DiSLClass` allows inspecting RSI of a class from the instrumentation server. In particular, it exposes the superclass (line 12) and the directly implemented interfaces (line 13) of a class. Our approach ensures that when a class is being instrumented, the corresponding `DiSLClass` and those of all its supertypes are available. Supertype information returned by a `DiSLClass` is fully compliant with the JVM specification [15].¹¹ `DiSLClass` also offers the convenience

¹¹The contracts of the methods `getSuperclass` and `getInterfaces` are the same of the homonym methods in `java.lang.Class`, with the exception that interfaces are returned using a `Stream` rather than an array.

Table 1. Abstract data types used in Figure 5 and their translation into Java, C or JNI types.

Type	Description	Java equivalent	C/JNI equivalent
C	A class.	<code>java.lang.Class</code>	<code>jclass</code>
B_c	A byte array representing class c .	<code>byte[]</code>	<code>char *</code>
L	A classloader (<i>null</i> represents the bootstrap classloader).	<code>java.lang.Classloader</code>	<code>jobject</code>
$\mathcal{S}$	A list of STRINGS.	<code>List<String></code>	<code>char **</code>
P	An ordered pair $\langle \text{STRING}, \text{LONG} \rangle$. The first element represents a fully qualified class name; the second element the ID of a classloader.	<code>class P { String name; long id; }</code>	<code>typedef struct { char *name; jlong id;} P</code>
$\mathcal{P}$	A list of P pairs.	<code>List<P></code>	<code>P *</code>

Table 2. Operations on abstract data types used in Figure 5. ϵ denotes a *null* value, while $()$ denotes the empty list.

Function	Description	C code
<code>superclass(B_c b): STRING</code>	Returns the fully qualified name of c 's direct superclass, or ϵ if c is <code>java.lang.Object</code> .	-
<code>interfaces(B_c b): $\mathcal{S}$</code>	Returns the list of fully qualified names of c 's directly implemented interfaces, in the order they are implemented, or $()$ if c does not implement any interface.	-
<code>loadClass(STRING s, L l): C</code>	Loads a class with fully qualified name s using classloader l , returning the corresponding <code>java.lang.Class</code> .	Figure 6
<code>classloader(C c): L</code>	If c is defined by a user-defined classloader, returns the corresponding <code>java.lang.ClassLoader</code> . If c is defined by the bootstrap classloader, returns ϵ .	Figure 7
<code>id(L l): LONG</code>	Returns 0 if $l = \epsilon$. Otherwise, returns the unique ID associated with l . If no ID has been associated to l , this function assigns a unique ID to l before returning it.	Figure 8
<code>instrumentRemotely(B_c b, LONG i, P p_s, $\mathcal{P}$ p_i): B_c</code>	Sends b to the server for instrumentation, along with the ID of c 's defining classloader (i) and information to univocally identify c 's superclass (p_s) and c 's directly implemented interfaces (p_i). Returns the instrumented byte array of c .	-

```

1 @Before(marker = BodyMarker.class, scope = "m",
2         guard = Guard.class)
3 static void beforeMethod() {
4     // Inserted instrumentation code - omitted here
5 }
6
7 // Executed at instrumentation-time
8 public final class Guard {
9     @GuardMethod
10    public static boolean guard(ReflectionContext rsc) {
11        return rsc.thisClass().getSupertypes().anyMatch(
12            s -> s.getName().equals("A") &&
13                s.getClassLoader().isApplicationLoader());
14    }
15 }

```

Figure 4. DiSL code for instrumenting method m in every subtype of A (including A itself) using the DiSL Reflection API.

method `getSupertypes` to retrieve all (direct and indirect) supertypes of a class, including the class itself.

The DiSL Reflection API is typically used in guards during the instrumentation process. To this end, our API introduces new context information than can be accessed by guards through the `ReflectionContext` (line 18). In particular, this class provides the `DiSLClass` (line 19) of the class currently under instrumentation, allowing guards to perform their checks based on RSI.

Profiling Subtypes of A Thanks to the provision of RSI and classloader namespaces, DiSL-based analyses running on our framework can avoid inserting expensive runtime checks in many loaded classes (see Figure 1(b)), as the weaver

can insert instrumentation code only into the classes in the scope of the analysis. Figure 4 shows a DiSL snippet and guard implementing the sample analysis introduced in Section 3. Instead of inserting runtime checks in every class, the snippet inserts instrumentation code (line 4) only if the associated guard (line 2) evaluates to true. The guard (executed at instrumentation-time) resorts to the DiSL Reflection API to instrument only A and its subtypes (line 8) by retrieving the `DiSLClass` of the class being instrumented through the `ReflectionContext` (line 11), and checking whether class A (line 12) is among its supertypes. If desired, the guard can specify the namespace where A should be defined. For example, the code in the figure ensures that only subtypes of a class or interface named A defined by the application classloader (line 13) are instrumented.

