Technical Report

Nr. TUD-CS-2010-0066
March 25th, 2010

CASED

Taming Reflection (Extended Version)

Static Analysis in the Presence of Reflection and Custom Class Loaders

Authors
Eric Bodden, Andreas Sewe, Jan Sinschek and Mira Mezini
all are with CASED | TU Darmstadt, Darmstadt, Germany

Taming Reflection (Extended version)

Static Analysis in the Presence of Reflection and Custom Class Loaders

Eric Bodden Andreas Sewe Jan Sinschek 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 the analysis get hold of a class that the program may load from a remote location or even generate on the fly? And if its results are used to transform classes offline, how can it ensure that the transformed classes are re-inserted into a running program that uses custom class loaders?

In this paper we present TAMIFLEX, a tool set for taming reflection. TAMIFLEX consists of two novel instrumentation agents. The Play-out Agent logs reflective calls into a log file and gathers all loaded classes, including generated ones. The Play-in Agent re-inserts offline-transformed classes into a running program. To show how researchers can use TAMIFLEX, we modified the Soot framework for static analysis, and in particular it's points-to-analysis component Spark, so that it uses the log file to construct a sound call graph and points-to sets even for programs that use reflection, custom class loaders, and dynamic class generation.

We prove our approach feasible by applying TAMIFLEX to the 9.12-bach release of the DaCapo benchmark suite, which uses all the aforementioned dynamic features. For the first time, TAMIFLEX enables researchers to conduct static whole-program analyses on this version of DaCapo. Our experiments show that our combination of Soot and TAMIFLEX produces sound call graphs, that TAMIFLEX usually produces less than 10% runtime overhead and that the reflection log files do not depend much on program input.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

General Terms Algorithms, 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 a variable's possible runtime types [17], may- and must-aliasing information [8] and call graphs [29] to analyses that determine the shape of custom data structures [21, 34], track an object's typestate [9, 16, 18, 32], or try to enforce restrictions on the program's information flow [30]. Many of these analyses are coupled with program transformations, for instance static optimizations. As one example, in previous work [9, 16, 32] we and others have used static typestate-analysis information to restrict the updates to the internal state of runtime monitors for typestate properties [37].

Virtually all of these analyses are whole-program analyses, i.e., the analyses must analyze 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 a "whole program" yields many challenges when analyzing Java programs that use reflection, or load classes using custom class loaders:

1. Industrial Java applications frequently use custom class loaders that may load classes from obscure locations, or even generate classes on the fly. A static analysis needs to get hold of these classes even if it has no knowledge of or access to these custom class loaders.
2. The same programs also frequently use reflection to invoke methods or instantiate objects of types that programmers cannot fully determine at compile time. To construct a complete call graph, a static analysis needs to be aware of these calls.
3. Even if a static analysis is aware of reflective calls and has access to all classes that are loaded at runtime, re-

searchers need to modify the analysis to handle the reflective calls and all the program's classes correctly.

4. The fourth problem concerns program transformations. If programmers use static-analysis results to transform (e.g. optimize or instrument) an application, then one needs to transparently re-insert the transformed classes into the application's class-loading process, even if this process relies on custom class loaders.

In this paper we present TAMIFLEX, a tool suite for "taming reflection", that can solve all of these problems. TAMIFLEX consists of two novel instrumentation agents, a Play-out Agent and a Play-in Agent. The Play-out Agent agent logs reflective calls into a reflection log file, and gathers all classes that the program loads, even when the program loads these classes through custom class loaders or generates them on the fly. This effectively solves Problems 1 and 2.

If users of TAMIFLEX simply wish to analyze a program statically, without transforming the program, they only need to use the Play-out Agent, not the Play-in Agent. Users can simply feed the class files and reflection log that the Play-out Agent generates to their favorite 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 this case, 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, Problem 4 from above, by re-inserting offline-transformed classes into a running program, i.e., online. The agent can even replace classes that an application generates at runtime.

To show how one can solve Problem 3 using TAMIFLEX, we modified the static-analysis framework Soot [39], and in particular its points-to-analysis component Spark [27], so that it uses the reflection log file and the class files produced by the Play-out Agent to construct a sound call graph and points-to sets even for programs that use reflection, custom class loaders and load runtime-generated classes. Supporting reflection required modifications to the construction algorithms for call graphs and points-to graphs. We did not, however, need to modify Soot to support custom class loaders: TAMIFLEX's Play-out Agent provides to Soot all classes that the program loads at runtime in the form of simple class files on disk.

We prove the feasibility of our approach by applying TAMIFLEX to the 9.12-bach release [13] of the DaCapo benchmark suite [7], which uses all the dynamic features we mentioned. For the first time, our tool suite enables researchers to conduct static whole-program analysis on this version of DaCapo. We further give experimental evidence that the call graphs that Spark produces in combination with TAMIFLEX are sound: For evaluation purposes, we implemented a JVMTI agent [25] that produces highly accurate dynamic call graphs without modifications to the underlying virtual machine. In particular, the dynamic call graphs thus produced contain calls from native code back into Java bytecode. We then used Lhoták's call-graph differencing tool PROBE [26] to compare the dynamic call graphs to the static call graphs that we compute with Soot and TAMIFLEX. The results show that our call graphs are complete, even for runtime-generated code. We further show that both our agents induce a runtime overhead of usually below 10%. The Play-in Agent in particular induces no overhead after all classes have been loaded. This shows that researchers can effectively use TAMIFLEX to run statically optimized versions of DaCapo: when set to run a benchmark with multiple iterations, the Play-in Agent will only cause a runtime overhead during the first iteration. Our initial results further suggest that the reflection log files, and thus also the static-analysis information, are largely input independent.

