Beyond Java 6: Reviving Reflection-Aware Static Analysis for Modern JVMs

Gauravsingh Sisodia
Sardar Patel Institute of Technology
Mumbai, India
gauravsingh.sisodia23@spit.ac.in

Poorna Teja Pasala
Indian Institute of Technology Bombay
Mumbai, India
poorna.teja.c2024@iitbombay.org

Aditya Anand
Indian Institute of Technology Bombay
Mumbai, India
adityaanand@cse.iitb.ac.in

Manas Thakur
Indian Institute of Technology Bombay
Mumbai, India
manas@cse.iitb.ac.in

Abstract

Static program analysis for Java is significantly hindered by the use of reflection and custom class loaders. To address this, the popular TamiFlex framework traces concrete targets of reflective calls and dumps loaded classes via a Play-out agent, enabling static analysis followed by re-insertion of transformed classes through a Play-in agent. While effective for Java 6, the evolution of the Java Virtual Machine (JVM) has rendered TamiFlex incompatible with Java versions released over the last decade. Modern runtime behaviors such as hidden-class introduction, randomized runtime-class generation, and class name collisions within frameworks cause the original agents to either crash or fail to converge.

In this paper, we present an updated TamiFlex for modern JVMs, addressing three key limitations of the original. We introduce a hidden-class dumping and loading mechanism that captures classes unavailable to the agents, a normalization strategy that reconciles bytecode variations across runs, and a classloader-aware isolation strategy that resolves class-name collisions across different classloaders. We evaluate our toolchain on the latest DaCapo benchmark suite running on Java 21 across two JVMs, and find that for all 22 benchmarks, our Play-out agent produces a reflection log, while our Play-in agent successfully re-inserts transformed classes. To demonstrate the practical value of the updated framework, we also extend a recent static escape analysis and find that both its coverage and precision improve noticeably.

CCS Concepts: • **Theory of computation → Program analysis**; • **Software and its engineering → Compilers**; *Dynamic analysis*; Object oriented languages.

VMIL '26, Oakland, CA, USA

ACM ISBN 979-8-4007-2932-4/2026/10
https://doi.org/10.1145/3840562.3844973

Keywords: reflection, static analysis, Java, hidden classes, call graphs

ACM Reference Format:
Gauravsingh Sisodia, Poorna Teja Pasala, Aditya Anand, and Manas Thakur. 2026. Beyond Java 6: Reviving Reflection-Aware Static Analysis for Modern JVMs. In *Proceedings of the 18th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages (VMIL '26), October 4–9, 2026, Oakland, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3840562.3844973

1 Introduction

Static program analyses often operate under a closed-world assumption, requiring access to the entire program to deliver sound results. This assumption breaks down in Java applications that utilize reflection to invoke methods or instantiate types determined only at run-time. Without visibility into these dynamic features, static analyses can produce incomplete call graphs and points-to sets, rendering subsequent optimizations and transformations unsound.

The de-facto approach to address static-analysis challenges in presence of these features is the tool TamiFlex [6], which consists of two Java instrumentation agents: a Play-out agent that logs reflective calls and dumps loaded class files to disk, and a Play-in agent that intercepts class loading to re-insert offline-transformed classes into the running application. This approach enabled static whole-program analysis on DaCapo 9.12-bach benchmark suite [4]. However, as the Java language and its virtual machine have evolved, the original TamiFlex framework no longer functions with modern suites like DaCapo 23.11-MR2-chopin [3] running on recent Java versions such as Java 21, and has never functioned across multiple popular Java Virtual Machines (JVMs). Consequently, state-of-the-art research in static analysis, including advanced pointer and points-to analyses [10, 11, 26] and JIT compilation optimizations [1, 2], remains severely constrained. To evaluate their approaches, researchers must rely on older benchmark suites such as DaCapo 2006 and 9.12 [4], which were released in 2006 and 2009, and respectively target Java versions 1.5 and 1.6 (and with some changes,

Java 1.8 partially, via DaCapo 9.12-MRI), or manually exclude benchmarks that use unsupported modern Java features.[1]

The evolution of the Java Virtual Machine (JVM) and modern application frameworks has introduced three major obstacles for reflection-aware static analysis:

- Since Java 15, lambda expressions are implemented using hidden classes, and the Play-out and Play-in agents fail to capture these classes.
- Modern frameworks use bytecode generation libraries that generate class files with randomized field names and non-deterministic method order across runs.
- Modern benchmarks load distinct class files that share the same fully qualified name through different class loaders.

In this paper, we present an updated TamiFlex framework specifically designed for modern JVMs. We update the tool to make it compatible with newer Java versions by introducing:

- A hidden-class dumping and loading mechanism that uses instrumentation to capture and load hidden classes at the point they are defined.
- A strategy that normalizes bytecode variation across runs for deterministic class-file generation.
- A class-loader aware dumping format that scopes each fully qualified class name with its class loader to prevent name collisions.

We evaluate our updated TamiFlex using the latest DaCapo 23.11-MR2-chopin benchmark suite and Soot [31]. We also explore the potential impact of our tool by leveraging its results to extend a recent static escape analysis designed for Java programs [1]. Our experiments show that:

- Our updated Play-out agent generates reflection logs and achieves convergence for all 22 benchmarks in the latest DaCapo suite, using JDK 21.
- Using our logs, the latest Soot successfully generates static call graphs for all the benchmarks, encompassing all dynamic call-graph edges for calls within the application.
- Our updated Play-in agent re-inserts dumped and subsequently transformed classes for all the benchmarks, across two popular JVM implementations.
- The extended static analysis achieves substantially higher coverage and precision.

We believe our updates will allow all Java static-analysis researchers to modernize their efforts and the engineering insights presented in this paper would help create new bridges between language designers and runtime engineers.

2 Background

2.1 Java Instrumentation

The Java Instrumentation API [22], introduced in Java 5, provides a mechanism for developers to add bytecode to classes as they are loaded into the Java Virtual Machine (JVM). This is achieved through Java agents, which are JAR files that the JVM loads prior to executing the main application.

When a Java agent is attached, the JVM invokes its `premain` method just before calling the application's actual `main` method. Within `premain`, the agent can register one or more `ClassFileTransformer` implementations. Whenever the JVM attempts to load a class, it passes the raw byte array defining that class to the `transform` method of all registered transformers. The transformer can then analyze and modify the byte array and return it to the JVM.

To facilitate this bytecode manipulation, tools rely on frameworks such as ASM [8]. ASM parses the byte array provided by the transform method, allows visiting and injecting instructions into methods and fields, and generates the resulting modified bytecode array for the JVM to execute.

2.2 TamiFlex Architecture

TamiFlex's architecture (See Figure 1) is composed of the Play-out agent, the Booster, and the Play-in agent. For the purposes of this paper we ignore the Booster and use only the Soot [31] framework for static analysis.

