# Taming Reflection

## Aiding Static Analysis in the Presence of Reflection and Custom Class Loaders

Eric Bodden, Andreas Sewe, Jan Sinschek, Hela Oueslati and Mira Mezini
Software Technology Group, Technische Universität Darmstadt
Center for Advanced Security Research Darmstadt (CASED)
bodden@acm.org

## ABSTRACT

Static program analyses and transformations for Java face many problems when analyzing programs that use reflection or custom class loaders: How can a static analysis know which reflective calls the program will execute? How can it get hold of classes that the program loads from remote locations or even generates on the fly? And if the analysis transforms classes, how can these classes be re-inserted into a program that uses custom class loaders?

In this paper, we present TamiFlex, a tool chain that offers a partial but often effective solution to these problems. With TamiFlex, programmers can use existing static-analysis tools to produce results that are sound at least with respect to a set of recorded program runs. TamiFlex inserts runtime checks into the program that warn the user in case the program executes reflective calls that the analysis did not take into account. TamiFlex further allows programmers to re-insert offline-transformed classes into a program.

We evaluate TamiFlex in two scenarios: benchmarking with the DaCapo benchmark suite and analysing large-scale interactive applications. For the latter, TamiFlex significantly improves code coverage of the static analyses, while for the former our approach even appears complete: the inserted runtime checks issue no warning. Hence, for the first time, TamiFlex enables sound static whole-program analyses on DaCapo. During this process, TamiFlex usually incurs less than 10% runtime overhead.

## Categories and Subject Descriptors

F.3.2 [**Semantics of Programming Languages**]: Program Analysis

## General Terms

Algorithms, Experimentation, Measurement, Reliability

## Keywords

Reflection, static analysis, dynamic class loading, dynamic class loaders, native code, tracing

## 1. INTRODUCTION

Researchers have developed many useful static program analyses, ranging from analyses that compute call graphs [23, 25] and relevant program slices [28] to analyses that determine the shape of custom data structures [18, 29], prove parameter immutability [3], track an object's typestate [7, 15, 16, 26], or support model checking in program verification [10]. Some of these are also coupled with program transformations, i.e., they optimize or instrument the analyzed program. As one example, in previous work [7, 15, 26] we and others have used static typestate-analysis information to optimize runtime monitors for typestate properties [30].

Virtually all of these analyses are whole-program analyses; the analyses must consider the entire program to deliver sound results. This is because most analyses operate under a closed-world assumption: for instance, the analyses frequently assume that a call graph is complete in the sense that if a call graph contains no edge from a method `m` to a method `n` then it can never be the case that `m` calls `n`.

Obtaining information about the "whole program" yields many problems when analyzing Java programs that use reflection or load or generate classes using custom class loaders. In Figure 1, we give an example program that, although simplified, highlights many of these problems. In the example, the `Generator` in line 4 runtime-generates and returns the class object for `Foo$42` (lines 10–14). The invoke statement at line 6 therefore calls `Foo$42.foo(c)`, closing the `Connection` object referenced by `c`. Unfortunately, line 7 writes to the closed connection, leading to a runtime error.

Static typestate analyses [30] allow programmers to obtain compile-time warnings for such situations. In the past, we ourselves have developed such an analysis. [6] It should warn

```
1  class Main {
2      public static void main(String[] args) throws Exception {
3          Connection c = new Connection();
4          Class clazz = Generator.makeClass(); // returns Foo$42
5          Method m = clazz.getMethod("foo", Connection.class);
6          m.invoke(null, c); // calls Foo$42.foo(..)
7          c.write("what a risky method call: c is closed!");
8      }
9  }
10 class Foo$42 {
11     public static void foo(Connection c) {
12         c.close();
13     }
14 }
```

Figure 1: Example program using dynamic class generation

the programmer of this program that the write at line 7 may go to a closed connection. However, current static analyses have no way of being aware of runtime-generated classes such as `Foo$42` and frequently make the unsound assumption that such classes execute no code. In our example, this would cause virtually any state-of-the-art typestate analysis to conclude that the above program is error-free, i.e., cannot write to closed connections—clearly an undesirable result. Even worse, the problem is not restricted to generated classes; when programs use custom class loaders, e.g., to load classes from nested JAR files or from remote locations, then static analyses will usually be unable to find these classes and yield equally unsound results.

And even if static analyses were made aware of reflective calls and reflectively loaded classes, further problems would remain. For instance, our earlier static typestate analysis [6] contains a runtime component: if the analysis detects that the program may write to a closed connection, then the analysis instruments the program with runtime checks that will issue a runtime error message in case the program really does write to a closed connection at this point. But in our example, the analysis would have to add checks to the generated class `Foo$42` because it is this class that closes the connection. Even if the static analysis has access to this class and can analyze and transform it, one needs to re-insert the instrumented version of `Foo$42` back into the program.

Current static analyses and transformations do not deal with these problems and thus virtually all existing static whole-program analyses are inherently unsound for the majority of Java programs. While one cannot solve this problem in full generality, we present TamiFlex, a tool chain for "taming reflection" as a pragmatic partial solution that works surprisingly well for a wide range of real-world Java programs. TamiFlex consists of two novel runtime components, the Play-out Agent and the Play-in Agent, which execute alongside the running program, and an offline component, called the Booster.

The Play-out Agent logs reflective calls into a reflection log, and gathers the classes that the program loads, be it through custom class loaders or on-the-fly code generation. This offers a partial solution to the first problem we mentioned: the Play-out Agent collects all loaded classes and all information about reflective calls for *every recorded run*.

But simply logging reflective calls is not enough as most static-analysis tools would be unable to interpret these logs. To avoid having to extend these tools, we therefore developed the Booster. Given the recorded log and class files, the Booster produces an enriched version of the program by "materializing" recorded reflective method calls into regular Java method calls. This enriched program version, while guaranteed to be behaviorally equivalent to the original program, aids static analyses: the analyses will usually take the materialized method calls into account whereas they would have unsoundly ignored reflective method calls.

The Booster fulfils a second important task. As mentioned, static analyses based on TamiFlex logs are sound only with respect to a set of recorded program runs. Thus, what happens if a later program run diverges from the runs previously recorded? The Booster inserts runtime checks into the program that will automatically warn the user in these cases; as long as no warnings are issued, the program is known to operate under sound assumptions.

If users of TamiFlex simply wish to analyze a program statically, without transforming the program, they only need to use the Play-out Agent and the Booster, not the Play-in Agent. Users can simply feed the enriched class files to any static-analysis tool. In many cases, however, users may want to use static-analysis results to transform classes, e.g., to optimize or instrument them. In these cases, one faces the problem of re-packaging the transformed classes in such a way that the original program finds the classes where it expects them. Without special tool support, this can be either hard, for instance if the program loads the classes from a remote location, or even impossible, if the program generates the classes on the fly. The Play-in Agent solves this problem by re-inserting offline-transformed classes into a running program. The agent even replaces classes that an application generates at runtime.