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 that program uses custom class loaders.
- An updated version of Soot and Spark that properly takes reflective calls into account.
- In combination, the first automated solution that allows researchers to conduct static whole-program analysis and transformation on the 9.12-bach release of DaCapo.
- A set of experiments that prove that our tool chain is efficient, yields sound call graphs for all DaCapo benchmarks, and is largely input independent.

We organized the remainder of this paper as follows: In Section 2 we briefly discuss the DaCapo benchmark suite and how its use of reflection and custom class loaders influenced our design decisions. In Section 3 we give an overview of the architecture of TAMIFLEX. Section 4 explains the Play-out Agent and the Play-in Agent in detail, while Section 5 discusses our modifications to Soot and Spark. We report on our experiments in Section 6, discuss related work in Section 7, and conclude in Section 8.

2. DaCapo Benchmark Suite

The premier goal of the DaCapo benchmark suite [7] is to offer "a set of open source, real world [Java] applications with non-trivial memory loads" to "the programming language, memory management and computer architecture communities" [13]. The designers of DaCapo originally implemented the suite primarily with runtime techniques like just-in-time compilation and garbage collection in mind. Nevertheless, in recent years DaCapo also appears to have become one of the most widely used benchmark suites in the static-analysis community [5, 22, 31, 36, 42]. We expect that the second incarnation of the benchmark suite (version 9.12-bach), released after three years of development, will achieve a similar status, as it provides a broad, up-to-date selection of Java programs. Table 1 gives an overview of the benchmarks in the "bach" release.

Analyzing realistic Java programs, such as the ones contained in the DaCapo suite, poses many technical challenges. In the following, we discuss the major challenges that motivated TAMIFLEX and its design.

References through dormant code The first major release of DaCapo, release 2006-10, consists of a single JAR file containing the combined classes of 11 different benchmarks. Unfortunately, when trying to analyze DaCapo statically, we found this JAR file to be incomplete: classes within the JAR file statically reference other classes that the JAR file does not contain and which are also not part of the Java Runtime Library. This problem went initially unnoticed because DaCapo's particular benchmark runs never cause the Java runtime to load these "missing" classes. A static analysis, however, cannot easily tell apart live code that a particular run executes from dormant code that the run does not execute. After all, which pieces of code are live likely depends on input data. A static-analysis tool that takes all possible inputs into account will therefore usually require the missing classes to be present or will cease working. In 2006, for DaCapo's 2006-10 release, the first author of this paper spent more than two weeks modifying DaCapo's build scripts so that they would include all missing classes. As we recently discovered, these modifications broke with the transition to the new DaCapo release. The prospect of spending another two weeks modifying build scripts greatly motivated us to search for a more maintainable, long-term solution. We present one such solution in Section 5, namely modifications to Soot (a widely used static-analysis tool for Java) that allow Soot to properly handle "missing classes."

Reflective method calls The second problem that we faced was the one of reflective method calls. In DaCapo, one uses a command-line parameter to the suite's main class, `dacapo.Harness`, to specify which benchmark to run. This harness class then executes the benchmark itself: it extracts the name of the benchmark's driver class from a configuration file, uses reflection to instantiate the class, and invokes a particular method on the resulting class object. The designers of DaCapo probably opted to use reflection for extensibility: to add a benchmark, one would only need to add the benchmark's classes and a particular configuration file, but one would not have to re-compile `dacapo.Harness` itself. But reflection poses a major obstacle to any static analysis: How should the analysis be aware of the particular benchmark name that the user chooses to provide, and how should the analysis be able to infer the name of a main class from a configuration file in a format that it is not aware of? Back in 2006, we chose to provide benchmark-specific driver classes that call the same methods directly that `dacapo.Harness` would call reflectively. While this is not a general solution, it seemed sufficient at the time. However, later on we found out that not only the harness uses reflection but also the benchmarks themselves. This caused the call graphs produced by our static analyses to be largely unsound. One important feature of TAMIFLEX is therefore its ability to produce a reflection log file. Static analyses can then use this log file to discover which reflective calls can occur at runtime. In Section 5 we will explain modifications to Soot that allow the static-analysis tool to compute sound call graphs using the log files that TAMIFLEX produces.

In December 2009, the DaCapo suite saw its second major release: 9.12-bach. While this release retains the challenges for static analysis posed by its predecessor, we discovered that this new DaCapo release introduced another challenge for static program analyses and especially for program transformations: custom class loaders.

Custom class loaders While DaCapo's earlier release was packaged as a single JAR file, the new release is packaged as a JAR file containing other JAR files, roughly one JAR file per benchmark[1]. When starting the suite, the `Harness` extracts the JAR files for the chosen benchmark from the main JAR file and then loads the benchmark's classes using a custom class loader. This design is not accidental: different benchmarks in DaCapo sometimes use the same libraries, but in different versions. Providing these different versions in different JAR files avoids name-space collisions. However, this design decision poses yet another challenge for analyzing DaCapo statically: How can we get access to these classes without knowing this class loader's internals? And how can we re-insert statically transformed classes into these internal JAR files without having to convert our static-analysis tool into a sophisticated build tool? In Section 3 we explain how TAMIFLEX solves this problem by using a pair of instrumentation agents, the Play-out Agent and the Play-in Agent, to obtain classes for static analysis, no matter where they are loaded from, and to re-insert offline-transformed classed into a running system, respectively.