2.2.1 Play-out agent. The Play-out agent is responsible for creating a reflection log file with all the reflective calls encountered during the run and dumping all the loaded class files to disk. It registers two `ClassFileTransformer` implementations:

- The `ReflectionMonitor` transformer uses ASM to instrument the Java reflection API (e.g., `Class.forName`, `Method.invoke`). It inserts instrumentation right before a reflective method returns. The injected code calls a static method within the TamiFlex runtime library (shipped within the agent JAR). This runtime method extracts the concrete target of the reflective call, temporarily stores the information, and writes the reflection log to disk when a JVM shutdown hook is triggered.
- The `ClassDumper` transformer captures the byte arrays of all classes loaded by the application, including those generated dynamically, and dumps them into a class dump directory as standard class files.

A typical reflection log entry is stored as a semicolon-separated record containing the reflection method, the resolved target, the qualified name of the enclosing method, and the source line number of the reflective call site (when available). For example, the following line in the log (shown across multiple lines for readability):

```
Method.invoke;
<org.apache.fop.cli.Main:
void startFOP(java.lang.String[])>;
org.dacapo.harness.Fop.iterate; 41;
```

indicates that the method `iterate` invoked method `startFOP` reflectively via `Method.invoke` at line 41.

[1] The authors can corroborate these facts through their own work as well as through reviewing exercises of Java static analysis papers that suffer from these limitations.

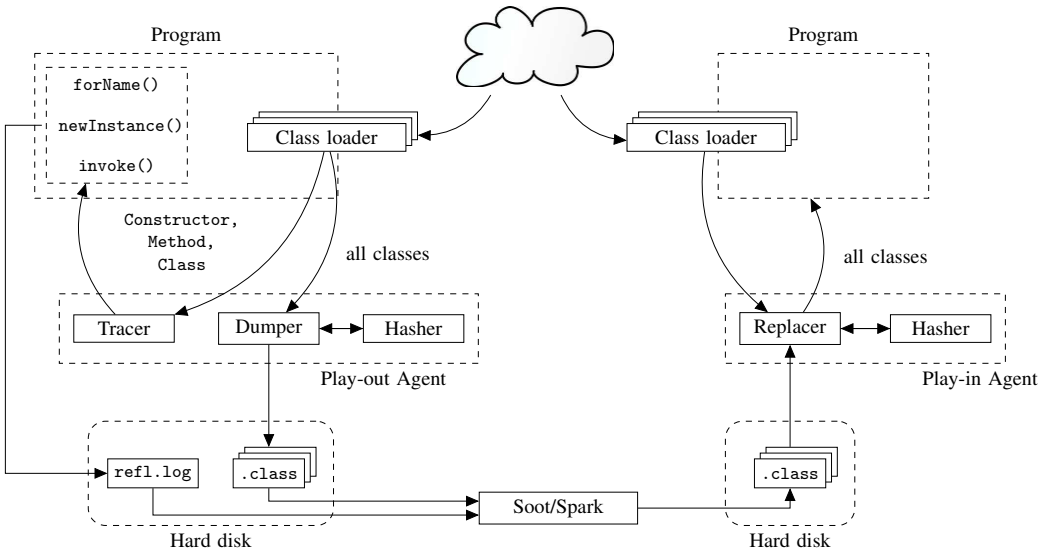

Figure 1. TamiFlex architecture. Reproduced from [5].

This agent is typically run for several iterations on the same benchmark, enriching the reflection log with new entries and the class dump directory with new classes on each iteration. We consider a benchmark to have converged when it no longer adds new reflection log entries or load new class files over multiple successive iterations.

2.2.2 Play-in agent. Once the dumped classes and reflection log have been utilized by a static analysis tool (such as Soot [31]) to perform transformations, the Play-in agent is used to execute the modified application. The Play-in agent uses a single `ClassFileTransformer` named `ClassReplacer`. When the application attempts to load a class, the `ClassReplacer` checks the local dump directory for an offline-transformed version. If a modified class file is found, it ignores the byte array passed in by the JVM and instead loads the bytes from the class file in the class dump directory. If the class is not found in the directory, it returns the original byte array unmodified. This can occur if the class was not encountered during the previous Play-out agent iterations.

3 Challenges Introduced by Modern Java

3.1 Newer Class Files

The original TamiFlex implementation uses ASM 3.2, which supports class files only up to Java 7 (class file version 51) [24]. Consequently, it is unable to parse the newer class files used by the DaCapo 23.11-MR2-chopin benchmark suite.

3.2 Default Interface Methods

Java 8 introduced default methods, allowing interfaces to provide concrete method implementations. Prior to Java 8, interfaces could not provide concrete method implementations, so resolving a virtual call only required searching the receiver's superclass hierarchy. This assumption was reflected in the original TamiFlex implementation, which resolves reflective `Method.invoke` calls by locating the corresponding method declaration in the receiver's class hierarchy.

With default methods, however, a class may inherit a concrete implementation directly from an interface without overriding it. In such a case, the receiver's class hierarchy does not contain a declaration of the invoked method; instead, the implementation is inherited from an interface. As a result, resolving a reflective invocation by searching only the superclass hierarchy may fail to identify the right method. This behavior occurs in the h2o benchmark of DaCapo chopin. TamiFlex fails to resolve the method `OperatingSystemMXBean.getTotalPhysicalMemorySize` for the receiver class `OperatingSystemImpl`, because it is inherited as a default method from the interface `OperatingSystemMXBean`.

3.3 Lambda Expressions

Lambda expressions provide a concise way to represent anonymous functions, allowing behavior to be passed as data without defining a separate class.

When `javac` encounters a lambda expression, it generates a private method containing the lambda's body, and inserts an `invokedynamic` instruction at the call site. The behavior of this instruction is controlled by a bootstrap method `LambdaMetafactory.metafactory`. The actual class implementing the functional interface is therefore not created at compile time, but rather generated at run-time by the JVM.

Since lambda expressions were introduced in Java 8, the internal implementation has changed across versions.

3.3.1 Java 8 to 14: VM anonymous classes. In the initial implementation, the JDK generated lambda classes used

the `Unsafe.defineAnonymousClass` API [19]. These VM anonymous classes were generated at run-time and had no corresponding class file on disk.

Naming convention. Calling `.getClass().getName()` on a lambda object produced names such as `example.Test$$Lambda$1/1418481495`, which consisted of the enclosing class name, a `$$Lambda$` marker, a per-host-class counter (e.g., 1), and an implementation-specific numerical identifier.