We study two applications of TamiFlex: evaluating static optimizations using the DaCapo benchmark suite [5] and statically analyzing nine different interactive real-world applications. Static analysis without considering reflection produces inherently unsound results in both cases. As our results show, with TamiFlex our approach appears complete for DaCapo: the runtime checks in the enriched program code never triggered for any run that we observed. Hence, for the first time, our tool chain enables researchers to conduct static whole-program analysis on this version of DaCapo. We further show that TamiFlex induces a runtime overhead of usually below 10%. The Play-in Agent in particular induces no overhead after all classes have been loaded. Because of this, researchers can effectively use TamiFlex to run statically optimized versions of DaCapo: when iterating a benchmark, the Play-in Agent will only cause a runtime overhead during the initial (warm-up) iteration.

For the other nine Java programs TamiFlex may need to record multiple program runs to cover a sufficient amount of reflective calls. However, as we show, often just a few recorded runs can significantly improve code coverage of the static analysis. Therefore, while TamiFlex does have its limitations, it greatly improves over the state of the art even for programs that highly depend on user input.

To summarize, this paper presents the following original contributions:

- The design and implementation of two Java instrumentation agents that can emit all loaded classes into a local class repository, log reflective method calls, and re-insert offline-transformed classes into a program, even if the program uses custom class loaders. The agents use the `java.lang.instrument` interface, making them compatible with every modern Java virtual machine.

- A Booster component that enriches a program's class files by "materializing" reflective method calls, and at the same time inserts runtime checks that will warn the user when a program executes reflective calls that the Play-out Agent had not recorded.

- In combination, the first solution to allow researchers to conduct static whole-program analysis and transformation on the current 9.12-bach release of DaCapo.

- A set of experiments that prove that our tool chain is efficient, yields call graphs for all DaCapo benchmarks that are sound for all benchmark runs, and largely improves analysis coverage for nine other, interactive Java applications.

TamiFlex, all our experimental data, and all tools to reproduce this data are available to the public at:

`http://tamiflex.googlecode.com/`

The remainder of this paper is organized as follows. We explain TamiFlex in Section 2. We report on our experiments in Section 3, discuss related work in Section 4, and conclude in Section 5. A Technical Report [8] gives additional information on the implementation and experiments.

## 2. TAMIFLEX

We first describe the overall architecture of TamiFlex and then detail its individual components.

### 2.1 Architecture of TamiFlex

Figure 2 on the next page gives an overview of TamiFlex's architecture. On the top left, we show a program that potentially uses custom class loaders to load classes from arbitrary locations (the cloud), or even to generate classes on the fly. The program may further call methods such as `Constructor.newInstance()` or `Method.invoke()` to construct objects or invoke methods through reflection.

Let us now assume that the program executes with our first instrumentation agent installed, the Play-out Agent, which Figure 2 shows below the program. In this agent, the Tracer transforms the classes `Class`, `Method` and `Constructor` so that calls to methods such as `Method.invoke()` generate entries in a log file (shown on the bottom left). The agent further comprises a Dumper component, which writes into a local repository, i.e., a flat directory, all classes that the program loads. This includes classes that the program's class loaders have generated on the fly. Some runtime-generated classes bear randomized names. To allow re-identification of such classes across multiple runs, the Dumper assigns normalized names to these classes. To this effect, the Dumper communicates with a Hasher component (see Section 2.2).

Executing a program with the Play-out Agent will result in a repository that contains a reflection log file and all classes that the program loaded during the observed run. To obtain a reasonably complete log file and set of classes, users can run the program multiple times. The agent will update the log, appending information about reflective calls that had not previously been observed, and dump classes that had not been loaded on previous runs. The idea is to repeat this process until no changes are observed any longer.

There are now two options to enable static analyses based on the recorded information. First, shown dotted, one can derive a specialized static analysis that is TamiFlex-aware. In our Technical Report [8] we describe such a solution, tied to Soot [33] and Spark [23]. Extending the analysis is non-trivial, however, and would have to be repeated for every static-analysis tool. Therefore, we propose another approach: to use the Booster to convert the original program (as defined by the collected class files) into an enriched program version. The Booster inserts reflective method calls as additional regular method calls, based on the information from the reflection log. Simultaneously, the Booster inserts runtime checks that will issue a warning in case the program executes a reflective call that the log does not contain.

Next, users can feed the enriched program to any static-analysis tool to conduct static analyses, and to transform, e.g., optimize or instrument, the program code. Because the recorded reflective calls now appear as normal method calls in the program's code, typical static analyses will correctly pick up the respective calls during call-graph construction.

The right-hand side of Figure 2 shows what happens when the user runs the program with the second agent, the Play-in Agent, installed. Whenever the original program is about to load a class `clazz`, a Replacer within the agent tries to retrieve the offline-transformed version of `clazz` from the local repository. For classes that bear a randomly generated class name, the agent asks the Hasher component to compute its normalized name. This causes the Replacer to look for the offline-transformed class under the same normalized name that the Dumper used to store the class. If the Replacer finds a class in the repository, it replaces the originally-loaded (or generated) class with the found class on the fly. Otherwise, i.e., if the Replacer cannot find an appropriate class file, for instance because no such class was loaded on previous runs, the Replacer executes no replacement: In this case the program will instantiate the class that the class loader originally generated or loaded from "the cloud". Optionally, users receive a warning message in such situations.

Note the flexibility of this design; TamiFlex works with any Java virtual machine that supports transforming classes through the `java.lang.instrument` application programming interface. Through this interface, our agents are able to write out and replace classes that the program loads. With the aid of our Hasher component, this even works in cases where the program generates classes with randomized names. The Booster makes TamiFlex compatible with virtually every static-analysis tool able to process Java bytecode.

The current implementation of the Play-out Agent has some limitations that can be lifted in future work. There may be other native methods calling back into Java bytecode, not just reflective calls. As our experiments show, such calls are usually limited to finalizers and shutdown hooks (Sec. 3.1). Support for such call edges could easily be added. Further, programmers may use reflection to change the values stored in fields. Such store operations may make field-sensitive program analyses unsound. In future work, one could extend the Play-out Agent to record such stores and the Booster to materialize such stores similar to how we do now for other reflective calls.