5 Implementation

The DiSL Reflection API enables efficient analyses thanks to the provisioning of RSI and classloader namespaces. In this section, we discuss the key implementation details of our approach, which ensures that such information is always available in the instrumentation server. We first outline the algorithms involved by means of abstract data types; then, we expand the discussion showing C and Java code. The abstract data types involved in this section are shown in Table 1, and the operations defined on them are reported in Table 2. We assume the existence of the types `STRING` and `LONG`. Note that lists are ordered and support concatenation through the

$\parallel$ operator. In the following text, we often assume that the server is instrumenting a class c .¹²

5.1 Forced Loading of Supertypes

Our instrumentation server creates a new `DiSLClass` when it receives the corresponding class for instrumentation. To access RSI of c at instrumentation-time (e.g., in guards), the `DiSLClass` instances of all the supertypes of c must be available in the server when c is being instrumented. This implies that every supertype of c must be instrumented before c itself. Unfortunately, the current DiSL release sends classes to the server in the same order as they are loaded by the JVM; hence, some of the supertypes of c may not have been received by the server when instrumenting c . In such a case, attempts of retrieving RSI at instrumentation-time will fail, as some required `DiSLClass` instances are not yet available. We tackle this issue by modifying the order in which classes are sent to the server for instrumentation, ensuring that all the supertypes of c are sent before c . To this end, our framework forces the loading of each supertype of c that has not yet been loaded.

Figure 5 reports the algorithm used to force the loading and instrumentation of c 's supertypes. The algorithm is executed in a callback, invoked when c has been loaded in the target JVM (but before the JVM has created a `Class` representing c). The callback has access to the byte array representing c as loaded from the classfile (which can be modified by the instrumentation server), and the defining classloader of c . First, the algorithm retrieves the fully qualified name of c 's superclass (line 2) and triggers its loading through the helper function `loadAndGetClassLoaderID` (line 3). In particular, this function asks the JVM to load the supertype using the defining classloader of c via `loadClass` (line 14). This process is then repeated for each of the interfaces directly implemented by c (lines 5–7). Note that `loadClass` triggers the nested execution of the callback in the context of a supertype of c , resulting in the recursive loading of *every* supertype of c , including indirect ones. When the `foreach` loop terminates (line 10), every supertype of c has been loaded and instrumented. At this time, complete RSI of c is available in the instrumentation server, and the algorithm can send the class to the server for instrumentation (line 11).

The above algorithm is implemented in the JVMTI agent attached to the target JVM. In particular, the callback and its arguments are provided by JVMTI, which allows an agent to be notified and execute custom code upon class loading.¹³ Functions `superclass` and `interfaces` are implemented by looking up the byte array representing c at specific indices, following the class file format [15]. Finally, the C code for

```

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( $B_c$ ,  $b$ ,  $l$ ) begin
2    $STRING\ s_s \leftarrow \text{superclass}(b)$ 
3    $P\ p_s \leftarrow \langle s_s, \text{loadAndGetClassLoaderID}(s_s, l) \rangle$ 
4    $\mathcal{P}\ p_i \leftarrow ()$ 
5    $\mathcal{I}\ s_i \leftarrow \text{interfaces}(b)$ 
6   foreach  $s_j \in s_i$  do
7      $P\ p_j \leftarrow \langle s_j, \text{loadAndGetClassLoaderID}(s_j, l) \rangle$ 
8      $p_i \leftarrow p_i \parallel (p_j)$ 
9   end
10  /* Here, all supertypes have been (recursively)
    instrumented */
11  return instrumentRemotely( $b$ ,  $\text{id}(l)$ ,  $p_s$ ,  $p_i$ )
12 end

input :  $s$  fully qualified class name of the class to be loaded
         $l$  classloader used for initiating the loading of  $s$ 
output:  $LONG$  ID of the defining classloader of the loaded class
13 function loadAndGetClassLoaderID( $S\ s$ ,  $L\ l$ ) begin
14 | return  $\text{id}(\text{classloader}(\text{loadClass}(s, l)))$ 
15 end

```

Figure 5. Algorithm to force supertype loading and retrieve classloader IDs.

```

1 jclass loadClass(char *s, jobject l) {
2   if (l == NULL) return (*jni)->FindClass(jni,s);
3   jclass loaderClass = (*jni)->GetObjectClass(jni,l);
4   jmethodID methodID = (*jni)->GetMethodID(jni,
        loaderClass, "loadClass",
5     "(Ljava/lang/String;)Ljava/lang/Class;");
6   s = formatName(s); // Replace '/' with '.'
7   jstring className = (*jni)->NewStringUTF(jni,s);
8   return (jclass) (*jni)->CallObjectMethod(jni,l,
        methodID, className);
9 }

```

Figure 6. Native code of `loadClass(STRING  $s$ ,  $L\ l$ ): C`.