3.3.2 Java 15 to 20: Hidden classes.

With Java 15 [9], the JVM replaced the internal `Unsafe.defineAnonymousClass` API with the `Lookup.defineHiddenClass` mechanism. Hidden classes are intended for frameworks that generate bytecode at run-time. They are not discoverable through normal class loaders, cannot be loaded using `Class.forName`, and cannot be referenced symbolically from bytecode.

Naming convention. The generated class names changed to include a hexadecimal suffix assigned by the JVM, producing names like `T$$Lambda$1/0x00000008000c0840`.

3.3.3 Java 21 onwards: Simplified naming.

With Java 21, the JVM simplified the naming convention for hidden lambda classes by removing the sequential per-host-class counter.

Naming convention. Lambda class names now appear in the form `example.Test$$Lambda/0x00000008000c0840`.

If a lambda class is invoked reflectively via `Method.invoke`, it contributes an edge to the program's call graph, making it essential to capture both its bytecode and the corresponding reflection log entry. Although such reflective invocations are relatively uncommon, lambda classes frequently participate in other reflective operations. In DaCapo chopin, the spring and cassandra benchmarks make extensive use of reflective introspection methods, including `Class.getDeclaredMethods`, `Class.getDeclaredFields`, and `Method.getModifiers`, on lambda classes. For example, spring's `ReflectionUtils.doWithLocalMethods` method retrieves all methods of a class using `Class.getDeclaredMethods` and iterates over the resulting `Method[]` to invoke a `MethodCallback` on each method [29]. If lambda classes are not captured, they cannot appear in the reflection log, preventing a reflection-aware static analysis from precisely modeling such reflective queries. Capturing lambda classes therefore improves the precision of reflection-aware static analyses, even when the lambdas themselves are never invoked reflectively via `Method.invoke`.

Original TamiFlex relies on the Java Instrumentation API to capture classes as they are loaded. However, since Java 15, lambda expressions have been implemented using hidden classes, which are not visible to the API. As a result, these lambda classes are not captured by the original TamiFlex.

3.4 Runtime-Generated Classes

Modern Java programs often generate classes dynamically at run-time using bytecode manipulation libraries. Although the Play-out agent captures these classes as they are loaded, benchmarks such as spring, tradebeans, and tradesoap in the DaCapo 23.11-MR2-chopin benchmark suite generate certain classes with slightly different bytecode across executions. This non-determinism prevents the reflection log from converging over multiple benchmark runs, as each execution produces classes that are logically identical to previously generated ones but have slight differences in the bytecode. We discuss these differences in Section 4.4. Since this behavior was absent in DaCapo 9.12-bach, the original TamiFlex implementation did not require any mechanism to handle it.

3.5 Classes with Identical Names

In Java, multiple classes can share a fully qualified name if they are loaded by different class loaders. The original TamiFlex implementation ignored this scenario because these name collisions did not occur in the DaCapo 9.12-bach benchmark suite. However, in some benchmarks of the DaCapo 23.11-MR2-chopin suite (e.g., spring, cassandra, kafka, jython, pmd), this behavior is common. Classes share a name but contain different bytecode and originate from distinct class loaders. For example, in the spring benchmark, two different versions of the class `org.apache.commons.logging.LogFactory` are loaded by different class loaders. Although they share the same fully qualified name, their bytecodes differ significantly: one contains the method `createFactoryStore`, whereas the other does not.

4 Modernizing TamiFlex

4.1 Updating ASM

We upgraded the ASM library from version 3.2, which supports class files up to Java 7 (class file version 51), to version 9.9.1, which supports class files up to Java 26 (class file version 70) [24]. As part of this migration, we refactored the instrumentation code to replace the deprecated `ClassAdapter` and `MethodAdapter` APIs with the newer `ClassVisitor` and `MethodVisitor` APIs.

4.2 Extending Virtual Method Resolution

To support default interface methods, we extended TamiFlex's virtual method resolution algorithm. Because Java interfaces support multiple inheritance, resolving the correct inherited default method requires adhering to Java's Maximally Specific Rule (JLS §15.12.2.5) to handle ambiguities. The updated resolution logic operates in two steps:

1. Similar to the original implementation, the algorithm first iterates over the receiver's superclass chain using `Class.getDeclaredMethod`. This ensures that non-public class methods (which are inaccessible via `Class.getMethod`) are still located.
2. If no matching method is found in the class hierarchy, the implementation falls back to invoking `Class.getMethod` on the receiver's run-time class. As default methods are inherently `public`, this standard API natively traverses

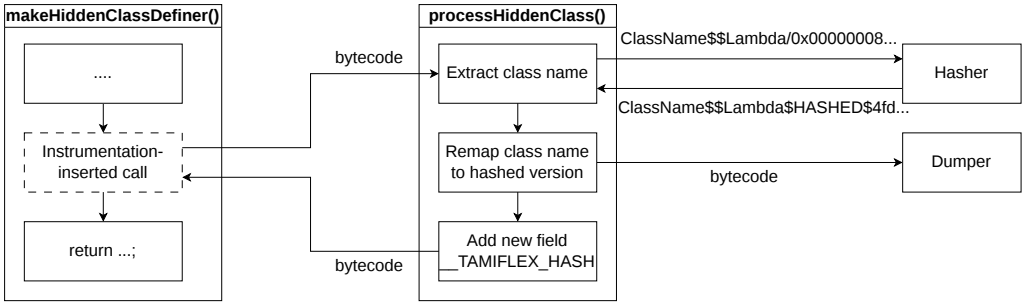

Figure 2. Instrumentation of `makeHiddenClassDefiner` method for dumping hidden classes.

the interface hierarchy and resolves the maximally specific default method exactly as the JVM does at runtime. If this returned method is verified as a default method (via `Method.isDefault`), it is recorded as the target.

This approach enables TamiFlex to resolve reflective calls that dispatch to inherited default interface methods while remaining faithful to JVM virtual dispatch semantics.

4.3 Handling Lambda Expressions

Since Java 15, lambda expressions have been implemented as hidden classes, making them invisible to the standard Java Instrumentation API. To address this, we developed a specialized mechanism that enables the Play-out agent to capture these classes and the Play-in agent to load them.

Hidden classes are instantiated via the `makeHiddenClassDefiner` method within the `java.lang.invoke.MethodHandles$Lookup` class. We use the Java Instrumentation API to intercept this class, inserting a call to a static method `processHiddenClass` in the TamiFlex runtime library just before it returns. This allows Play-out agent to access and dump the bytecode into the class dump directory. See Figure 2 for the instrumentation flow of `makeHiddenClassDefiner`. We employ a similar instrumentation strategy for the Play-in agent. The Play-in agent's runtime library method checks if an offline-transformed version of the passed bytecode exists in the class dump directory; if found, it returns the transformed bytes, otherwise it returns the original bytecode unmodified.