### 2.2 Play-out Agent

Figure 2 shows the Play-out Agent on the left-hand side. Before the program starts up, the Play-out Agent registers two class-file transformers, the Tracer and the Dumper, with the virtual machine (VM). The VM notifies such transformers about every class that the program loads, no matter which class loader is used, including such classes that the program generates on the fly. When the program executes, the Dumper records the byte arrays of all loaded classes and writes them as `.class` files to disk. At the same time, the Tracer waits for the classes `Class`, `Constructor` or `Method` to be loaded. When encountering one of these classes, it instruments the methods `forName`, `newInstance` and `invoke` (modifying their bytecode) so that they will create a log entry whenever they execute. We make sure to insert the logging code at the end of these methods, just before every `return` statement. This ensures that we do not log erroneous executions of these methods, which throw an exception, e.g., because a certain class or method could not be found.

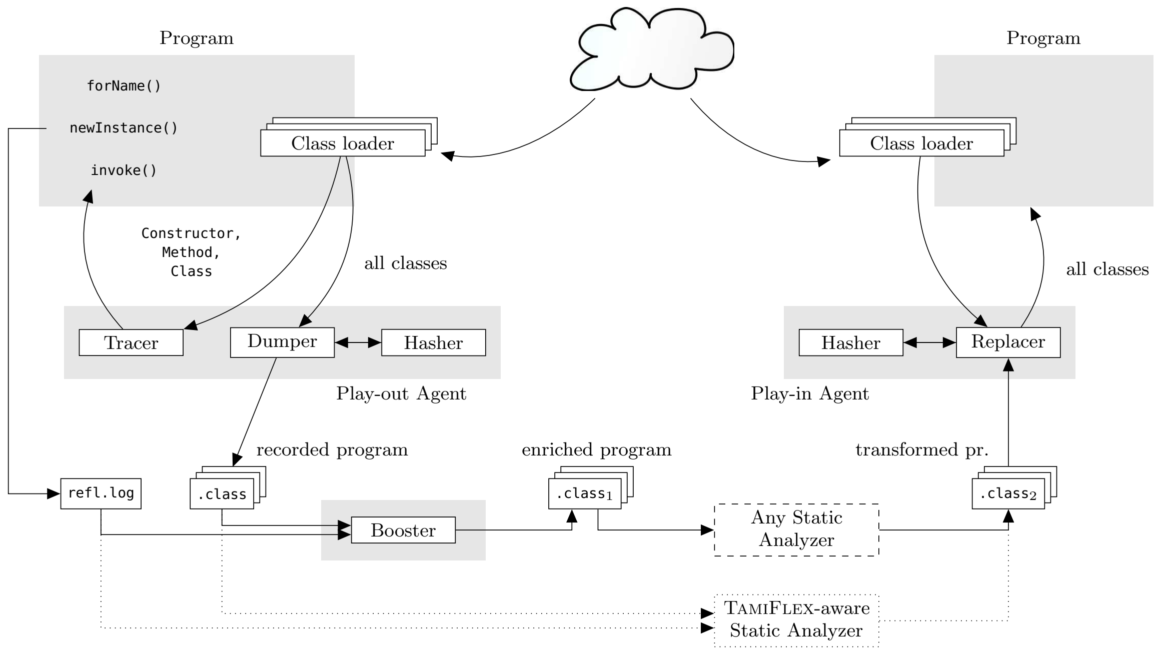


Figure 2: Overview of TamiFlex

*Normalizing randomized class names.* When programs generate classes at runtime, they sometimes assign these classes randomized names. Consider again the example from Figure 1. In the figure, the generated class bears the name `Foo$42`. On the next run, however, the `Generator` may generate the same class under another name, for instance `Foo$23`.

Randomized names complicate our approach. First, we may not easily reach a fixed point when recording program runs with the Play-out Agent. More importantly, however, it would likely happen that the Play-in Agent would search for a class using a name different from the name that the Play-out Agent used to store the same class. This would break the play-out/transform/play-in sequence for such classes.

We therefore decided to extend TamiFlex with a Hasher component that detects classes with a randomized name (using an extensible list of class-name patterns) and then creates a stable SHA1 [27] hash over the classes' contents to assign normalized names to these classes. SHA1 hashes are very unlikely to clash. Both the Play-out Agent and the Play-in Agent use the Hasher component consistently, which allows both agents to re-identify classes across multiple runs. To maintain consistency among classes, we not only change the name of class *files* but also rename class *references* within the dumped bytecode, both in class constants and string constants. On purpose, we designed the hashing algorithm to be maximally sensitive: even small changes to a class will result in a new hash code.

Hashing is less trivial than it may seem: because classes with randomized names can refer to each other, their hash codes can be interdependent. We hence cannot simply hash over every class in isolation but need to hash over entire interdependent groups of classes. Fortunately, all the classes with randomized names that we observed in our experiments only had a tree-like reference structure; there were no cycles. Our agents can therefore compute a dependency tree and compute hash codes starting at leaf nodes. Our Technical Report [8] gives additional information on hashing.

While a program may conversely load different classes with the same name, we did not find this to be a problem in practice: the Play-out Agent issues a warning in such cases but the warning never triggered for our benchmarks.

## 2.3 Booster

The Booster takes as input a repository of class files and a reflection log file, and produces as output an enriched program version by "materializing" reflective method calls into standard Java method calls. Figure 3 shows the main method of the enriched version of our example program. The Hasher has replaced the name `Foo$42` by the normalized name `Foo$H1`. The Booster has added lines 5 and 7 and modified line 6. Line 5 contains the "materialized" call. Because this (previously hidden) call is now present in the bytecode, static-analysis tools can pick up the call when constructing their call graph. Line 7 contains the runtime check that warns the programmer when executing reflective calls that the Booster did not materialize because these calls were not previously recorded. The check only amounts to a hash-set look-up in the Booster-generated class `BoosterRuntime` (bottom of figure). The Booster distinguishes calls by their caller method: a call to a target method may be known and allowed when called from one caller but not from another. The check method therefore uses a static caller id to look up the information for the right caller. To issue a warning, the check-method calls a user-definable warning listener.

Observant readers will have noticed that the materialized call in line 5 is guarded by a conditional. The expression `Opaque.false()` resembles an "opaque" predicate, i.e., a Boolean value that is always false, but which no static analysis can evaluate to "always false." Due to this predicate, the materialized call will never actually execute, although any sound static analysis will safely assume that the call may execute. Because the program still executes the original reflective call in line 6, its behavior is unchanged.