`loadClass` is shown in Figure 6.¹⁴ The purpose of this function is to call `l.loadClass(s)` from native code through JNI, triggering the loading of a class named s by classloader l (lines 3–8). Note that no class is loaded if l was already requested to load a class named s , and that l can delegate loading to another classloader. Special handling is needed if l is the bootstrap classloader. No object representing the bootstrap classloader is accessible in Java, thus no JNI calls can be made on l (l is null). In this case, we resort on the JNI function `FindClass` (line 2), which can look up and load classfiles located in the classpath using the bootstrap classloader.

According to the JVM specification, all supertypes of c must have been loaded before c is initialized and used by the target JVM. Our algorithm does not load any class that would not be loaded otherwise, resulting in no observable difference from the application perspective (apart from calls to the classloader possibly arriving in a slightly different order). Moreover, the JVM would initiate the loading of the

¹²For simplicity, error-handling code is not shown here.

¹³See the JVMTI documentation of the `ClassFileLoadHook` event for more information.

¹⁴In all C functions shown in the paper, the variables `jni` and `jvmti` refer to function pointers offered by the JNI or JVMTI API, respectively.

supertypes of c using the same defining classloader of c . Our algorithm follows the same approach, resulting in no alteration of classloader namespaces in the target application.

5.2 Classloader Namespaces

The design behind DiSL mandates that the instrumentation server be completely independent from the target application; thus, any information related to the classes used by the application must be explicitly sent to the server by the JVMTI agent. The communication protocol of the current DiSL release just sends (to the server) the byte array of the class to be instrumented, without any classloader information. As a result, the current DiSL release is unaware of classloader namespaces.

Our framework maintains different classloader namespaces in the instrumentation server, ensuring that homonym classes defined by different classloaders are handled correctly. While maintaining separate namespaces, the server must ensure that the DiSLClass of a supertype can be retrieved by a subtype even if the two are defined by different classloaders. This situation occurs frequently; for example any application class (loaded by a user-defined classloader) has at least one supertype (Object) in the `java.*` package, and classes within `java.*` can be loaded only by the bootstrap classloader [15]. To handle such cases correctly, we modify the agent-to-server communication protocol, including additional data to identify classes univocally. The rest of this section describes the rationale of our approach and some implementation details.

Retrieving and Sending Classloader Data Apart from the byte array representing c (needed for instrumenting the class), the agent in our framework sends (to the server) the defining-classloader ID of c and complete information (fully qualified name and defining-classloader ID) of all direct supertypes of c . This information is retrieved during the forced loading of supertypes, as shown in Figure 5. As soon as a supertype of c has been loaded, the agent retrieves the ID of its defining classloader (line 14). Note that such classloader can be different from l in case of classloader delegation. The agent stores the ID along with the fully qualified name of the supertype just loaded in a dedicated structure (p_s) for the superclass (line 3) or in a dedicated list (p_i) for the interfaces implemented by c (line 8). When all the supertypes have been loaded, p_s and p_i contain all the necessary information to identify each supertype of c univocally. Such information is then set sent to the server (line 11) along with the defining-classloader ID of c .

Figure 7 reports the C code for retrieving the defining classloader of c . The purpose of function `classloader` is to call `c.getClassLoader()` from native code (lines 3–6) through JNI. The call is avoided if c is defined inside the `java.*` package (line 2) for two reasons: 1) classes inside that package can only be loaded by the bootstrap classloader,

```
1 jobject classloader(jclass c) {
2   if (isInJavaPackage(c)) return NULL;
3   jclass kl = (*jni)->FindClass(jni, "java/lang/Class");
4   jmethodID methodID = (*jni)->GetMethodID(jni, kl,
5     "getClassLoader", "()Ljava/lang/ClassLoader;");
6   return (*jni)->CallObjectMethod(jni, c, methodID);
7 }
```

Figure 7. Native code of `classloader(C c): L`.

```
1 static jlong next_tag = 1;
2 jlong id(jobject l) {
3   if (l == NULL) return 0;
4   jlong *id = malloc(sizeof(jlong));
5   (*jvmti)->GetTag(jvmti, l, id);
6   if (*id != 0) return *id;
7   // l still untagged
8   jlong tag = next_tag++;
9   (*jvmti)->SetTag(jvmti, l, tag);
10  return tag;
11 }
```

Figure 8. Native code of `id(L l): LONG`. The code is atomically executed in a critical section (not shown here for simplicity).

and 2) the call can occur during JVM bootstrap (when Java classes are being initialized), interfering with their loading. In this case, the function returns null (to represent the bootstrap classloader).