While this mechanism successfully dumps hidden classes, naming instability remains an issue. Since Java 21, lambda classes are assigned randomized runtime names, such as: `example.Test$$Lambda/0x00000008000c0840`. The hex suffix changes with every execution. To normalize the names, we strip the randomized suffix and append a `SHA-1` hash of the class bytecode. This creates a stable identifier for dumping and retrieving the class files across different runs.

However, this naming instability also affects the reflection log. The Play-out agent instruments reflection API methods (e.g., `Method.invoke`) to pass the target `java.lang.Class` object to the runtime logger. Because the logger only has access to the `Class` object (not its bytecode), calling `getName` on a lambda object returns the randomized run-time name, causing log convergence to fail. We resolved this by modifying the method `processHiddenClass`, to inject (using ASM) a new static String field named into the hidden class, assigning it the computed bytecode hash. While logging, the TamiFlex runtime library checks the target `Class` object for this field via the method `Class.getDeclaredField`. If the hash field exists, the logger records the normalized hash name; otherwise, it records the standard class name. See Figure 3 for the log entry generation process.

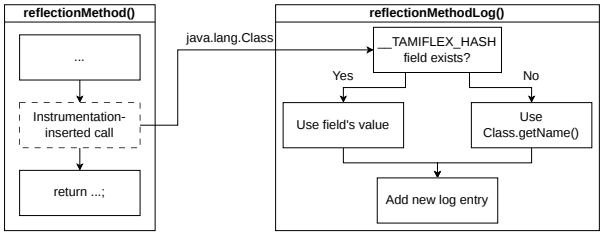

Figure 3. Instrumentation of a reflection method to add a new log entry to the reflection log.

4.4 Normalizing Runtime-Generated Classes

Some runtime-generated classes might have different bytecode on each run but logically represent the same class. Our analysis of these classes puts them into three categories.

4.4.1 Method ordering. The variation in bytecode arises occasionally from the difference in the ordering of the methods within the bytecode. Bytecode generation libraries (such as ByteBuddy [32]) rely on the reflective method `Class.getMethods` to inspect classes. However, this method's specification does not guarantee a deterministic return order [21]. To enforce consistent class generation, we used the Java Instrumentation API to intercept `java.lang.Class` and instrument the `getMethods` method to sort the returned array of methods deterministically. As shown in Algorithm 1, methods are ordered lexicographically by the concatenation of the method name, its JVM method descriptor, and its declaring class name, which together uniquely identify the method.

Algorithm 1 Deterministic ordering of methods

Require: Array of methods M
if M is `null` or $|M| \leq 1$ **then**
  return M
end if
$S \leftarrow$ copy of M
Sort S using comparator C
function $C(m_1, m_2)$
  $k_1 \leftarrow \text{name}(m_1) \parallel \text{desc}(m_1) \parallel \text{declaringClass}(m_1)$
  $k_2 \leftarrow \text{name}(m_2) \parallel \text{desc}(m_2) \parallel \text{declaringClass}(m_2)$
  return lexicographic comparison of k_1 and k_2
end function
return S

4.4.2 Randomized field names.

These bytecode generators can also assign randomized names to class fields, causing bytecode hashes and reflection targets to vary across executions. We resolved this by targeting the source of the randomness within the generation library. Specifically, we instrumented ByteBuddy's `net.bytebuddy.utility.RandomString` class's method `nextString` to return a constant string ("TAMIFLEX") rather than a random value.

4.4.3 Dynamic proxy classes.

Java's dynamic proxy mechanism generates classes at run-time that implement a specified set of interfaces [20]. For example, a generated proxy class may be named `jdk.proxy2.$Proxy5`. These generated names include non-deterministic counters. Because the same logical proxy class receives a different name on each execution, the Play-out agent continuously identifies them as new classes, hindering convergence.

To fix this naming instability, we implemented a mechanism in both the Play-out and Play-in agents. We use regular expressions to strip the non-deterministic numbers from the proxy name and append a SHA-1 hash of the class bytecode (computed by TamiFlex's Hasher component). This results in a stable naming format, such as `jdk.proxy.$Proxy$HASHED$<hash>`. However, stabilizing the name alone was insufficient; the Play-in agent initially failed to load the dumped proxy classes due to hash mismatches. We traced these bytecode variations back to two underlying sources of non-determinism, which we addressed as follows:

- **Constant pool and member ordering**: Generated proxy classes exhibited non-determinism in both their constant-pool layout and the ordering of class members. Although the generated classes were semantically identical, differences in constant-pool entry ordering (and consequently constant-pool indices referenced by the bytecode) resulted in different class files. We normalized the bytecode by rewriting each class with an ASM `ClassWriter`, which reconstructs the constant pool, and by explicitly sorting all class members by name and descriptor before computing the final hash.

Original

```
out/
|-- com/
|-- java/
|-- javax/
|-- jdk/
|-- org/
|-- sun/
|-- ...
`-- refl.log

Soot:
-cp out
-process-dir out
```

Updated

```
out/
|-- AppClassLoader/
|   |-- javax/
|   |-- jdk/
|   |-- ...
|   `-- org/
|-- null_loader/
|   |-- java/
|   |-- jdk/
|   |-- ...
|   `-- sun/
|-- DacapoClassLoader/
|   `-- org/
|-- ...
`-- refl.log

Soot:
-cp out/AppClassLoader:
    out/null_loader:
    out/DacapoClassLoader:...
-process-dir out/AppClassLoader
-process-dir out/null_loader
-process-dir out/DacapoClassLoader
```

Figure 4. Comparison of original and modified class-dump layouts and Soot configuration. Class loader names are abbreviated and directory trees are simplified for readability.

- **Proxy field assignment**: The JVM assigns cached `Method` objects to fields of the proxy class (e.g., `m0`, `m1`, `m2`) based on the array returned by `Class.getMethods`. Because this method's return order is non-deterministic [21], the internal structure of the proxy bytecode varied across runs. To normalize bytecode generation, we extended the `Class.getMethods` sorting instrumentation (previously applied to the Play-out agent) to the Play-in agent as well.

Combined, these three steps enforce deterministic bytecode generation, enabling the Play-out agent to converge on the benchmarks of the DaCapo chopin benchmark suite. Although these changes are specific to this suite, the general approach is to identify the sources of non-deterministic bytecode generation and eliminate them.

4.5 Handling Identically Named Classes

The original TamiFlex's Play-out agent maps packages directly to directories, allowing only one class file per fully qualified name. This structure causes conflicts during the Play-in agent run when an application requires different classes through different class loaders but the classes have the same fully qualified name. To resolve this, we modified the implementation to identify classes by a composite of their defining class loader and their fully qualified name. The new directory structure places the class loader name at the root level of the class dump directory, enabling the storage of multiple classes with the exact same name.