The reader may wonder why the need for opaque predi-

```
public static void main(String[] args) throws Exception {
    Connection c = new Connection();
    Class clazz = Generator.makeClass(); // returns Foo$H1
    Method m = clazz.getMethod("foo", Connection.class);
    if (Opaque.false()) Foo$H1.foo(c); // materialized call
    else m.invoke(null, c); // calls Foo$H1.foo()
    BoosterRuntime.check(0, m); // check if m is known
    c.write("what a risky method call: c is closed!");
}

public class BoosterRuntime {
private static Set knownTargets = ... // {"0-Foo$H1.foo()" }
public static void check(int caller, Method m) {
    if (!knownTargets.contains(caller+"-"+m.signature())) {
        Listener.warnMethodInvoke(m); // issue warning
    }
} ... }
```

Figure 3: Enriched version of the program from Figure 1

cates arises and why we do not execute the materialized call directly. Unfortunately, such direct calls may not work if multiple class loaders are involved. Consider this code:

```
1 Class c = myLoader.loadClass("C");
2 Method m = c.getMethod("m");
3 m.invoke(null, null);
4 C.m(); // does not use <myLoader, C> but C in current scope
```

The call at line 3 calls `C.m()` using reflection, where `C` is a class loaded by the class loader `myLoader`. The call at line 4 attempts to perform the same method call directly. Direct method calls, such as in this line, however, will always use the class loader of the class containing the method call. But if this class loader has no access to `C` then the call `C.m()` will result in an exception. Executing the original reflective call instead of the materialized call prevents this problem.

The Booster is an extension of the Soot [33] bytecode analysis and transformation framework. Nevertheless, the programs that the Booster produces can be processed by a wide range of program-analysis tools, not just by Soot.

### 2.4 Play-in Agent

The Play-in Agent uses another class-file transformer to re-insert offline-transformed classes into a running program, irrespective of the program's class loaders. At runtime, the agent uses the Replacer to replace all classes as they are loaded into the virtual machine with the contents of the respective `.class` files from the repository on disk. Consider Figure 2: For every class, the original class loader will first load the original class from "the cloud," and then pass the byte array of this class to the Replacer. The Replacer will then deliberately ignore this array, however, and return the contents of the respective `.class` file instead. The Replacer consults the Hasher to compute normalized names for classes with randomized names. When loading a `.class` file with a normalized name, the Replacer renames the normalized class again, this time to the name that the class loader of the currently running program originally requested. In our example, the name `Foo$H1` may, for example, be replaced by `Foo$23`. This renaming adopts the offline-transformed class to the current class-loading context.

## 3. EXPERIMENTS

In this section, we present experimental evidence showing that programmers can effectively use TamiFlex to conduct static whole-program analysis on a wide range of programs that use reflection, custom class loaders, and runtime-generated classes. But our experiments also highlight some limitations: for highly interactive or extensible applications it may be hard to obtain a reasonably complete log, as different features often trigger different reflective calls.

With our experiments, we aim to answer the following research questions.

**RQ1** Correctness: The transformations that our Booster applies are non-trivial. Therefore, when given an enriched program as input, will existing static-analysis tools indeed produce call graphs that are sound with respect to all runs that the Play-out Agent recorded?

**RQ2** Effectiveness: How do input size and code coverage affect the quality of the logs?

**RQ3** Efficiency: Does TamiFlex induce a runtime overhead low enough for practical use?

The next three sections will answer these questions. Section 3.4 discusses threats to the validity of our experiments.

### 3.1 RQ1: Relative soundness of call graphs

The main reason for using a TamiFlex-generated log in a static analysis is that the analysis can use the log to obtain a complete picture of the program's calling structure, i.e., a complete call graph. Because TamiFlex enriches the program based on data collected from recorded runs, the information encoded in the enriched program can only be as complete as the coverage of reflective calls on these runs. In the following, we will refer to call graphs that are complete with respect to the recorded runs as call graphs that are "representative" (for these runs).

The Booster aids the construction of representative call graphs by materializing reflective calls in the form of normal Java method calls. This materialization is non-trivial. Because call-graph construction is inter-dependent with the computation of points-to sets [23], one not only has to add calls to the right methods but also needs to supply these methods with their arguments, un-boxing these arguments from the arrays that are normally passed to reflective calls, and to correctly capture the return value. We hence conducted the following experiments to rule out mistakes in the Booster's implementation and omissions in the instrumentation performed by the Play-out Agent.

We gave TamiFlex-enriched versions of all programs from version 9.12-bach of the DaCapo [5] benchmark suite as input to the Soot program analysis framework [33]. We configured Soot to use Spark [23] to construct a static call graph for each enriched program. For such a call graph to be representative in the above sense, the graph must contain an edge `m` $\rightarrow$ `n` for every call from a method `m` to a method `n` that occurred on a run that the programmer recorded with TamiFlex when producing the reflection log file that the programmer then provided as input to the Booster.

To test whether the Booster's transformations are correct and sufficient, we compared the static call graphs that we obtain through our combination of TamiFlex and Spark with dynamic call graphs for the same benchmark configurations. If the static graphs contain the dynamic ones, this confirms that the static call graphs are representative. But obtaining dynamic call graphs is a non-trivial task in itself. For the purpose of this evaluation, we wrote a native

JVMTI [21] agent that produces highly accurate dynamic call graphs. The agent is able to record even method calls in the very early stages of the VM's start-up sequence, long before `main` is called; in particular, the agent can record calls from native code back into Java bytecode. We believe that the call graphs recorded that way are as complete as possible without modifications to the underlying virtual machine.

After recording dynamic call graphs, we used Lhoták's call-graph differencing tool ProBe [22] to compare these graphs to the static call graphs computed with Spark and TamiFlex. Without TamiFlex, the call graphs that Spark produces for the DaCapo benchmarks are hopelessly unsound: DaCapo uses a call to `Method.invoke` to bootstrap every benchmark, and without applying the Booster, Spark would miss this call and its entire transitive closure, i.e., the entire benchmark. In combination with TamiFlex, however, our results confirm that Spark does indeed produce call graphs that are representative for all program runs that the DaCapo benchmarks can produce on their pre-defined inputs. The only missing edges were native calls to shutdown hooks and finalizers. Static analyses traditionally ignore such edges. However, one could treat those edges soundly by extending the Play-out Agent and Booster appropriately.

## 3.2 RQ2: Effect of code coverage

We next sought to determine how much the quality of a reflection log depends on the size of the input yielding the program run that produces this log. The current DaCapo release "bach" offers up to four input sizes for each benchmark: small, default, large, and huge. Because "huge" only exists for a small subset of benchmarks (4 out of 14), we restrict ourselves to the other three input sizes.