Although our experience with DaCapo motivated our work, we wish to note that the problems that we describe are by no means specific to the DaCapo benchmark suite. Instead, the problems are simply a consequence of using real-world Java applications. When analyzing such programs, these problems will naturally arise sooner or later. One major point of this paper is to show that static-analysis tools can cope with such awkward but realistic features instead of either abandoning soundness by ignoring them or forcing programmers to write programs that avoid such features.

[1]Note that the input data to a few benchmarks (eclipse, tomcat, tradebeans, tradesoap) encompasses further JAR files (plug-ins, web applications, web services).

Benchmark	Description
avrora	a set of simulation and analysis tools in a framework for AVR micro-controllers
batik	a Scalable Vector Graphics (SVG) toolkit that renders a number of SVG files
eclipse	executes some of the (non-gui) JDT performance tests for the Eclipse IDE
fop*	takes an XSL-FO file, parses it and formats it, generating a PS or PDF file
h2	executes a TPC-C like benchmark written by Apache as part of the Apache Derby database, the application models: customers, districts, warehouses, purchases and deliveries
jython	executes (interprets) the pybench benchmark or a small Python program
luindex*	uses lucene to index a set of documents; the works of Shakespeare and the King James Bible
lusearch	uses lucene to do a text search of keywords over a corpus of data comprising the works of Shakespeare and the King James Bible
pmd	analyzes a set of Java classes for a range of source code problems
sunflow	renders a classic Cornell box; a simple scene comprising two teapots and two glass spheres within an illuminated box
tomcat	runs the tomcat sample web applications
tradebeans	a EJB-container version of the Daytrader benchmark from Apache
tradesoap	a SOAP version of the Daytrader benchmark from Apache
xalan	transforms XML documents into HTML

*=only available in sizes small and default, not large

Table 1: The DaCapo benchmarks in the new "bach" release; information taken from DaCapo's release notes

Similarly, we note that TAMIFLEX is by no means limited to DaCapo. Researchers can use TAMIFLEX in combination with any Java program. The same holds for our modified versions of Soot and Spark.

3. TamiFlex

Figure 1 gives an overview of the architecture of TAMIFLEX. 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 `Class.forName()`, `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 (details in Section 4.2), which Figure 1 shows below the program. In this agent, the Tracer, a class-file transformer, instruments the classes `Class`, `Method` and `Constructor` so that calls to methods such as `Class.forName()` generate entries in a log file (shown on the bottom left). The agent further comprises a Dumper component, which writes all classes loaded by the program, including classes that the program's class loaders may have generated on the fly, to a local repository, i.e., a flat directory. Certain class loaders assign randomized names to such classes. To be able to re-identify these classes across multiple runs, the Dumper renames the classes using a hash code over the contents of each class. The Dumper communicates with a Hasher component (Section 4.4) to obtain these hash codes.

Executing a program with the Play-out Agent enabled 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 then update the log, appending information about reflective calls that were not previously observed, and dump additional classes that had not been loaded on previous runs. One can repeat this process until reaching a fixed point.

Next, users can feed the log file and the dumped classes into some static-analysis tool to conduct static analyses, and to transform, e.g., optimize or instrument, the code. We use Soot [39] with Spark [27]. Running Soot results in a set of transformed class files (shown on the bottom right).

The right-hand side of Figure 1 shows what happens when the user runs the program with the second agent, the Play-in Agent (Section 4.3), installed. Whenever the original program is about to load a class `c`, a Replacer within the agent tries to retrieve the offline-transformed version of `c` from the local repository. For classes that bear a randomly generated class name, the agent asks the Hasher component to compute a normalized class name. The Replacer then looks 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. This means that in this case the program will instantiate the class that the class loader originally loaded from "the cloud." Through a command-line option to the Play-in Agent, users can opt to have a warning message issued when such a situation occurs.