Finally, Figure 8 details the process of retrieving the ID of classloader l . The agent makes use of the JVMTI heap tagging feature to assign or retrieve a unique long tag to/from each classloader, using the tags as IDs. According to the specification of the DiSL Reflection API, the bootstrap classloader has ID 0 (line 3). For all other classloaders, the agent first checks whether the classloader has already been tagged, returning the corresponding tag in such case (lines 4–6). Otherwise, it tags the classloader with a successive unique ID (lines 7–10), returning the assigned tag. This implementation ensures that classloaders are tagged in the same order they are used by the JVM, such that the application classloader has ID 1, while subsequent user-defined classloaders have successive IDs. The code is atomically executed in a critical section, to avoid any race condition.

Maintaining Classloader Namespaces The instrumentation server uses the additional data sent by our agent to maintain classloader namespaces, enabling the inspection of RSI. For each loaded class c , this process involves: 1) creating a DiSLClassLoader instance cl corresponding to the defining loader of c , 2) creating a DiSLClass instance kl corresponding to c , and 3) inserting kl into cl .

Figure 9 details the above operations. Note that this process occurs before c is actually instrumented (line 27), such that guards can access complete RSI during the instrumentation of c . First, the server obtains cl , (atomically) creating it


```

1 class DiSLClassImpl implements DiSLClass {
2     final private String className;
3     final private DiSLClassLoader classloader;
4     final private DiSLClass superclass;
5     final private List<DiSLClass> interfaces =
6         new LinkedList<>();
7
8     DiSLClassImpl(String name, DiSLClassLoader cl, P ps,
9         List<P> pi){
10         className = name;
11         classloader = cl;
12         superclass = getDiSLClass(ps);
13         for (P p: pi) { interfaces.add(getDiSLClass(p)); }
14     }
15
16     DiSLClass getDiSLClass(P p) {
17         DiSLClassLoader cl = getClassLoaderFromID(p.id);
18         return cl.forName(p.name);
19     }
20     ...
21 }
22
23 byte[] instrument(byte[] b, long i, P ps, List<P> pi){
24     DiSLClassLoader cl = getOrCreateClassLoader(i);
25     String name = parseClassName(b);
26     DiSLClass kl = new DiSLClassImpl(name, cl, ps, pi);
27     insert(cl, kl); // Inserts kl in the namespace of cl
28     ... // Instrument class
29 }

```

Figure 9. Code for maintaining classloader namespaces within the instrumentation server.

if c is the first class loaded by the classloader (line 23). Then, the server creates a new `DiSLClass` (line 25) using the information received from the agent. A `DiSLClass` instance is immutable, since all the information required to completely represent its state are known at creation time. Upon creation, an implementation of a `DiSLClass` (shown as `DiSLClassImpl` in the figure) needs to obtain the `DiSLClass` instances of its supertypes (lines 11–12). To this end, the server queries the defining classloader of each direct supertype for the corresponding `DiSLClass` (via method `getDiSLClass`; lines 15–18). Due to the forced loading of supertypes, `DiSLClass` and `DiSLClassLoader` lookup is guaranteed to succeed (apart from the corner cases described in Section 5.3). As last step, the server (atomically) inserts kl into cl (line 26).

5.3 Preprocessing Java Core Classes

The forced loading of supertypes makes RSI available in the instrumentation server. This process relies on JNI to call Java methods from native code (see Figure 6). Unfortunately, JNI is not available during the primordial phase of the JVM, when the JVM and the core Java classes are being initialized. As a result, the forced loading of supertypes cannot be ensured for all the Java classes loaded during the primordial phase of the JVM. This fact can lead to failures when retrieving RSI, if c is loaded during the primordial phase and one or more supertypes of c have not been loaded yet.

For example, in the JVM used in the evaluation (more details in Section 6), `java.lang.Thread` (which implements `java.lang.Runnable`) is loaded before `Runnable` during the primordial phase, leading to failures when a guard attempts to retrieve RSI of `Thread`.

Our framework tackles this issue by precomputing the `DiSLClass` of each class inside package `java.*`. The correctness of our approach is supported by the fact that all classes inside `java.*` can be loaded only by the bootstrap classloader (such that their defining classloader is always known). Precomputing reflective information of the Java core classes ensures the correct provisioning of complete RSI, assuming that the Java class library used for the precomputation and the one of the target application is the same.

6 Evaluation

In this section, we show that our approach can significantly reduce the overhead of an analysis with full code coverage.

Experimental Setup The target of our evaluation is task profiling on DaCapo [3], a well-known Java benchmark suite including several multi-threaded applications. We resort to `tgpr`¹⁵, a DiSL-based task profiler for the JVM. `tgpr` profiles every task (i.e., subtypes of `java.lang.Runnable` and `java.lang.Callable`) spawned by the target application. Amongst others, `tgpr` profiles the granularity of each task in terms of machine instructions and reference cycles using hardware performance counters. Such metrics are very sensitive to perturbations introduced by the inserted instrumentation code; hence, low profiling overhead is fundamental to collect accurate metrics. For each detected task, `tgpr` instruments the methods `run`, `call`, and the constructors.