This change also requires modifications to Soot's configuration. As the updated class dump is organized by the defining class loader, the top-level class dump directory is no longer a valid `-process-dir` for Soot. Instead, the analysis must enumerate each class loader subdirectory and add it both to Soot's classpath (`-cp`) and as a separate `-process-dir` argument. Figure 4 compares the original and modified class dump layouts and their corresponding Soot configurations.

4.6 OpenJDK 21 Compatibility Fixes

We initially developed and tested our updated TamiFlex with OpenJ9 21. Migrating it to OpenJDK 21 revealed several JVM implementation-specific incompatibilities. Although these issues did not affect TamiFlex's overall design, addressing them was necessary to ensure correct execution on OpenJDK.

4.6.1 Avoiding class loading circularities. After registering the Java agent's `ClassFileTransformer`, OpenJDK lazily loads additional JDK classes upon first use. During agent initialization, iterating over a `LinkedList` triggered the loading of its internal iterator class `ListItr`. Because the transformer attempted to inspect the class while it was still being loaded, the JVM detected a circular dependency and threw a `ClassCircularityError`. OpenJ9 did not exhibit this behavior because the required JDK classes had already been loaded during its startup sequence. We resolved this issue by eagerly initializing `ListItr` before registering the transformer, preventing its load during instrumentation.

4.6.2 Resolving `Unsafe.getUnsafe`. When processing reflective calls to `Method.invoke`, TamiFlex resolves the actual target method based on the receiver object, rather than relying solely on the `Method` object on which `invoke` is called. We observed that OpenJDK attempts to resolve the method `sun.misc.Unsafe.getUnsafe`, whereas OpenJ9 does not encounter this method during this resolution. The two JVMs differ in the `sun.misc.Unsafe` methods encountered during this process. For the `jython` benchmark, OpenJDK attempts to resolve 89 distinct methods of this class, whereas OpenJ9 attempts to resolve only 4. The attempt to resolve `Unsafe.getUnsafe` gives a method-not-found error, causing the Play-out agent to fail. We therefore directly resolve `Unsafe.getUnsafe` when encountered.

4.6.3 Bootstrap class loading. The Play-out agent adds itself to the bootstrap class path to instrument bootstrap classes. On OpenJDK, however, the outer class `ClassDumper` was initially loaded by the system class loader, while its anonymous inner class (`ClassDumper$1`) was loaded by the bootstrap class loader. This mismatch caused an `IllegalAccessError`, as the inner class could not legally access its enclosing class loaded by a different loader. To ensure consistent loading, we modified the agent packaging so that the entire agent JAR is loaded by the bootstrap class loader by specifying the `Boot-Class-Path` manifest attribute.

5 Evaluation

We now present an evaluation that establishes the correctness and convergence properties of our updated TamiFlex, along with an impact study on a recent static analysis.

5.1 Experimental Setup

We evaluated our updated TamiFlex using the DaCapo 23.11-MR2-chopin benchmark suite. The evaluation was conducted on Java 21 using OpenJDK as well as OpenJ9 (both versions 21.0.9). Static call graphs were generated using our fork of Soot [27], based on upstream commit `12ef08e`, which incorporates two minor modifications: adding the missing `Field.toGenericString` reflection method and synchronizing `SmallNumberedMap`. We excluded Java 25 from this testing phase because several benchmarks within this suite do not support it. However, our changes generally work with Java 25 (the latest LTS version) as well. We used the "small" benchmark workload for all executions, as prior research [6] suggests that the number of reflective call sites encountered does not strongly correlate with the input size.

To systematically evaluate the framework, we applied a three-step experimental pipeline to each benchmark:

1. We execute the benchmark with the Play-out agent for 200 iterations (or until convergence) to generate the reflection log and a class dump directory of all loaded classes.
2. We provide this reflection log and class dump to our fork of the Soot framework [27] to verify that a static call graph can be successfully constructed using Spark [15].
3. We execute the benchmark a second time with the Play-in agent, pointing it to the class dump directory from step 1, to verify the successful re-insertion of offline classes.

For the spring benchmark, three reflection log entries must be removed before Soot analysis. These entries correspond to the class-loader name collision described in Section 3.5, where two versions of `org.apache.commons.logging.LogFactory` are loaded by different class loaders. Although our Play-out and Play-in agents distinguish such classes (Section 4.5), Soot cannot distinguish classes with the same fully qualified name across class loaders and therefore cannot resolve the referenced methods.

Our evaluation yielded the following results:

- The Play-out agent runs successfully and achieves convergence across all 22 benchmarks.
- Our fork of the Soot framework successfully generates static call graphs (via Spark) from the reflection logs provided by the Play-out agent.
- The Play-in agent successfully re-inserts the dumped class files into the running benchmark.

5.2 Convergence

We consider a benchmark to have converged when it no longer adds new reflection log entries or loads new class files

Table 1. Number of reflection-log entries and call-graph edges with our updated TamiFlex (with OpenJDK 21).

Benchmark	Reflection log entries	Spark CG size	CHA CG size
avrora	180	176210	756655
batik	297	318276	2039401
biojava	5065	282688	1397600
cassandra	34153	1037525	6449814
eclipse	12806	482905	2000071
fop	1333	437694	2515051
graphchi	3078	212257	932392
h2	353	326163	1104414
h2o	43872	579016	3574349
jme	2733	290884	2161030
jython	68098	6611880	7709867
kafka	14223	622754	6140955
luindex	683	259240	1073959
lusearch	573	208559	1003497
pmd	4001	247006	1274253
spring	200315	1550337	7636407
sunflow	427	248494	1955329
tomcat	22115	680369	3624451
tradebeans	82572	500425	14229057
tradesoap	83530	529368	14800030
xalan	1208	201688	1112103
zxing	1006	257138	1993190

over multiple successive iterations. This notion of convergence does not guarantee soundness, as it is based only on the executions observed during Play-out. Nevertheless, it provides a practical approximation that is substantially more informative than performing no reflection analysis at all.

To evaluate reflection log convergence, we executed the Play-out agent 200 times per benchmark on OpenJDK 21, tracking the discovery of new log entries and newly loaded classes across iterations. Some of the benchmarks—avrora, batik, biojava, fop, graphchi, jme, and luindex–reach a fixed point in a single iteration. The remaining benchmarks, with the exception of spring, successfully converge within 200 iterations. Because spring continued to discover new entries after 200 iterations, we extended it until 400 iterations, but it generates few classes once in a while even after.

Table 1 details the final metrics for all benchmarks using the small workload. Specifically, it reports the reflection log size (the total number of entries captured after the final Play-out iteration) alongside the sizes of the static call graphs computed by Soot using both the Spark and Class Hierarchy Analysis (CHA) algorithms. As shown, our updated TamiFlex allows one to analyze all the latest DaCapo benchmarks and obtain call graphs for static analysis using Soot.