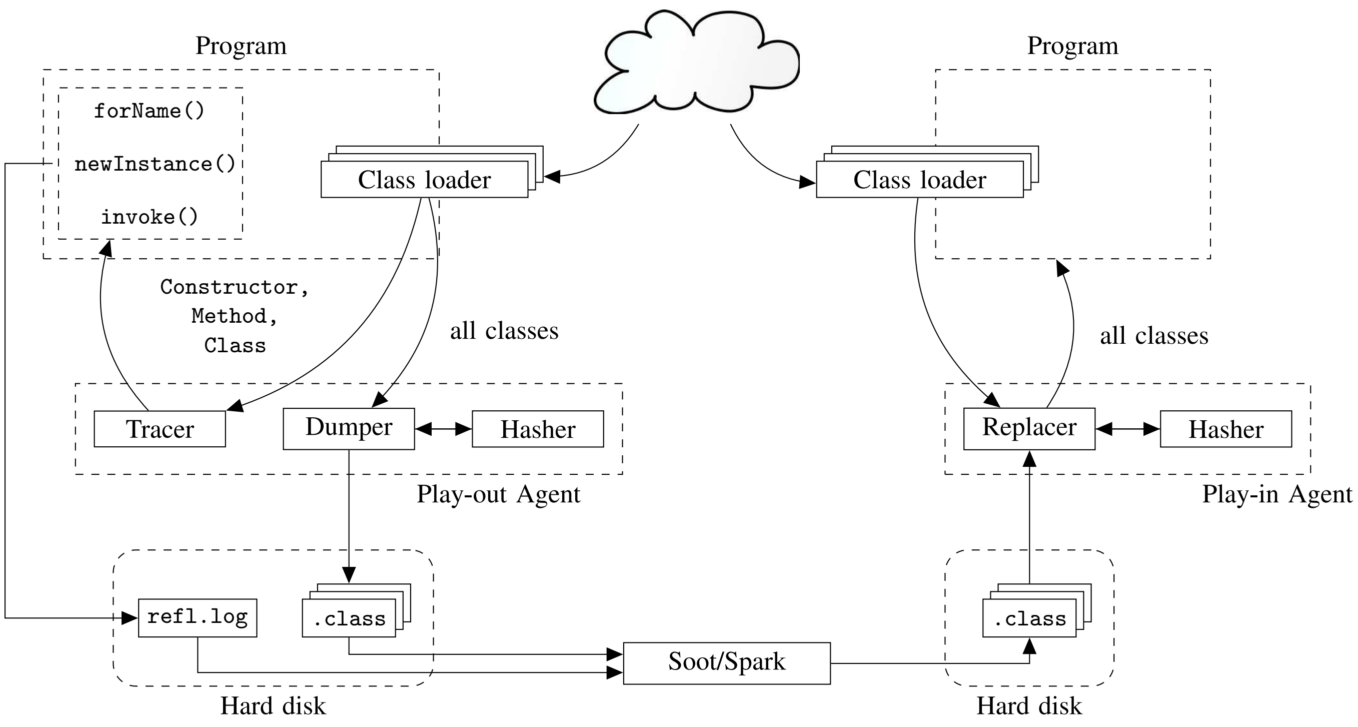

Figure 1: Overview of TAMIFLEX

Note the flexibility of this design; TAMIFLEX works with any Java 6-compatible virtual machine that supports re-transforming classes through the `java.lang.instrument` application programming interface (API). Through this API, 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. Further, TAMIFLEX poses no special restrictions on the static-analysis component (here Soot/Spark), except that it must be able to load class files from disk, to write class files to disk, and to correctly interpret the reflection log file that TAMIFLEX generates.

TAMIFLEX, all our experimental data and all tools to reproduce this data are available at the TAMIFLEX website:

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

4. Java Agents

As Figure 1 illustrates, TAMIFLEX consists of two Java instrumentation agents: the Play-out Agent and the Play-in Agent. Both use the `java.lang.instrument` API. We first give a general overview of this API. Then, in Section 4.2, we explain the Play-out Agent. We explain its counterpart, the Play-in Agent, in Section 4.3. Finally, in Section 4.4 we explain how both agents consistently normalize randomized names of generated classes by hashing on the class contents.

4.1 The java.lang.instrument API

With Java 5, Sun introduced the `java.lang.instrument` API that allows programmers not only to transform classes as they are loaded but also to re-transform classes that have been loaded already. Hereby, programmers manage class transformations through an instrumentation agent that registers a set of class-file transformers. Just before invoking the program's `main` method, the Java virtual machine invokes the agent's `premain` method. To transform a class, the agent passes the byte array that defines this class to a class-file transformer. The transformer can then return a modified byte array, replacing the original definition of the class, or opt to return `null`, indicating that the un-modified class definition should be used. During class *re*-transformations, a transformer may only replace method bodies; all signatures of classes, fields and methods need to remain unchanged.

4.2 Play-out Agent

The Play-out Agent fulfills two tasks: (1) logging information about reflective calls, and (2) dumping all classes to disk that the running program loads or generates.

4.2.1 Logging reflective calls

To log reflective calls, the agent first instantiates the Tracer, a class-file transformer that uses the ASM toolkit [11] to insert instrumentation into methods that may load or call code through reflection (see left of Figure 1). Our experiments will show that, at least for our benchmark set, it suffices to restrict ourselves to the following reflection methods:

`Class.forName(String)` We log the String argument that the program passes to this method.

`Class.newInstance()` We log the qualified name of the class that receives the call.

`Constructor.newInstance(..)` We log the complete signature that is encoded in the constructor object that receives the call.

`Method.invoke(..)` We log the complete signature that is encoded in the method object that receives the call.

The agent instruments the methods by inserting appropriate calls to a small runtime library. At every such inserted call, the library adds a data-set entry to a global set. Every entry contains the target of the reflective call (as explained in the list above), the line number of the call site at which the method was invoked (if available), and the qualified name of the surrounding method. We obtain the last two pieces of information through a stack trace. TAMIFLEX stores log files as lines of semicolon-separated values (CSV file format), which makes it easy to import log files into spreadsheet applications. A typical entry that the fop benchmark generates (without the line breaks) looks like this:

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

This entry tells us that method `iterate` called method `startFop` at line 41, using a call to `Method.invoke`.

The data-set entries are not written to disk until just before the program shuts down: the agent installs a "shutdown hook" [35] that prints all entries to the reflection log. Class-file transformers are generally invoked through multiple threads: every thread that loads a class causes transformers to execute in this thread. By executing the bulk of the work in a shutdown hook, which is executed by one single thread only, we can restrict synchronization to a necessary minimum, lowering thread contention.