For the evaluation, we use a server-class machine equipped with two NUMA nodes, each with an Intel Xeon E5-2680 (2.7 GHz) processor with 8 cores and 64 GB of RAM, running under Ubuntu 14.04 LTS. Turbo Boost and Hyper-threading are disabled, and the CPU governor is set to “performance”. We use Java 1.8.0_131 and the Java HotSpot 64-Bit Server VM (build 25.131-b11). We profile all the benchmarks included in the suite; the input size is set to “default”. The profiling overhead of eclipse and tomcat is not reported because the two benchmarks do not pass output validation on Java 8 (even without profiling).

Efficiency Here, we evaluate the speedup enabled by our framework. To this end, we compare the slowdown caused by `tgpr` when running on the current DiSL release (resorting to runtime checks), and when running on our extended framework (using the DiSL Reflection API). In both settings, we set the number of warmup iterations per benchmark to 14; we profile the 15th iteration to measure steady-state performance. The observed JVM is pinned to a different NUMA node from the one executing the instrumentation server, to

¹⁵<http://www.inf.usi.ch/phd/rosaa/tgpr/>

Figure 10. Profiling overhead on our framework (using the DiSL Reflection API) vs. the current DiSL release (resorting to runtime checks).

Table 3. Number of tasks detected by runtime checks and amount of checks executed without preprocessing.

Benchmark	# tasks	# checks	Benchmark	# checks
eclipse	225	2 212 511 891	h2	1 100 808 144
sunflow	17	2 532 326 406	jython	2 099 028 967
tomcat	54	1 676 032 506	luindex	9 702 354
tradebeans	3	421 535 666	lusearch	424 962 714
avroa	-	165 528	pmd	181 144
batik	-	57 835 086	tradesoap	2 890 739 487
fop	-	108 613 932	xalan	648 704 220

minimize interference of the instrumentation server with the observed JVM.

We show our results in Figure 10 (for each benchmark, we report the median overhead of 11 runs in separate JVM processes). The overhead introduced by our framework is low for most of the benchmarks, with slowdowns below 1% in batik, luindex, lusearch, and xalan; below 2% in fop, h2, jython, and sunflow; below 5% in tradebeans and tradesoap. The only benchmarks that suffer from moderate overhead are avroa and pm, with slowdowns around 14% and 12%, respectively. On the other hand, the overhead caused by the current DiSL release is always higher than the one of our new framework, with slowdowns that can be as high as 46% (fop). On average, using our framework incurs a minor slowdown of 3%, whereas the current DiSL release causes a non-negligible overhead of 19%. Such a high overhead stems from the expensive runtime checks needed in several methods when the DiSL Reflection API is not available. The average difference between the overhead introduced by our framework and the current DiSL release is 16%, while the maximum is 45% (fop).

Overall, the DiSL Reflection API can significantly speed up certain analyses wrt. the current DiSL release. In the case

of tgp, the high overhead introduced by runtime checks jeopardizes the accuracy of the collected hardware-level metrics, which are very sensitive to perturbations caused by the inserted instrumentation code. On the other hand, the metrics collected with our framework suffer much less from such perturbations (as indicated by the much lower overhead), enabling a more accurate analysis.

Preprocessing Java Core Classes Our framework preprocesses reflective information of the Java core classes to provide complete RSI. Here, we support the need for preprocessing such classes. Without preprocessing, a guard may fail to retrieve RSI of a Java class, making it impossible to determine whether the class shall be instrumented or not. As reported in Table 3 (second column), this situation affects task profiling in four benchmarks: eclipse (225 tasks), sunflow (17 tasks), tomcat (54 tasks), and tradebeans (3 tasks). The classes defining such tasks (e.g., `java.lang.Thread`) are loaded before their supertypes (e.g., `java.lang.Runnable`) during the primordial phase of the JVM, where the forced loading of supertypes is not possible. As a result, the guards in tgp cannot access their RSI at instrumentation-time. This can lead to several undetected tasks.

One approach to guarantee full code coverage in the analysis is inserting runtime checks in all classes where RSI is not available. Unfortunately, this approach causes the execution of several runtime checks in all benchmarks. As shown by Table 3 (third and fifth columns), this solution causes the execution of more than 2 billion checks in eclipse, jython, sunflow, and tradesoap, and more than 1 billion checks in h2 and tomcat, resulting in unnecessary slowdowns. Moreover, performing dynamic checks in inserted code is known to be deadlock-prone if such code is inserted in the Java class library (as in this case) [9]. Such checks can be totally avoided by preprocessing the Java core classes, ensuring that complete RSI is available even for classes loaded during the primordial phase. Thanks to preprocessing, our framework does not introduce any unnecessary slowdown, enabling more efficient analyses while providing full code coverage.