### *3.2.1 Input size and coverage of reflective call sites*

A simple metric for the quality of a reflection log is the number of reflective call *sites* that it covers. The more call sites it covers, the more call sites Spark can model using the information in the log. In Figure 4 we show the number of reflective call sites covered by the log for every benchmark configuration. We were pleasantly surprised to see that the number of covered reflective call sites is *not* heavily correlated with the input size of the respective benchmark run. For nine benchmarks, the number of covered sites does not increase at all for larger inputs, and for all others except for jython and eclipse the number increased only slightly.

Some benchmarks, like avrora, appear to cover more reflective call sites on their small and/or default configuration than on the large input. To explain this effect, we decided to further measure the correlation between input size and code coverage. We used ProBe to create intersections of all possible combinations of the dynamic call graphs that we obtained by running DaCapo with each of the input sizes and with our call-graph generating JVMTI agent enabled.

*Input size and code coverage.* Figure 5 shows the result of this process for avrora, batik, and eclipse as a set of three Venn diagrams. (Our Technical Report [8] contains diagrams for all benchmarks.) In the figure, "no_bm" denotes the run where we start the DaCapo suite without stating the required command-line parameter that selects the benchmark to run. We found this to be an interesting "input" too, because it can be regarded as an erroneous benchmark run that diverges from the benchmark's normal execution.

| Benchmark | Input size | | |
|---|---|---|---|
| | small | default | large |
| avrora | 18 | 18 | 12 |
| batik | 41 | 44 | 44 |
| eclipse | 212 | 351 | 351 |
| fop | 142 | 130 | n/a |
| h2 | 31 | 31 | 31 |
| jython | 41 | 50 | 50 |
| luindex | 66 | 41 | n/a |
| lusearch | 40 | 42 | 42 |
| pmd | 32 | 32 | 32 |
| sunflow | 30 | 30 | 30 |
| tomcat | 165 | 165 | 165 |
| tradebeans | 624 | 620 | 618 |
| tradesoap | 638 | 634 | 640 |
| xalan | 54 | 54 | 54 |

Figure 4: Number of reflective call sites in log

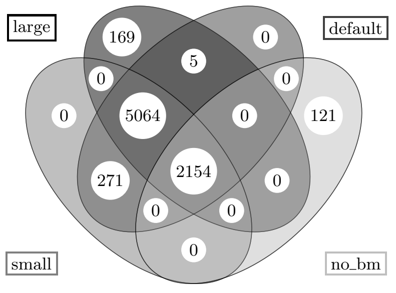
The Venn diagrams clearly explain the reduced coverage of reflective call sites in some of the runs, such as avrora/large: considering Figure 5a, the reduced coverage is not surprising, as the small and default inputs cover 271 methods that the large input misses and the large input covers only 169 methods that the other input sizes miss.

### *3.2.2 Input size and number of reflective call edges*

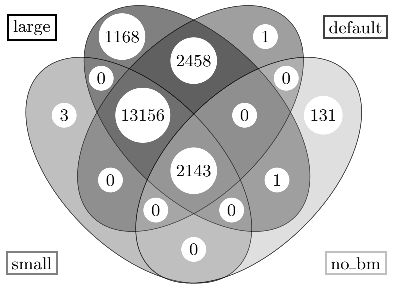
Even when a larger input does not cover more reflective call sites, the larger input could yield a larger variety of reflective calls (to more targets) at these call sites, producing more reflective "call edges" in the associated call graph. Figure 6 plots the number of call edges for each benchmark and input. We can see that the increase in the number of call edges is moderate as well: In all cases except eclipse and jython the larger inputs do not yield more than 20% additional call edges, or in other words, in all but two cases do the small configurations already yield a call-edge coverage of more than $1/1.2 \approx 83\%$ compared to the configuration that yields most coverage. As we show through annotations in the figure, the many additional calls in eclipse and jython arise from the fact that larger inputs exercised more different program parts than smaller inputs. For instance, eclipse uses the Java Development Tools (JDT) in "default" and "large" but not in "small."

For the scenario of benchmarking with DaCapo, this is unproblematic. Without much effort, one can collect different reflection logs for the up to four different input sizes. Input dependencies could, however, be a problem when more varied inputs occur, such as in interactive, user-driven programs. We therefore additionally experimented with nine well-known open-source Java applications to determine how input-dependent their logs are. How practical is our approach in combination with such applications?

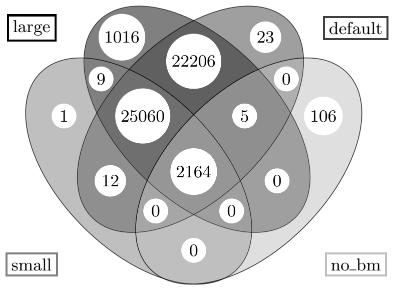
We summarize our findings in Figure 7, similar in style to Figure 6 but with a different baseline: in this case, we obtained the baseline by a trivial program execution, i.e., by starting and immediately closing the program. Then we executed up to 40 different program runs, each time trying out new features, e.g., new user-interface elements. Our project website contains descriptions of all user inputs. The figure clearly shows that if the input is more varied than in the case of the DaCapo benchmarks, then more iterations may be required until a plateau with a reasonably



(a) avrora



(b) batik



(c) eclipse

Figure 5: Venn diagrams showing the number of reachable methods shared by the dynamic call graphs at different input sizes (small / default / large) and with a run of the DaCapo suite without selecting a benchmark (no_bm)

complete log is reached; only PDFsam and SweetHome3D appear largely input-independent. Like in DaCapo, large jumps in the number of recorded calls occurred when we used certain reflection-intensive features for the first time, e.g., the Java or AspectJ Development Tools (JDT/AJDT) in Eclipse or a Swing-based application in Jython or JRuby.

We conclude that, in general, the quality of reflection logs does depend on the quality of the input to the recorded runs, in particular on the coverage that these runs produce. However, it appears often sufficient to execute program runs that just "touch" the program's reflection-using features; these features, such as Swing, usually incur many reflective calls when they are loaded, but not many different calls during subsequent use. This suggests that few recorded program runs may already yield static call graphs that show much better code coverage than call graphs that would have been produced without TamiFlex. For certain applications, like benchmarking with DaCapo, our approach is even complete: because there only exist a few possible inputs, users can produce logs for all inputs that exist, and based on each log, static-analysis tools can produce a call graph that is representative for the respective run.

## 3.3 RQ3: Performance overhead of TamiFlex

Users may need to apply the Play-out Agent across multiple runs. In addition, researchers may want to use the Play-in Agent to measure the performance impact of static optimizations applied to Booster-enriched versions of benchmark suites like DaCapo. It is therefore important to consider the runtime overhead that both agents and the Booster's transformations incur. To quantify this runtime overhead, we used the DaCapo benchmark suite for what it was designed for: runtime-performance evaluation. We used a 2.33 GHz Intel E6500 Core 2 Duo processor running Ubuntu Linux 9.10 (kernel 2.6.31) in single-user mode. The entire main memory of 2GB was available as heap to the Sun HotSpot Server VM (build 14.2-b01), running in mixed mode.