Our updated TamiFlex framework is open-source and available on GitHub [28] for the static analysis community.

5.3 Correctness

TamiFlex-generated call graphs and reflection logs are sound only with respect to the recorded Play-out agent runs. If a code path is not triggered during the Play-out agent phase, it cannot be modeled by the subsequent static analysis.

To evaluate the correctness of the static call graphs generated by our updated toolchain, we compared them against dynamic call graphs. We captured dynamic call graphs using a custom JVMTI agent [23] and then used the ProBe call-graph differencing tool [14] to compare them with the static call graphs generated by Soot using its Spark [15] and CHA call-graph construction algorithms. ProBe verifies whether the dynamic call graph is entirely contained within the static call graph and reports any missing edges—those present in the dynamic call graph but not in the static call graph.

We performed this correctness evaluation on the entire DaCapo 23.11-MR2-chopin suite. However, a few programs failed to complete under the JVMTI agent because of native JVM crashes or benchmark timeouts. Consequently, Table 2 reports results only for the benchmarks that completed successfully. Note that the JVMTI agent is used only for this correctness evaluation and is not required by TamiFlex users. TamiFlex requires only the Play-out and Play-in agents.

5.3.1 Analysis of missing edges.

In an ideal scenario, a sound static analysis would yield zero missing call graph edges when compared against a dynamic call graph. To demonstrate the practical soundness of our updated TamiFlex, we conducted an in-depth analysis of the missing edges across all evaluated benchmarks.

We classify the missing edges into five distinct categories, out of which the first four correspond to edges not relevant to the static analysis of a given application; see Table 2.

- **JVM Mechanics and Class Loading (*JVM* column):** This category includes edges arising from the JVM's internal execution mechanisms rather than application semantics. Representative examples include:
  - `loadClass` and `findNative` methods in `java.lang.ClassLoader`, which implement JVM's lazy class loading and native method resolution, respectively.
  - `jdk.internal.module.ServicesCatalog`, which performs internal Java module resolution.
  - `jdk.internal.misc.Unsafe.ensureClassInitialized0`, which triggers JVM's class initialization.
  - Internal classes and methods belonging to the `java.lang.invoke`, including `MethodHandleNatives`, `Invokers$Holder.linkToTargetMethod`, and `LambdaForm$*`. These are JVM implementation details used to realize `invokedynamic` and method handle execution.
- **Tooling Artifacts and Known Limitations (*Tooling* column):** These edges arise from representation differences between the dynamic JVMTI agent, the differencing tool, and the static analysis framework. They include:
  - Edges involving covariant return types, which cannot be faithfully represented by ProBe.
  - Edges involving lambda classes. Although these edges exist in the static call graph, Soot generates lambda classes using its own internal `LambdaMetafactory` implementation, resulting in different class names.

Table 2. Missing edges in static call graphs generated using Spark and CHA implementations in Soot.

Benchmark	Analysis	Total	JVM	Tooling	TamiFlex	Internal	Remaining
avrora	Spark	623	392	83	13	127	8
	CHA	524	409	86	2	27	0
batik	Spark	1130	722	135	26	231	16
	CHA	1069	812	137	6	112	2
biojava	Spark	1561	811	264	69	274	143
	CHA	1304	953	273	4	47	27
eclipse	Spark	2927	1888	332	84	302	321
	CHA	2746	2042	338	16	82	268
fop	Spark	1651	1159	181	41	249	21
	CHA	1553	1252	188	12	98	3
graphchi	Spark	1121	526	173	48	257	117
	CHA	858	601	178	4	49	26
h2	Spark	900	551	139	26	166	18
	CHA	823	627	143	6	42	5
jme	Spark	969	540	151	84	174	20
	CHA	891	635	152	51	46	7
jython	Spark	2293	1435	315	142	287	114
	CHA	2095	1652	312	53	61	17
kafka	Spark	6735	3165	1590	180	433	1367
	CHA	8463	6469	1792	19	109	74
luindex	Spark	1425	918	190	24	171	122
	CHA	1352	1070	203	6	57	16
lusearch	Spark	1102	637	150	20	158	137
	CHA	1261	1039	164	5	47	6
pmd	Spark	1212	684	170	51	204	103
	CHA	996	739	173	3	34	47
spring	Spark	12248	7968	2571	428	483	798
	CHA	11536	8714	2587	37	112	86
sunflow	Spark	726	393	140	27	157	9
	CHA	649	463	141	6	39	0
tomcat	Spark	3875	2194	343	164	349	825
	CHA	3330	2649	351	33	97	200
xalan	Spark	682	416	91	29	135	11
	CHA	614	486	94	6	28	0
zxing	Spark	977	490	151	41	267	28
	CHA	831	574	154	4	93	6

  - Edges involving dynamically generated proxy classes. As described in Section 4.4.3, we append a hash to the names of these classes during the Play-out agent run, but the hashing is not performed by the JVMTI agent, leading to naming discrepancies.
- **TamiFlex Design Choices (*TamiFlex* column):** These missing edges are an intentional consequence of TamiFlex's reflection modeling strategy.
  - TamiFlex deliberately omits structural edges to the Reflection API itself (e.g., calls to `Method.invoke` and `Constructor.newInstance`).
  - Instead, TamiFlex connects the original reflective call site directly to the concrete target method resolved from the reflection log. Consequently, retaining an intermediate edge to the Reflection API will be redundant.
- **Internal Library Edges (*Internal* column):** These edges occur entirely within standard library implementation code (e.g., `java.*`, `javax.*`, `jdk.*`, and `sun.*`). Since they do not involve application code, they are not relevant to application-level static analysis.

Table 3. Comparison of the existing analysis (Base) and the additional analysis enabled by the updated TamiFlex+Soot infrastructure (Newly Enabled). Meth., Obj., and SA denote methods analyzed, objects analyzed, and objects eligible for stack allocation, respectively. Δ columns denote the additional results over Base.