It is important to note that the Tracer inserts the calls to our runtime library just before every *return* statement within the reflection methods, and not at the methods' entry points. This matters because we only want to log calls of these reflection methods that return successfully. We found that some programs use, for instance, method calls `Class.forName(c)` to check whether a class `c` exists on the program's classpath. If we logged unsuccessful calls, this would confuse our static analyses, as the analyses would be unable to find all those classes `c` for which the call did not succeed. Interestingly, we also found that this design decision matters for another reason: to guard against buggy programs. The method `ClassUtils$2.run()` in benchmark tradesoap of the DaCapo benchmark suite calls `forName(..)` with "`Ljava.lang.Object;_Helper[]`" as argument, which is not even a valid class identifier. This method call can never succeed, and therefore appears to make no sense.

4.2.2 Dumping all loaded classes

The agent fulfils its second task, dumping class files, through a second class-file transformer, the Dumper. The Dumper's internals are quite simple. When the agent invokes the Dumper passing a byte array for a class `c`, then the Dumper simply stores the contents of the byte array in a mapping that associates `c` with its byte array. Note that, in general, multiple class loaders could load different classes with the same name. When encountering a class `c` for which a byte array is stored already, the Dumper checks whether the new byte array equals the stored one. If it does not, then the Dumper issues an error message.[2] For all the benchmarks that we tested, class names were consistent in the sense that the benchmarks do not load two different classes with the same name.

Similar to the Tracer, the Dumper does not actually dump the class contents to disk until the program shuts down: the same shutdown hook that triggers the writing of the reflection log file also causes the Dumper to write out all stored byte arrays to disk, in the form of `.class` files.

4.2.3 Bringing it all together

Algorithm 2 outlines the Play-out Agent's premain method. As we mentioned, the virtual machine executes this method just before calling the program's main method. First, the agent installs the shutdown hook. Then, in lines 4–6, the agent causes the Dumper to write out class files for all already-loaded classes. In lines 8–10, the algorithm next uses the Tracer to instrument `Class`, `Constructor` and `Method`, so that they log calls using the Play-out Agent runtime library. At line 12, the agent registers the Dumper with the instrumentation API again. This causes the Dumper to also be invoked on all classes that are still about to be loaded. Note that the Tracer does not remain active: after (re-)transforming the classes `Class`, `Constructor` and `Method`, it is no longer needed.

Users can instruct a Java virtual machine to use the Play-out Agent when running a program by a simple set of command-line options. For instance, the following command will execute a program using class `Main` from the current directory.

[2]Note that the perfect solution would be to store classes under a class name that is qualified by the defining class loader. We considered this design choice, but opted against it. Firstly, it would be hard to uniquely identify class loaders across multiple runs, but this is necessary so that our Play-in Agent can re-discover the correct class files. Secondly, none of the static-analysis tools that we know of is class-loader aware. Therefore, qualifying class names by a class-loader identifier would most likely yield complications with these tools.

```
premain(Instrumentation inst, String[] agentArgs) {
  Runtime.getRuntime().addShutdownHook(HOOK);

  inst.addTransformer(dumper, CAN_RETRANSFORM);
  inst.retransform(alreadyLoadedClasses());
  inst.removeTransformed(dumper);

  inst.addTransformer(tracer, CAN_RETRANSFORM);
  inst.retransform(Class.class, Constructor.class, Method.class);
  inst.removeTransformed(tracer);

  inst.addTransformer(dumper, !CAN_RETRANSFORM); }
```

Figure 2: premain method of Play-out Agent (pseudo code)

```
java -javaagent:poa.jar=/classdir Main
```

To enable the Play-out Agent, the user has instructed the virtual machine to use `poa.jar` as an agent, using the special `-javaagent` command-line parameter. The virtual machine passes the "`/classdir`" portion of the command line directly to the agent's premain method. This causes the Play-out Agent to place all class files and the reflection log file into the directory `/classdir`.

4.3 Play-in Agent

The Play-in Agent uses a second class-file transformer to re-insert offline-transformed classes into a running program, irrespective of the program's class-loader setup. Using the Play-in Agent, the user can just start the program using a command line as follows:

```
java -javaagent:pia.jar=/classdir Main
```

When started this way, the virtual machine will first invoke the Play-in Agent's `premain` method, and then invoke the program as usual, invoking `Main.main(..)`. The Play-in Agent's `premain` method first registers its own class-file transformer, the Replacer. Then, the agent uses the Replacer to replace all already-loaded, re-transformable[3] classes by the contents of the respective `.class` files from the directory `/classdir` that the user provided as a command-line option. Afterwards, the class-file transformer remains active to replace in the same way all classes that still get loaded: for every such 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.

This may seem like wasted work. After all, why should we first load the definition of a class to then ignore it? There is two reasons for this design. Firstly, as mentioned earlier, we may need the original class definition to compute a normalized name for the class in case the class carries a randomized class name. (We will give details in the following section.) In these cases, the original byte array is therefore not ignored entirely: it is used to compute a hash code. Secondly, the `java.lang.instruments` API offers no way to prevent class definitions from being loaded (or generated) before a class-file transformer is invoked.

[3]Not all virtual machines support re-transformation and not all classes are re-transformable. Array classes, for instance, are not, as the virtual machine generates them on demand.

4.4 Normalizing randomized class names

The above scheme of storing and re-inserting classes crucially depends on the assumption that the name of any class uniquely identifies that class. However, in general, there can be multiple classes with the same name, and even multiple names for virtually the same class. The first case can happen when multiple class loaders load different classes that have the same class name. As we explained the previous section, we ignore this issue because we found that it does not seem to occur in practice, at least not in the programs that we tested. The second issue, however, multiple names for the same class, is worth treating.