Coverage Finally, we evaluate the coverage of our approach in terms of the number of profiled tasks. To this end, we compare the coverage of an AspectJ-based recast of tgp (effectively excluding the packages `java.*`, `javax.*`, and `sun.reflect.*` from instrumentation) with the coverage of tgp when running on our framework. Table 4 reports our results. In all 14 benchmarks, our framework profiles the same number of tasks (in 3 benchmarks) or more tasks (in 11 benchmarks) than AspectJ, with notable differences in eclipse (229 tasks), pm (254 tasks), and tomcat (346 tasks). Considering the coverage offered by AspectJ vs. DiSL (right-most column), AspectJ profiles on average only 71% of the tasks spawned, failing to profile any task in the fop benchmark. The limited coverage results from the fact that AspectJ cannot instrument any task defined in the Java class library

Table 4. Coverage of our framework vs. an AspectJ-based analysis.

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

(such as Thread or FutureTask). On the other hand, our framework offers full code coverage, enabling the instrumentation of tasks inside the Java class library.

7 Discussion

In this section, we report additional features offered by our approach and discuss its limitations.

Reclamation of Classloader Namespaces In the target application, ClassLoader or Class objects may be reclaimed by the garbage collector. When this occurs, our approach ensures that the corresponding DiSLClassLoader or DiSLClass instances in the instrumentation server can be reclaimed as well, to avoid a memory leak in the instrumentation server. To this end, the JVMTI agent attached to the target application is notified when a tagged ClassLoader object has been freed by the garbage collector (via the JVMTI ObjectFree event). As our agent only tags ClassLoader objects, this event can only be sent upon the reclamation of a classloader. When the event occurs, the agent signals to the server the need for invalidating the namespace of the just freed classloader. In turn, the server removes any reference to the corresponding DiSLClassLoader and the DiSLClass instances composing the namespace, such that they will eventually be reclaimed by the garbage collector of the instrumentation-server JVM.

Freeing up a DiSLClass along with the corresponding DiSLClassLoader is fully compliant with the Java language specification [14], stating that a class can be reclaimed if and only if its defining classloader may be reclaimed (because classloaders maintain strong references to the loaded classes). As a result, our agent does not need to tag each Class object and notify the server upon the reclamation of each of them. While classloader reclamation does not strictly imply that classes in its namespace have been already reclaimed, they are by no means reachable by application code and will

eventually be reclaimed as well; hence, it is safe to reclaim the corresponding DiSLClass instances in the instrumentation server.

Preprocessing Classes Outside java.* Our framework can safely preprocess the Java core classes (i.e., classes inside the java.* package) because they are always defined by the bootstrap classloader [15]. To enable complete RSI, our framework must also preprocess the supertypes of each Java core class; however, some of them fall outside java.* (for example, we observed the presence of supertypes inside javax.*, sun.*, or com.sun.*). Our framework ensures a correct preprocessing of such classes, even if their defining classloader is (in principle) not regulated by the JVM specification. Indeed, not-yet-loaded supertypes would be loaded during subtype resolution, using the same defining classloader of the subtype. Since the subtype will always be loaded by the bootstrap classloader (that cannot delegate the loading), the supertypes will always be loaded by the bootstrap classloader, too. Consequently, preprocessing supertypes of the Java core classes is compliant with the JVM specification, even if they are declared outside java.*.

Instrumentation State The forced loading of supertypes employed in our approach ensures that a subtype will be instrumented after all its supertypes (except for the classes loaded during the primordial phase). Thus, custom *instrumentation state* can be passed from the instrumentation of a supertype to the instrumentation of a subtype. Our framework allows developers to attach custom instrumentation state to DiSLClass instances. The instrumentation state of supertypes is available through the DiSL Reflection API. Instrumentation state eases the propagation of instrumentation-time information to subtypes, such as properties of a class or bytecode-level statistics. Note that propagation of the instrumentation state cannot be guaranteed for classes loaded during the primordial phase, when subtypes may be instrumented before supertypes (see Section 5.3).

Checking Classloader IDs Our approach allows instrumenting the subtypes of a class based not only on its fully qualified name, but also on its defining classloader, by setting up guard checks against classloaders (see Figure 4). In this respect, our approach offers more flexibility than the AspectJ *subtype pattern* ('+' operator) that does not allow specifying the defining classloader.

Limitations Other analyses making use of the JVMTI heap tagging feature may interfere with our framework. In particular, such analyses could modify the order in which classloaders are tagged or could modify the tag of an already-tagged classloader (JVMTI supports only a single tag per object). Moreover, they could interfere with the reclamation of classloader namespaces, because the agent may receive ObjectFree events even for objects that are not classloaders. Such interference can be avoided by resorting to a weak-key

hash map¹⁶ in native code to map classloaders to IDs, instead of relying on JVMTI heap tagging. This solution would guarantee a correct assignment of IDs to classloaders. Moreover, it would still allow namespace reclamation by periodically checking whether a weak reference to a classloader points to a freed object (via the JNI method `IsSameObject`), notifying the server in such case.