	Base			Newly Enabled		
Bench.	Meth.	Obj.	SA	ΔMeth.	ΔObj.	ΔSA
avrora	37251	42538	3225	23331	16119	1592
biojava	34415	40376	2949	51482	51355	4217
eclipse	34176	42039	4272	80122	92602	8999
fop	77163	100861	4587	23624	20867	6417
graphchi	38963	48613	3898	25601	24878	1877
h2	34164	40261	2823	34410	35740	3133
h2o	92683	103226	10830	29364	30587	2987
jme	63470	81634	10533	23604	22731	2187
kafka	151626	138149	14947	44727	43892	9056
luindex	34162	40305	2816	35924	33259	3147
lusearch	34159	40417	2893	34407	27620	3005
pmd	66557	72544	5435	30623	29813	4446
sunflow	49167	57936	4399	31256	32824	4978
zxing	50986	63851	5332	30871	30135	4141

After excluding these four categories, only the edges reported in the *Remaining* column of Table 2 require further investigation. These correspond to application-level call graph edges that are present in the dynamic trace but absent from Soot's static call graph. Table 2 shows that the number of *Remaining* missing edges is consistently higher for Spark than for CHA. Since CHA successfully identifies most of these edges using the same reflection log and class dump directory, this discrepancy indicates limitations in Spark itself. The large number of missing edges observed in benchmarks—such as kafka, spring, and tomcat—when analyzed with Spark can therefore be attributed to limitations in Spark rather than a failure of TamiFlex to capture reflective calls.

The *Remaining* edges across both algorithms can be attributed to limitations in Spark and CHA, or to JVM implementation details that were not categorized by our analysis. These edges can also serve as useful inputs for improving both call-graph construction algorithms by identifying cases that are currently not handled. Nevertheless, they constitute only a negligible fraction of the corresponding static call graphs (Table 1), providing strong evidence for the practical soundness of the call graphs generated by the updated TamiFlex framework.

5.4 Improvements in Static Analysis

TamiFlex has primarily been used with Java 8 and the DaCapo 9.12-bach benchmark suite [4]. Several recent works (2023–2025) that rely on TamiFlex continue to evaluate their

analyses using this software stack, despite the availability of newer Java versions and benchmark suites [10, 11, 26].

Our updated implementation enables analyses that depend on TamiFlex to execute on modern Java runtimes and the latest DaCapo benchmark suite. In particular, analyses can now be performed using Java 21 and DaCapo-chopin [3].

To demonstrate this capability, we reproduced the escape analysis of [1] using our updated TamiFlex on the complete DaCapo 23.11-MR1-chopin benchmark suite. The original evaluation required two separate experimental configurations: OpenJ9 built with Java 8 for one subset of benchmarks and OpenJ9 built with Java 21 together with DaCapo 23.10-chopin for another. In contrast, our updated TamiFlex enables all the benchmarks to be analyzed using a single configuration based on Java 21 and DaCapo 23.11-MR1-chopin.

Table 3 compares configurations **Base** and **Newly Enabled**, where "Base" reports the escape analysis results obtained using the original TamiFlex+Soot on JDK 8, while "Newly Enabled" reports the additional results enabled by the updated TamiFlex+Soot on JDK 21 beyond the Base configuration. Specifically, it represents the additional methods that become reachable and analyzable, the additional objects analyzed within those methods, and the additional objects subsequently identified as eligible for stack allocation. As can be seen in Table 3, the updated configuration consistently enables the analysis of substantially more methods and objects across all benchmarks, leading to a significant increase in the number of objects identified as eligible for stack allocation. These improvements are primarily due to the enhancements made to TamiFlex, which allow additional methods to be included in the call graph, and the updates to Soot, which enable analysis of all JDK 21 library methods. However, it is possible that some of these come directly as a result of being able to analyze the newer JDK libraries.

We evaluated the run-time impact of these improved static analysis results using the run-time implementation from the recent work mentioned above [1] and observed corresponding gains in stack allocation. For example, biojava, which previously did not allocate any objects on the stack, now marks 4 allocation sites for stack allocation, resulting in 4.8M objects being allocated on the stack at run-time. Similarly, zxing, which previously marked 2 allocation sites and allocated 0.2M objects on the stack, now marks 5 allocation sites, increasing the number of stack-allocated objects to 0.4M. Overall, the updated TamiFlex and Soot significantly improve analysis coverage, enabling escape analysis to identify more stack-allocation opportunities.

6 Related Work

Reflection is among the notorious features known to prevent sound and precise static analysis for many languages [17], and particularly for Java programs because of its popularity in various benchmarks [3, 4] used to showcase advances in static-analysis research. Several works [13, 25] study the usage of reflection in modern Java applications, and conclude that reflective calls are not only prevalent, but also affect the soundness of static analysis significantly, using popular tools such as Soot [31], WALA [12] and DOOP [7].

Initial efforts to support static analysis of Java programs in presence of reflection proposed a combination of string-usage analysis and type inference, as a way to soundly approximate call targets at reflective sites [16, 18]. However, they are insufficient to infer precise call graphs for Java applications that use reflection even to choose the entry point of the program, or Android apps that use reflection heavily to choose paths in the program based on user interaction. Furthermore, these approaches have not been upgraded and tested in presence of modern Java features (starting with Java 8), the challenges behind handling which have been described in this paper.

The most popular way of proceeding with static analyses in presence of reflection has been the tool TamiFlex [6], which dumps a log of the reflective calls encountered during a program's execution, to be fed into static-analysis tools such as Soot. The analyses using this approach rely on an empirical convergence of the dumped reflection log along with dumping dynamically loaded class files to subsequently generate a call graph for static analysis. Our work brings this approach up to date with the latest Java features that have previously marred the generation and utilization of several kinds of reflective logs with the existing infrastructure.

The ideas we utilized to handle modern Java features in this paper can also be used to support the static analysis of modern Android versions. An interesting future direction, thus, would be to apply our ideas for extending tools such as DroidRA [30], which use an instrumentation technique much like TamiFlex to analyze Android apps.

7 Conclusion

This paper identified the challenges facing TamiFlex – arguably the most popular approach to enable static analysis of Java programs that use reflection – and described how we have subsequently upgraded the framework to work with the modern Java ecosystem. The effort not only involved a deep understanding of the language features under consideration and their connection with various phases of the reflection-instrumentation-analysis pipeline, but also a tremendous amount of engineering changes in the existing TamiFlex tool. The result of our paper is an upgraded reflection-analysis infrastructure that successfully analyzes all the latest DaCapo benchmarks, and supports both the popular Java runtimes (OpenJ9 and OpenJDK). We hope that our work would not only aid in the analysis of reflection usage in Java programs and possibly Android applications, but also help upgrade existing interprocedural static analyses in terms of their soundness and precision in the presence of reflection.