We found that the programs that we tested TamiFlex on generate classes at runtime, and sometimes even assign randomized names to these classes. This includes class names of the form `GeneratedConstructorAccessor42` or `SomeClass$$EnhancerByCGLIB$$4ac69885`. These names are random in the sense that multiple equivalent program runs will generate different class names. This is detrimental to our purposes for two reasons. Firstly, with every new run our Play-out Agent will create new entries in the reflection log, because the agent discovers reflective calls to classes with names that the agent has not seen before. This prevents us from easily determining when the reflection log is "stable", i.e., when it reaches a fixed point at which we have seen all the kinds of reflective calls that the program can execute. Secondly, we would like to offline-transform classes with randomized names in the same way as we can transform all other classes. The randomized names, however, would cause the Play-in Agent to look for a class under a name that is different from the name that the Play-out Agent used to store the class on disk.

We therefore decided to enhance both the Play-in Agent and the Play-out Agent with an additional mechanism to normalize such randomized class names. Normalization assigns the same names to the same classes, even over multiple runs. We implement normalization using a Hasher component that both agents have access to.

The Hasher component generates a new, normalized class name, based on the contents of the class. Because this is computationally expensive, we compute normalized names only for classes that would otherwise carry a randomized name. To identify such classes, agents carry a user-definable list of infix strings. When a class name contains the infix, the agent assumes that the fraction of the class name that follows the infix may be randomized. For the DaCapo benchmark suite, we identified the following infixes:

- `$Proxy`
- `ByCGLIB`
- `GeneratedConstructorAccessor`
- `GeneratedMethodAccessor`
- `GeneratedSerializationConstructorAccessor`
- `org.apache.derby.exe.`

This means, for example, that the agent would identify the class `GeneratedConstructorAccessor42` as a class with randomized name, and would infer that `42` is the random portion of that name. The Hasher would then replace this name with a normalized name of the form `Generated-ConstructorAccessor$HASHED$a3f4e2b3`, where the suffix `a3f4e2b3` is the hexadecimal hash code for this class.

4.4.1 Computing normalized names

For a class `c`, we compute `c`'s normalized name *normalize(*`c`*)* as follows:

1. Use ASM to disassemble the byte array that defines `c`.
2. Replace all references to the randomized name `c` by the string `$$$NORMALIZED$$$`. (We replace type references and fully qualified references in string constants.)
3. Replace all references to another class c' with normalized name *normalize(*c'*)* .
4. Use ASM to re-convert the resulting class to a byte array.
5. Compute the SHA1 [33] hash over this byte array.

We perform Step 3 because the hash code for `c` would not be stable without replacing c' by its normalized counterpart: we would get different hash codes for `c`, depending on what randomized name that class generator chose for c'. In a first attempt, we tried a simpler approach that would replace c' by `$$$NORMALIZED$$$`, too. However, this yielded hash collisions, assigning the same normalized name to different classes. Such collisions would break a program running with the Play-in Agent, as the agent would likely confuse offline-transformed classes.

Note that Step 3 implies recursion: to compute the normalized name of `c` we need to compute the normalized name of c'. This may cause an infinite recursion, when two classes with randomized names `c` and c' reference each other. Fortunately, all the classes with randomized names that we observed in our experiments only had a tree-like reference structure; there are no cycles. Our agents can therefore compute a dependency tree and compute hash codes starting at leaf nodes. If a case occurred in which the dependency "tree" contains cycles, our agent would not assign normalized names to classes within the cycle and would issue a warning. A general solution to this problem would have to use a fixed-point iteration to assign normalized names to entire strongly-connected components of classes. We leave such a solution to future work.

We avoid hash collisions by hashing on the complete definition of a class, i.e., its complete byte array, modulo references to the class itself. Because we replace references to other randomized names by their normalized class names instead of a constant, the hash code is also sensitive to references to those classes. Assuming a perfect hash function that never causes collisions by itself, we therefore know that two classes can only result in the same hash code when they are in fact equal, and in particular reference equal classes. The SHA1 hash function which we use has a very small chance of producing accidental collisions: 2^{-160}.

4.4.2 How the agents use normalized names

In the Play-out Agent, we normalize class names at the end of the program run, when executing the agent's shutdown hook. The Tracer replaces each reference to a randomized class name in the reflection log by its normalized counterpart. In addition, the Dumper uses the ASM toolkit to replace in each stored byte array all references to randomized class names `c` by their normalized counter part *normalize(*`c`*)*. When the Dumper stores class files on disk, it also uses the normalized name. This causes the normalized reflection log to be consistent with the dumped class files.

The Play-in Agent computes normalized class names on the fly. When the virtual machine invokes the Replacer for a class `c` with a randomized name, the agent computes the normalized name *normalize(*`c`*)* in the same way as above. In cases where `c` references another randomized class name c', this requires that the normalized class name for c' can be computed as well. This is only possible when c' is already known to the Replacer. Fortunately, in our experiments it was always the case that in such situations the virtual machine would present c' to the Replacer before `c`. Therefore, computing hash codes posed no special challenges to the Replacer. Again, this simple scheme would break in the case of circular references among classes with randomized names.