While our framework enables the instrumentation of a method `m` based on complete RSI of the class where `m` is defined, it does not guarantee the availability of RSI for the arguments of `m`. Similarly, for call sites, RSI for the receiver and method arguments is not guaranteed available. This limitation arises from the fact that class loading is forced only for the supertypes of a class being loaded, but not for types that may occur as method arguments or as receivers of method calls. Enabling forced loading in the latter cases would likely result in a huge memory footprint and slow startup due to excessive class loading.

8 Related Work

Here, we compare our work with related frameworks for the JVM and discuss some profilers that can benefit from our work.

AspectJ [10] is a mainstream AOP language and weaver. In addition to AOP, AspectJ has been used for various instrumentation tasks. Since version 5, AspectJ provides a reflection API which is fully aware of the AspectJ type system [24]. However, the AspectJ compile-time weaver resorts to static analysis to precompute the type hierarchy, compromising accuracy and completeness of the available RSI (since information on classloaders and on dynamically loaded classes is missing). On the other hand, the load-time weaver of AspectJ (which can access complete RSI) is unable to weave aspects in the Java class library (as it performs the instrumentation within the observed JVM), resulting in limited code coverage. This issue is reported as a major limitation of AspectJ by the authors of the DJProf profiler [19]. In contrast, our approach enables the provision of accurate and complete RSI, while offering full code coverage. The AspectBench Compiler (abc) [1] is an extensible AspectJ compiler that eases the implementation of extensions to the AspectJ language and of optimizations. It uses the Polyglot [12] framework as its front-end and the Soot framework [25] as its back-end for improving code generation. As abc only supports compile-time instrumentation, only limited RSI is available.

Several runtime monitoring and verification tools for the JVM rely on AOP to weave the monitoring logic into the observed program, such as JavaMOP [7], Tracematches [4], or MARQ [20]. These framework are based on AspectJ and suffer from the same limitations. In particular, Tracematches is based on AspectJ's compile-time weaver (resulting in limited RSI), while JavaMOP and MARQ rely on the load-time

weaver of AspectJ (running in the same process as the target application, hence resulting in limited code coverage).

DPAC [6] is a dynamic analysis framework for the JVM. Similarly to DiSL, DPAC performs the instrumentation at load-time in a JVM running in a separate process, intercepting class loading through a JVMTI agent. To the best of our knowledge, DPAC does not provide access to complete RSI at instrumentation-time. Our approach can benefit frameworks like DPAC, providing complete RSI in the weaver.

Several bytecode engineering libraries facilitate bytecode instrumentation. Javassist [5] is a load-time bytecode manipulation library that enables structural reflection, i.e., altering the definition of classes or methods. Javassist provides convenient source-level abstractions and also supports a bytecode-level API allowing one to directly edit a classfile. ASM [17] and BCEL [23] provide low-level APIs to analyze, create, and transform Java class files. Java classes are represented as objects that contain all the information of the given class: constant pool, methods, fields, and bytecode instructions. Additionally, ASM supports load-time transformation of Java classes. Soot [25] is a bytecode optimization framework supporting multiple bytecode representations in order to simplify the analysis and the transformation of Java bytecode. Spoon [18] is a framework for program transformation and static analysis in Java, which reifies the program with respect to a meta-model. This allows direct access and modification of its structure at compile-time and allows inserting code using an AOP-based notation.

Several profilers based on Java bytecode instrumentation can benefit from our approach. LiLa [8] is a listener latency profiler that helps tuning interactive Java applications. LiLa captures all calls to and returns from listener methods via compile-time instrumentation. LiLa instruments every class implementing a listener interface, resorting to static analysis to determine the subtypes of such interfaces. As a result, the traces produced by LiLa may lack of full code coverage. This problem could be fixed by recasting LiLa in our framework, exploiting accurate and complete RSI at load-time. AkkaProf [21] profiles the utilization of Akka actors and the communication between them. To this end, AkkaProf instruments several methods and the constructors of every subtype of the abstract class `akka.actor.Actor`. AkkaProf is based on the current DiSL release; hence, it relies on expensive runtime checks to correctly identify subtypes of `Actor`. Such checks can be avoided with our framework to reduce the profiling overhead.

9 Conclusions

This paper presents a novel technique to reify the class hierarchy (including classloader namespaces) of an instrumented application within a separate instrumentation process. Our technique ensures that accurate and complete RSI is available

¹⁶Weak keys can be implemented by creating weak global references in JNI.

for each class to be instrumented. We implement our technique in the dynamic analysis framework DiSL. The instrumentation process in our enhanced framework is aware of classloader namespaces and correctly deals with homonym classes defined by different classloaders. An evaluation conducted on task profiling of the DaCapo benchmarks shows that our approach can lead to significant speedups (up to 45%) for an analysis with full code coverage.