We recorded the runtime of ten invocations each for all benchmarks under three configurations: (1) without any agents (acting as a baseline), (2) with the Play-out Agent enabled, and (3) with the Play-in Agent enabled, running the enriched program version. During each invocation, the benchmark performed two iterations of its default workload and we report the runtime of both iterations. Technical problems (see [8]) prevented us from determining runtime overheads for tomcat. Figure 8 shows both the arithmetic mean and standard deviation of the recorded runtimes. As we can see, in all three configurations the first iteration takes noticeably longer than the second one. First, the VM's just-in-time compiler successively optimizes the generated code; thus, not only has more code already been optimized during the second iteration, the optimizing compiler will also spend less time compiling new code. There is another reason, however, which has more impact on the workings of TamiFlex: the VM loads most (if not all) classes during the first iteration; not only has the VM less work to do during the second iteration, but so do the Play-out Agent and Play-in Agent because they are triggered at class load time. As we found, the Booster's transformations have no measurable effect. As opaque predicate we used a simple static boolean field set to `false`. This allows the virtual machine to perform efficient evaluation and prediction; materialized calls cause zero overhead. The error-checking code inserted by the Booster amounts to a single inclusion test in a hash set.

Figure 8 shows that TamiFlex incurs little overhead during either iteration. The one notable exception is the tradesoap benchmark, for which the **Play-out Agent** causes a 85.5% overhead during the first iteration and a 160.2% overhead during the second iteration. This is due to the large number of reflective calls that tradesoap makes—more than 10 times as many as made by tradebeans, the close cousin of tradesoap and second on the list. (We report the total number of reflective calls per benchmark in our Technical Report [8].) The main performance bottleneck in the Play-out Agent is the repeated creation of stack traces that it creates to determine the method in which a reflective call is issued. When using a "dummy" stack trace instead of constructing stack traces anew upon every reflective call, the Play-out Agent's overhead on tradesoap drops to a mere 10.0%. Optimized implementations, e.g. using probabilistic calling context [9], could therefore implement the Play-out

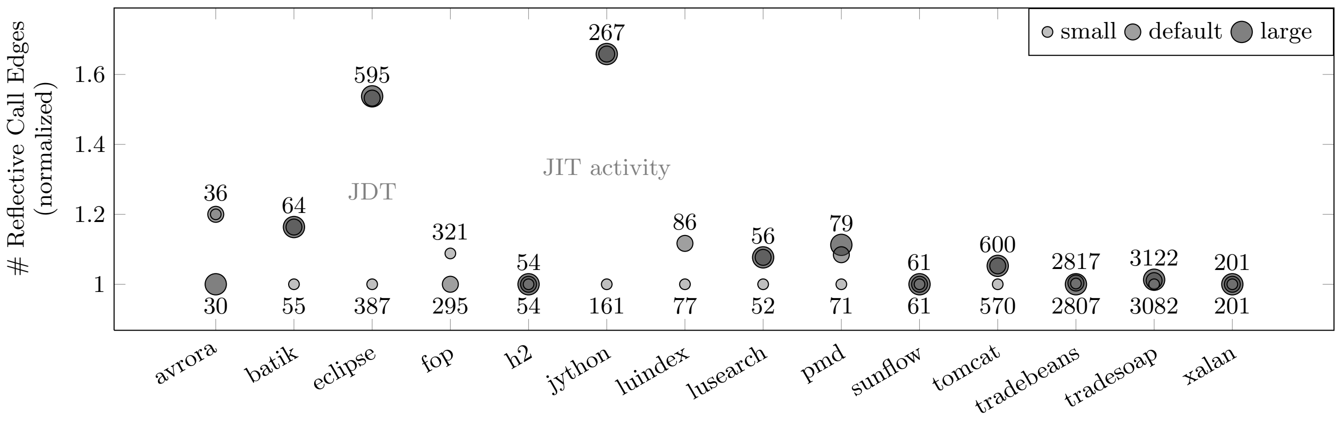


Figure 6: Number of reflective call edges discovered when running the DaCapo benchmarks at different input sizes (normalized w.r.t. least number of call edges)

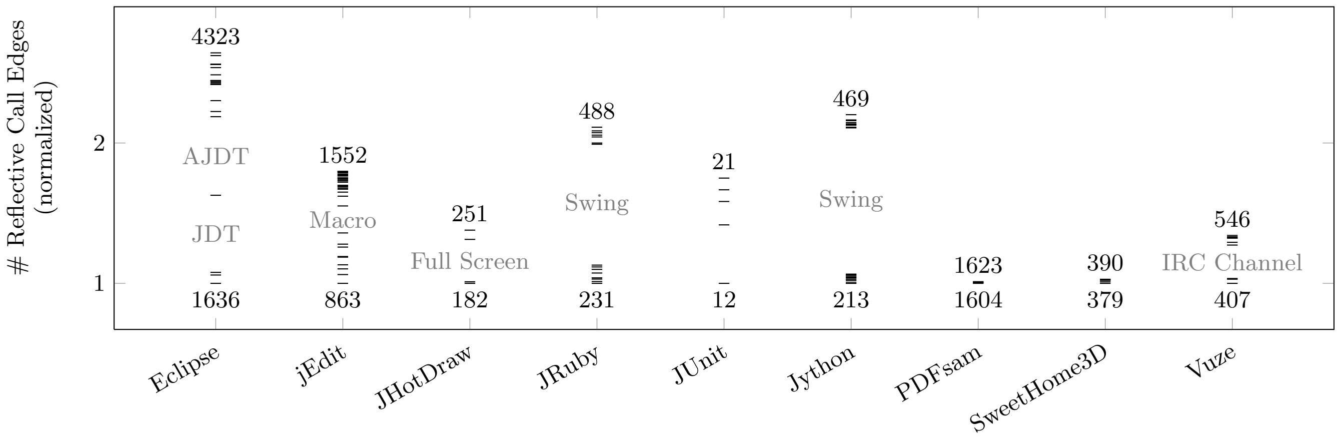


Figure 7: Number of reflective call edges discovered during manual usage (normalized w.r.t. least number of call edges); large jumps are annotated with the program feature whose first-time usage introduced new edges

Agent with low overhead, however, would likely require some support by the underlying virtual machine.

In contrast to the Play-out Agent, the overhead incurred by the **Play-in Agent** is mainly limited to the first iteration, during which it replaces newly-loaded classes. Once all classes have been loaded, the agent does not slow down the running application any more. This is important, as it means that researchers can indeed use the Play-in Agent to evaluate statically optimized versions of DaCapo.