After the Replacer has computed *normalize(*`c`*)*, it tries to retrieve a stored `.class` file under that name. When no such class file is found, the Replacer returns the original byte array, optionally issuing a warning. When the class file is found on disk, the Replacer loads its contents into a fresh byte array. Then the Replacer again uses ASM to replace within this byte array all references to normalized class names by their un-normalized, i.e., randomized counterparts. This assigns to the class the "correct" names that the running program expects in the context in which the Replacer was called. We determine the correct names for the current context by using ASM to discover references in the original byte array that the program passes to the Replacer.

5. Modifications to Soot

Soot [39] is one of the most widely used static-analysis frameworks and researchers have implemented countless static program analyses on top of Soot. The fact that Soot comes pre-equipped with Spark [27], an efficient pure-Java framework for flow-insensitive and context-insensitive points-to analysis and call-graph construction, makes Soot particularly attractive for people who aim at implementing static *whole-program* analyses. In the following, we describe how we extended Spark to make use of the information that TAMIFLEX's Play-out Agent provides, enabling whole-program analysis for programs that use reflection and custom class loaders.

5.1 Pointer assignment graphs & call graphs in Spark

Spark computes both points-to sets[4] and a call graph using a fixed point iteration. Spark starts by inspecting an initial set of possible entry methods, which comprises the program's main method and the static initializers of all classes that the program references. For each such method, Spark inserts nodes into a global "pointer assignment graph." Spark first discovers all allocation sites within these methods and creates an "alloc node" for each such site.

Next, Spark creates "variable nodes" for all variables (including fields) that the method references. Spark also wires these nodes with edges, according to all possible assignments within the method. (This includes non-trivial assignments like, for instance, assignments to formal parameters of outgoing method calls.) Spark can then use the pointer assignment graph to compute points-to sets: for every variable `v`, the allocation sites that may reach `v` in the pointer assignment graph form `v`'s (intermediate) points-to set *points-to(*`v`*)*.

Next, Spark determines the possible targets of method calls that originate from the current method. Method calls induced by `invokestatic` and `invokespecial` bytecodes are simple to resolve, as their call targets are determined statically. For method calls induced by `invokeinterface` and `invokevirtual` bytecodes, however, Spark needs to determine the possible receivers that the Java runtime could invoke using dynamic dispatch. This is where the points-to sets come into play again. For a call `v.foo()`, Spark can overestimate the set of receivers through the set *points-to(*`v`*)* that it computed above: if `v` can point to an object of type `Foo`, then `Foo.foo()` is a possible receiver of the call `v.foo()`. To every call target determined that way, Spark inserts an edge into the global call graph.

As new methods become reachable by expanding the call graph, new allocation sites become reachable, too. Therefore Spark iterates the process of computing points-to sets and expanding the call graph until it reaches a fixed point.

5.2 Native methods and reflection in Spark

This way of computing points-to sets and a call graph works very well for closed programs, for which all source code or bytecode is known, and which reference code only through explicit method and constructor calls. However, one needs to take care when analysing programs that use native calls and reflection. Although most researchers will not be interested in analyzing native code itself, native calls may, in certain cases, call back into the program's own bytecode.

One special group of native calls[5] comprises calls to the Java reflection API. In this paper, we frequently name these calls "reflective calls". Ignoring such calls can lead to a call graph and points-to sets that are unsound even for the pure-Java portion of the program.

Spark therefore includes a framework for native-method simulation that allows researchers to automatically insert points-to relationships for a pre-determined set of native calls, recognized by signature. Of particular interest to call-graph construction are calls to the methods `Class.forName`, `Class.newInstance`, `Constructor.newInstance` and `Method.invoke`. The left-hand side of Figure 3 shows how the *default* version of Spark handles these calls.

At a statement "`c = Class.forName(String)`", Spark connects the variable node for `c` to a fresh alloc node of type `Class` in the pointer-assignment graph, capturing the return value of the call. Loading a class may invoke the static initializer of that class. To extend the call graph appropriately, Soot first tries to determine the name of the loaded class from the string argument to the method call (not shown in the figure). This sometimes works when the program uses a string constant to refer to the name of the class. For all other cases, Soot allows users to provide a special command-line flag "`-dynamic-class`", listing the set of names (`D` in the figure) of all classes that the program may load using `Class.forName` at runtime. For every such class `c` $\in$ `D`, Spark inserts a call edge to `c`'s static initializer `<clinit>`.

Spark treats calls to `Class.newInstance` by creating and connecting alloc nodes for all user-provided "dynamic classes" and by creating call-graph edges to the zero-argument constructors of these classes. (Spark cannot infer class names from string constants in this case because there is no string argument to this call.)

Calls to `Constructor.newInstance` are treated the same, except that Spark creates call-graph edges not only to zero-argument constructors but to all constructors of all "dynamic classes".

Regarding calls to `Method.invoke`, it is important to note that the default version of Spark does not treat such calls soundly. A possible sound solution would assume that *every* method could be called. However, this would result in a call graph so conservative that it would likely be useless. Spark therefore ignores such calls entirely, except for optionally issuing a warning message.

[4]A points-to set for a variable `v` is the set of all allocation sites that create objects that may reach `v`. A call graph is a directed graph that contains an edge `m` $\rightarrow$`n` if method `m` may call method `n`.

[5]There are other kinds of native methods that call back into Java bytecode, for instance calls that invoke shutdown hooks [35] or finalizers. But these calls are not often relevant and we ignore them for now.