Our extended DiSL framework is integrated in the latest open-source release of DiSL (available at <https://disl.ow2.org/>).

Acknowledgments

The research presented in this paper was supported by Oracle (ERO project 1332), by the Swiss National Science Foundation (project 200021_153560), and by the European Commission (contract ACP2-GA-2013-605442).

References

- [1] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhoták, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. 2005. Abc: An Extensible AspectJ Compiler. In *AOSD*. 87–98.
- [2] Walter Binder, Jarle Hulaas, and Philippe Moret. 2007. Advanced Java Bytecode Instrumentation. In *PPPJ*. 135–144.
- [3] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khang, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanović, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. 2006. The DaCapo Benchmarks: Java Benchmarking Development and Analysis. In *OOPSLA*. 169–190.
- [4] Eric Bodden, Laurie Hendren, Patrick Lam, Ondřej Lhoták, and No-mair A. Naeem. 2007. Collaborative Runtime Verification with Tracematches. In *RV*. 22–37.
- [5] Shigeru Chiba and Muga Nishizawa. 2003. An Easy-to-use Toolkit for Efficient Java Bytecode Translators. In *GPCE*. 364–376.
- [6] Yanyan Jiang, Chang Xu, and Xiaoxing Ma. 2013. DPAC: An Infrastructure for Dynamic Program Analysis of Concurrency Java Programs. In *MDS*. 2:1–2:6.
- [7] Dongyun Jin, Patrick O’Neil Meredith, Choonghwan Lee, and Grigore Roşu. 2012. JavaMOP: Efficient Parametric Runtime Monitoring Framework. In *ICSE*. 1427–1430.
- [8] Milan Jovic and Matthias Hauswirth. 2008. Measuring the Performance of Interactive Applications with Listener Latency Profiling. In *PPPJ*. 137–146.
- [9] Stephen Kell, Danilo Ansaloni, Walter Binder, and Lukáš Marek. 2012. The JVM is Not Observable Enough (and What to Do About It). In *VML*. 33–38.
- [10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. 2001. An overview of AspectJ. In *ECOOP*. 327–353.
- [11] Lukáš Marek, Alex Villazón, Yudi Zheng, Danilo Ansaloni, Walter Binder, and Zhengwei Qi. 2012. DiSL: A Domain-specific Language for Bytecode Instrumentation. In *AOSD*. 239–250.
- [12] Nathaniel Nystrom, Michael Clarkson, and Andrew C. Myers. 2003. Polyglot: An extensible compiler framework for Java. In *Compiler Construction*. Springer Berlin/Heidelberg, 138–152.
- [13] Oracle. 2015. Package java.lang.instrument. <https://docs.oracle.com/javase/8/docs/api/java/lang/instrument/package-summary.html>. (2015).
- [14] Oracle. 2015. The Java Language Specification. <https://docs.oracle.com/javase/specs/jls/se8/html/index.html>. (2015).
- [15] Oracle. 2015. The Java Virtual Machine Specification. <http://docs.oracle.com/javase/specs/jvms/se8/html/index.html>. (2015).
- [16] Oracle. 2015. The Reflection API. <https://docs.oracle.com/javase/tutorial/reflect/>. (2015).
- [17] OW2 Consortium. 2016. ASM. <http://asm.ow2.org/>. (2016).
- [18] Renaud Pawlak. 2006. Spoon: Compile-time Annotation Processing for Middleware. *IEEE Distributed Systems Online* 7, 11 (2006).
- [19] David J Pearce, Matthew Webster, Robert Berry, and Paul HJ Kelly. 2007. Profiling with AspectJ. *Software: Practice and Experience* 37, 7 (2007), 747–777.
- [20] Giles Reger, Helena Cuenca Cruz, and David Rydeheard. 2015. MarQ: Monitoring at Runtime with QEA. In *TACAS*. 596–610.
- [21] Andrea Rosà, Lydia Y. Chen, and Walter Binder. 2016. Actor Profiling in Virtual Execution Environments. *SIGPLAN Not.* 52, 3 (Oct. 2016), 36–46.
- [22] Simon Kitching. 2013. OSGi Classloading. <http://moi.vonos.net/java/osgi-classloaders/>. (2013).
- [23] The Apache Software Foundation. 2016. BCEL. <http://commons.apache.org/bcel/>. (2016).
- [24] The Eclipse Foundation. 2016. AjTypeSystem. <https://eclipse.org/aspectj/doc/next/adk15notebook/reflection.html>. (2016).
- [25] Raja Vallée-Rai, Etienne Gagnon, Laurie J. Hendren, Patrick Lam, Patrice Pominville, and Vijay Sundaresan. 2000. Optimizing Java Bytecode Using the Soot Framework: Is It Feasible?. In *CC*. 18–34.