## 3.4 Threats to validity

The internal validity of our experiments is high. Showing that our static call graphs entirely contain the dynamic call graph for the same run allows us to conclude that the static call graphs are sound with respect to those runs. We compared call graphs using ProBe, a state-of-the-art tool developed and successfully tested by others [22].

The external validity of our experiments may be threatened by our choice of programs, and the program inputs we selected. However, we took care to select a wide variety of programs, 23 different programs from different sources and domains. Therefore the chances are high that the same approach applied to different programs would yield results allowing for the same conclusions that we make in this paper. For reproducibility, we used for DaCapo the inputs provided with the suite, and documented our manual inputs to the other nine programs on the project website.

## 4. RELATED WORK

TamiFlex addresses three major problems: (1) custom class loaders, (2) reflection, and (3) re-inserting transformed classes. While related work rarely makes a concerted effort to address all three problems, various solutions have been proposed to (1) and (2). Our ideas of a Booster and a Play-in Agent, however, appear entirely novel.

In an effort to reduce application size, Tip et al. [31,32] developed Jax, an application extractor, which removes from a program methods, fields, and even entire classes that are not used in the user's application scenario. To make call-graph construction sound with respect to this application scenario, the authors proposed MEL, a modular extraction language to specify assumptions about the environment. To assist the users with this task, Jax includes a tool that identifies reflective calls, similar to TamiFlex's Play-out Agent.

In a recent article [4], the developers of Coverity [13] explain some of the difficulties they encountered when making their research tool for bug finding ready for the market, an important one being that "You can't check code you don't see." The authors solved this problem (in a C-based setting) by intercepting system calls during the program's build pro-

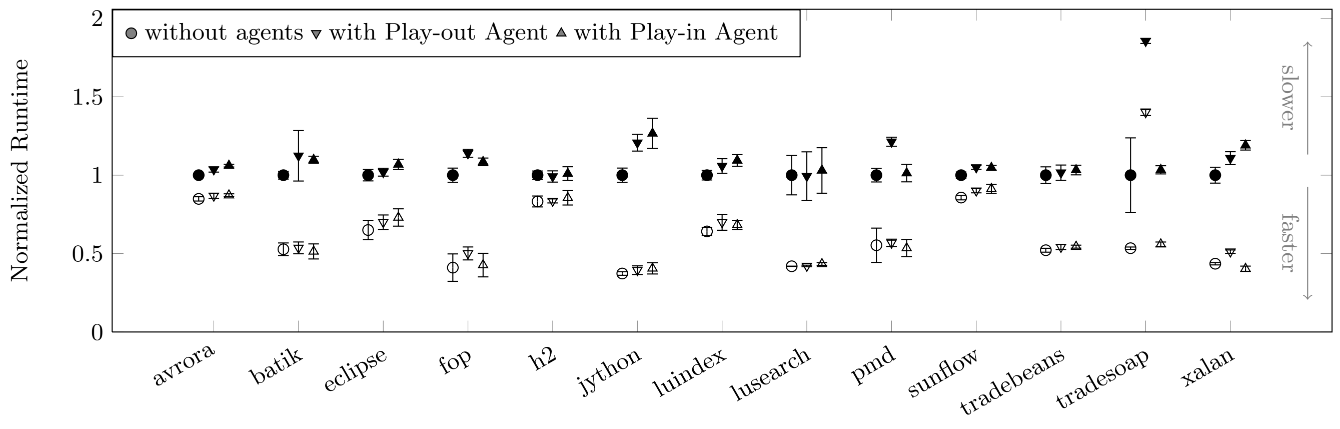


Figure 8: Runtime of the first (■) and second (□) iteration of 13 different DaCapo benchmarks using the default input size (arithmetic mean ± standard deviation, normalized w.r.t. the first iteration without agents)

cess. In a Java-based setting, TamiFlex's Play-out Agent solves the same problem by dumping all loaded classes.

To enable static analysis in the presence of reflection, Livshits, Whaley and Lam [24] present a static-analysis approach that attempts to infer additional information about reflective call sites directly from program code. The analysis attempts to use information stored in string constants to resolve reflective calls statically. For call sites for which this information is insufficient, their approach allows programmers to provide additional information through manual hints. Christensen et al. present a general-purpose string analysis [12]. As we observed, many reflective calls are resolved not using string constants but using information from the environment or configuration files. TamiFlex covers all these cases. Further, TamiFlex does not require researchers to make their static analyses reflection-aware.

Similar to Livshits et al., Braux and Noyé [11] propose a static approach to handling reflection at compile-time. The author's solution, aimed at increasing runtime performance, propagates type information through the program's abstract syntax tree. In some cases this information is sufficient to substitute dynamically loaded classes by concrete types and calls to the reflection API by concrete method calls.

Hirzel et al. [19,20] present an online version of Andersen's points-to analysis [2] that executes alongside the program, as an extension to the Jikes RVM [1], an open-source Java Research Virtual Machine. As an online algorithm, the approach can exploit runtime information; for instance, it can observe reflective calls as they execute. The authors do not present how programmers can effectively use the points-to sets and the call graph that their approach computes at runtime. While for any given point in time both the points-to sets and the call graph correctly model the part of the program that has already executed, they cannot soundly model program parts that have not yet executed. Most existing analyses that use call graphs and points-to sets operate under a closed-world assumption, i.e., they assume that call graphs and points-to sets soundly model all possible executions. It appears non-trivial to adopt such algorithms such that they could use the incomplete, online-generated points-to sets and call graphs instead. TamiFlex aims at supporting programmers in obtaining call graphs that are at least almost complete for the entire program, by collecting reflection information and class files across multiple program runs, e.g., using test cases. That way, assuming sufficient test coverage, one can obtain a call graph that is mostly complete for all possible executions.

Dufour [14] uses dynamically-recorded calling structure data as input to a static method-escape analysis. In the process, termed blended analysis, a runtime component feeds information to a static component. The purpose of this approach is a detailed static analysis of parts of a large program that have been identified as a performance bottleneck. A dynamic component records information about reflective calls and about the classes that are loaded at runtime, and then feeds this information, along with information about the performance bottlenecks, to a static-analysis component. While being similar in intent to our Play-out Agent, using TamiFlex has the advantage that the static analysis can remain unmodified, and that one can use the Play-in Agent to re-insert offline-transformed classes.