References

[1] Aditya Anand, Solai Adithya, Swapnil Rustagi, Priyam Seth, Vijay Sundaresan, Daryl Maier, V. Krishna Nandivada, and Manas Thakur. 2024. Optimistic Stack Allocation and Dynamic Heapification for Managed Runtimes. *Proc. ACM Program. Lang.* 8, PLDI, Article 159 (June 2024), 24 pages. doi:10.1145/3656389

[2] Aditya Anand, Vijay Sundaresan, Daryl Maier, and Manas Thakur. 2025. CoSSJIT: Combining Static Analysis and Speculation in JIT Compilers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 371 (Oct. 2025), 27 pages. doi:10.1145/3763149

[3] Stephen M. Blackburn, Zixian Cai, Rui Chen, Xi Yang, John Zhang, and John Zigman. 2025. Rethinking Java Performance Analysis. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Rotterdam, Netherlands) *(ASPLOS '25)*. Association for Computing Machinery, New York, NY, USA, 940–954. doi:10.1145/3669940.3707217

[4] 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. *SIGPLAN Not.* 41, 10 (Oct. 2006), 169–190. doi:10.1145/1167515.1167488

[5] Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. 2010. *Taming Reflection (Extended Version): Static Analysis in the Presence of Reflection and Custom Class Loaders.* Technical Report TUD-CS-2010-0066. Center for Advanced Security Research Darmstadt (CASED), Technische Universität Darmstadt. https://www.bodden.de/pubs/TUD-CS-2010-0066.pdf Technical Report.

[6] Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati, and Mira Mezini. 2011. Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *Proceedings of the 33rd International Conference on Software Engineering* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 241–250. doi:10.1145/1985793.1985827

[7] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*. ACM, 243–262. doi:10.1145/1640089.1640108

[8] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. 2002. ASM: A code manipulation tool to implement adaptable systems. (01 2002).

[9] Mandy Chung. 2020. JEP 371: Hidden Classes. https://openjdk.org/jeps/371. OpenJDK Java Enhancement Proposal 371..

[10] Dongjie He, Yujiang Gui, Wei Li, Yonggang Tao, Changwei Zou, Yulei Sui, and Jingling Xue. 2023. A Container-Usage-Pattern-Based Context Debloating Approach for Object-Sensitive Pointer Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 256 (Oct. 2023), 30 pages. doi:10.1145/3622832

[11] Dongjie He, Jingbo Lu, and Jingling Xue. 2023. IFDS-based Context Debloating for Object-Sensitive Pointer Analysis. *ACM Trans. Softw. Eng. Methodol.* 32, 4, Article 101 (May 2023), 44 pages. doi:10.1145/3579641

[12] IBM T. J. Watson Research Center. 2024. WALA: T. J. Watson Libraries for Analysis. https://github.com/wala/WALA.

[13] Davy Landman, Alexander Serebrenik, and Jurgen J. Vinju. 2017. Challenges for static analysis of Java reflection: literature review and empirical study. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 507–518. doi:10.1109/ICSE.2017.53

[14] Ondrej Lhoták. 2007. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering* (San Diego, California, USA) *(PASTE '07)*. Association for Computing Machinery, New York, NY, USA, 37–42. doi:10.1145/1251535.1251542

[15] Ondřej Lhoták and Laurie Hendren. 2003. *Scaling Java Points-to Analysis Using Spark*. Vol. 2622. 153–169. doi:10.1007/3-540-36579-6_12

[16] Yue Li, Tian Tan, and Jingling Xue. 2019. Understanding and Analyzing Java Reflection. *ACM Trans. Softw. Eng. Methodol.* 28, 2, Article 7 (Feb. 2019), 50 pages. doi:10.1145/3295739

[17] Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In defense of soundiness: a manifesto. *Commun. ACM* 58, 2 (Jan. 2015), 44–46. doi:10.1145/2644805

[18] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection Analysis for Java. In *Programming Languages and Systems*, Kwangkeun Yi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 139–160.

[19] OpenJDK. 2014. InnerClassLambdaMetafactory.java (JDK 8). https://github.com/openjdk/jdk8/blob/6a383433a9f4661a96a90b2a4c7b5b9a85720031/jdk/src/share/classes/java/lang/invoke/InnerClassLambdaMetafactory.java. The method `spinInnerClass` invokes `Unsafe.defineAnonymousClass` to define lambda implementation classes. See line number 324.

[20] Oracle. [n. d.]. Dynamic Proxy Classes. https://docs.oracle.com/javase/8/docs/technotes/guides/reflection/proxy.html. Java SE 8 Documentation.

[21] Oracle. 2023. Class (Java SE 21 & JDK 21). https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/lang/Class.html.

[22] Oracle 2023. *Interface Instrumentation (Java SE 21 & JDK 21)*. Oracle. https://docs.oracle.com/en/java/javase/21/docs/api/java.instrument/java/lang/instrument/Instrumentation.html

[23] Oracle Corporation. 2023. JVM Tool Interface (JVM TI) Version 21.0.0. https://docs.oracle.com/en/java/javase/21/docs/specs/jvmti.html.

[24] OW2 ASM Project. 2026. ASM Version History. https://asm.ow2.io/versions.html.

[25] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. 2018. Systematic evaluation of the unsoundness of call graph construction algorithms for Java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops* (Amsterdam, Netherlands) *(ISSTA '18)*. Association for Computing Machinery, New York, NY, USA, 107–112. doi:10.1145/3236454.3236503

[26] Chenghang Shi, Dongjie He, Haofeng Li, Jie Lu, Lian Li, and Jingling Xue. 2025. Fast Client-Driven CFL-Reachability via Regularization-Based Graph Simplification. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 287 (Oct. 2025), 28 pages. doi:10.1145/3763065

[27] Gauravsingh Sisodia, Poorna Teja Pasala, Aditya Anand, and Manas Thakur. 2026. Soot: Fork for Updated TamiFlex. https://github.com/CompL-Research/soot-for-tamiflex. GitHub repository.

[28] Gauravsingh Sisodia, Poorna Teja Pasala, Aditya Anand, and Manas Thakur. 2026. TamiFlex: Updated TamiFlex for Java 21. https://github.com/CompL-Research/tamiflex. GitHub repository.

[29] Spring Framework Contributors. 2026. ReflectionUtils.java. https://github.com/spring-projects/spring-framework/blob/e8729d043887bf0d0baf91e062e909b56eb2b708/spring-core/src/main/java/org/springframework/util/ReflectionUtils.java. Line number 319 has the method doWithLocalMethods..

[30] Xiaoyu Sun, Li Li, Tegawendé F. Bissyandé, Jacques Klein, Damien Octeau, and John Grundy. 2021. Taming Reflection: An Essential Step Toward Whole-program Analysis of Android Apps. *ACM Trans. Softw. Eng. Methodol.* 30, 3, Article 32 (April 2021), 36 pages. doi:10.1145/3440033

[31] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. 2010. Soot: a Java bytecode optimization framework. In *CASCON First Decade High Impact Papers* (Toronto, Ontario, Canada) *(CASCON '10)*. IBM Corp., USA, 214–224. doi:10.1145/1925805.1925818

[32] Rafael Winterhalter. 2026. Byte Buddy. https://bytebuddy.net. Runtime code generation library for the Java Virtual Machine..

Received 2026-08-14; accepted 2026-08-27