With $\mathcal{P}$Ruby [17], Furr et al. propose a static-type inference system for the Ruby programming language. Using runtime profiles $\mathcal{P}$Ruby is able to cope with most uses of `send`, `require`, and `eval`, three language features which are akin to reflection, dynamic class loading, and on-the-fly code generation in the Java language. $\mathcal{P}$Ruby can furthermore deal with one feature with does not have a Java analog: `method_missing` methods. Like TamiFlex, $\mathcal{P}$Ruby adds instrumentation to catch cases not covered by the previously recorded profiles. Unlike TamiFlex, $\mathcal{P}$Ruby requires two separate profiling runs: a first run has to gathers all source files, which are then instrumented offline, so that a second run can then records the runtime profile.

## 5. CONCLUSION

We have presented TamiFlex. This flexible tool chain allows users to significantly enhance their static analyses' coverage and soundness when applied to Java programs that invoke methods and load classes using reflection, or even generate classes at runtime. Moreover, TamiFlex allows researchers to transform, e.g., optimize or instrument, these classes and re-insert the offline-transformed classes into the original application. TamiFlex is compatible with most static analyses for Java.

We have shown the feasibility of our approach by applying it to version 9.12-bach of the DaCapo benchmark suite and nine other interactive open-source Java programs, a realistic cross-section of the current state of the art in Java programming. The results show that researchers can effectively use TamiFlex to create call graphs for all DaCapo benchmarks that are sound with respect to all runs that these benchmarks can produce, despite their use of reflection, custom class loaders, and dynamic class generation. For the interactive Java programs, the approach, while incomplete, can significantly improve static-analysis quality.

*Acknowledgements.* We thank the DaCapo developers for all the work involved with developing and continuously maintaining such an extensive benchmark suite. We are especially grateful to Steve Blackburn for answering many questions even at—for him—inconvenient hours in (Australian) Eastern Time. We further wish to thank Ondřej Lhoták for providing and supporting ProBe; this call-graph viewing and differencing tool proved enormously useful. Much gratitude also goes to Walter Binder, Nurudeen Lameed and the reviewers of OOPSLA 2010 and ICSE 2011, whose comments helped us to improve much over initial versions of this paper. This work was supported by CASED (www.cased.de). Hela Oueslati received funding from the DAAD.

## 6. REFERENCES

[1] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, S. F. Hummel, D. Lieber, V. Litvinov, M. F. Mergen, T. Ngo, J. R. Russell, V. Sarkar, M. J. Serrano, J. C. Shepherd, S. E. Smith, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994. DIKU report 94/19.

[3] Shay Artzi, Adam Kiezun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE'07*, pages 104–113. ACM, 2007.

[4] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *CACM*, 53(2):66–75, 2010.

[5] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *OOPSLA'06*, pages 169–190. ACM, 2006.

[6] Eric Bodden. Efficient Hybrid Typestate Analysis by Determining Continuation-Equivalent States. In *ICSE'10*, pages 5–14. ACM, 2010.

[7] Eric Bodden, Patrick Lam, and Laurie Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *FSE'08*, pages 36–47, 2008.

[8] Eric Bodden, Andreas Sewe, Jan Sinschek, and Mira Mezini. Taming Reflection (Extended version). Technical Report TUD-CS-2010-0066, CASED, March 2010. http://cased.de/.

[9] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA'07*, pages 97–112. ACM, 2007.

[10] Guillaume Brat and Willem Visser. Combining static analysis and model checking for software analysis. In *ASE'01*, page 262. IEEE, 2001.

[11] Mathias Braux and Jacques Noyé. Towards partially evaluating reflection in java. In *PEPM'99*, pages 2–11. ACM, 1999.

[12] Aske Christensen, Anders Møller, and Michael Schwartzbach. Precise analysis of string expressions. In *SAS'03*, volume 2694 of *LNCS*, pages 1–18. Springer, 2003.

[13] Coverity static-analysis tool. http://coverity.com/.

[14] Bruno Dufour, Barbara G. Ryder, and Gary Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA'07*, pages 118–128. ACM, 2007.

[15] Matthew B. Dwyer and Rahul Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *ASE'07*, pages 124–133, 2007.

[16] Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanual Geay. Effective typestate verification in the presence of aliasing. In *ISSTA'06*, pages 133–144. ACM, 2006.

[17] Michael Furr, Jong-hoon (David) An, and Jeffrey S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA'09*, pages 283–300. ACM, 2009.

[18] Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):227–242, 1992.

[19] Martin Hirzel, Daniel Von Dincklage, Amer Diwan, and Michael Hind. Fast online pointer analysis. *TOPLAS*, 29(2):11, 2007.

[20] Martin Hirzel, Amer Diwan, Michael Hind, Martin Hirzel, Amer Diwan, and Michael Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP'04*, pages 96–122. Springer, 2004.

[21] Java Virtual Machine Tool Interface (JVM TI). Version 6. http://download.oracle.com/javase/6/docs/technotes/guides/jvmti/index.html.

[22] Ondřej Lhoták. Comparing call graphs. In *PASTE'07*, pages 37–42. ACM, 2007.

[23] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using Spark. In *CC'03*, volume 2622 of *LNCS*, pages 153–169. Springer, 2003.

[24] Benjamin Livshits, John Whaley, and Monica S. Lam. Reflection analysis for java. In Kwangkeun Yi, editor, *APLAS'05*, volume 3780 of *LNCS*, pages 139–160. Springer, 2005.

[25] Gail C. Murphy, David Notkin, William G. Griswold, and Erica S. Lan. An empirical study of static call graph extractors. *TOSEM*, 7(2):158–191, 1998.

[26] Nomair A. Naeem and Ondřej Lhoták. Extending typestate analysis to multiple interacting objects. Technical report, University of Waterloo, 04 2008. CS-2008-04.

[27] National Institute of Standards and Technology, Information Technology Laboratory. *Secure Hash Signature Standard (SHS)*, 2008. FIPS PUB 180-3.

[28] Venkatesh Ranganath and John Hatcliff. Slicing concurrent Java programs using Indus and Kaveri. *STTT*, 9:489–504, 2007.

[29] Shmuel Sagiv, Thomas W. Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.

[30] R. E. Strom and S. Yemini. Typestate: A programming language concept for enhancing software reliability. *TSE*, 12(1):157–171, 1986.

[31] Peter F. Sweeney and Frank Tip. Extracting library-based object-oriented applications. In *FSE'00*, pages 98–107. ACM, 2000.

[32] Frank Tip, Peter F. Sweeney, Chris Laffra, Aldo Eisma, and David Streeter. Practical extraction techniques for java. *TOPLAS*, 24(6):625–666, 2002.

[33] Raja Vallée-Rai, Phong Co, Etienne Gagnon, Laurie Hendren, Patrick Lam, and Vijay Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON'99*, page 13. IBM, 1999.