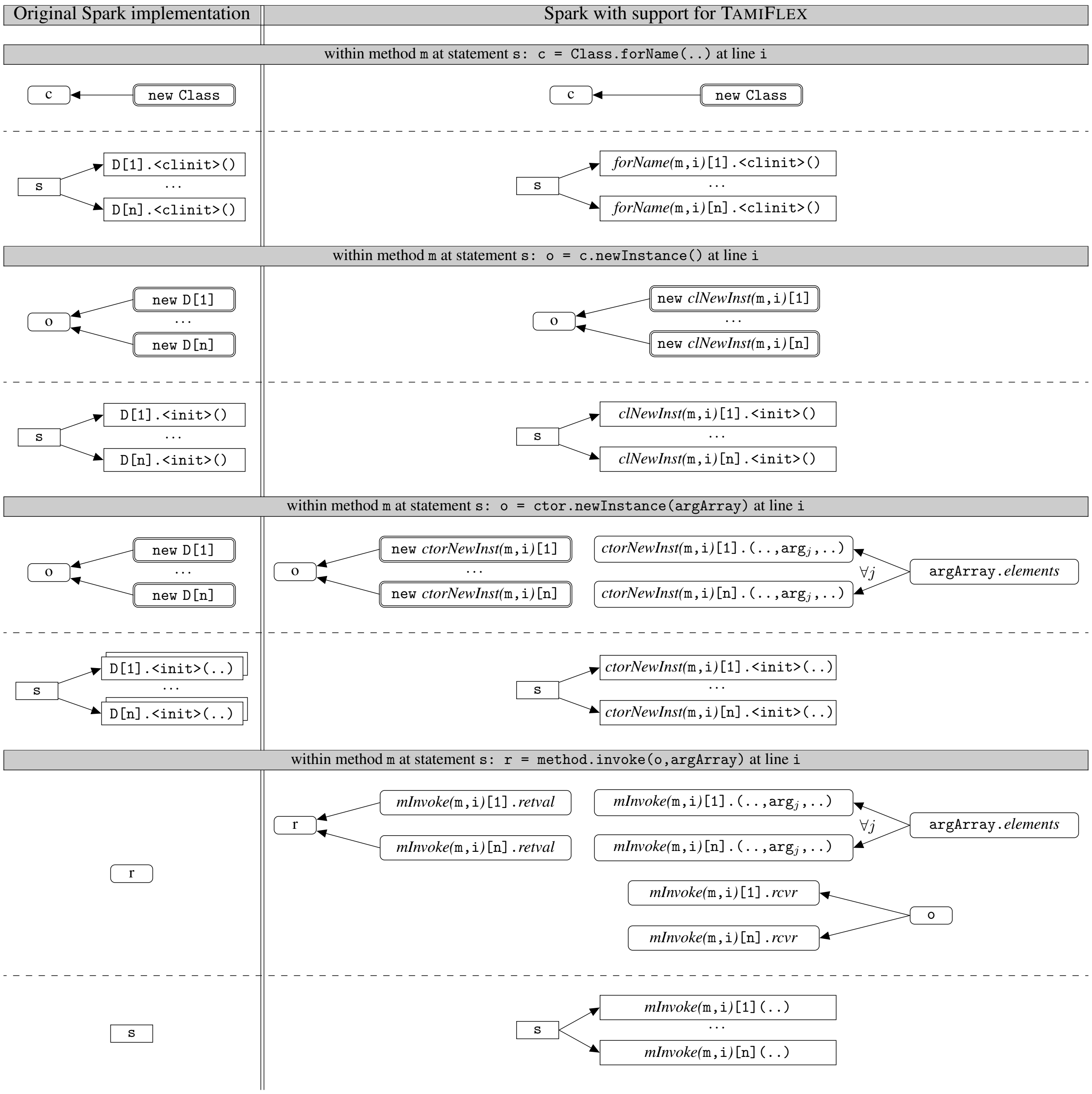
Original Spark implementation
Spark with support for TAMIFLEX
within method m at statement s: c = Class.forName(..) at line i
c
new Class
s
D[1].<clinit>()
D[n].<clinit>()
forName(m,i)[1].<clinit>()
forName(m,i)[n].<clinit>()
within method m at statement s: o = c.newInstance() at line i
o
new D[1]
new D[n]
new clNewInst(m,i)[1]
new clNewInst(m,i)[n]
D[1].<init>()
D[n].<init>()
clNewInst(m,i)[1].<init>()
clNewInst(m,i)[n].<init>()
within method m at statement s: o = ctor.newInstance(argArray) at line i
new ctorNewInst(m,i)[1]
new ctorNewInst(m,i)[n]
ctorNewInst(m,i)[1].(..,arg_j,..)
ctorNewInst(m,i)[n].(..,arg_j,..)
∀j
argArray.elements
D[1].<init>(..)
D[n].<init>(..)
ctorNewInst(m,i)[1].<init>(..)
ctorNewInst(m,i)[n].<init>(..)
within method m at statement s: r = method.invoke(o,argArray) at line i
r
mInvoke(m,i)[1].retval
mInvoke(m,i)[n].retval
mInvoke(m,i)[1].(..,arg_j,..)
mInvoke(m,i)[n].(..,arg_j,..)
mInvoke(m,i)[1].rcvr
mInvoke(m,i)[n].rcvr
mInvoke(m,i)[1](..)
mInvoke(m,i)[n](..)

Figure 3: Construction of pointer assignment graph and call graph.
For each of the four kinds of reflective call sites `Class.forName`, `Class.newInstance`, `Constructor.newIntance` and `Method.invoke` we show the nodes that Spark creates in the pointer assignment graph (portion above each dashed line) and call graph (below dashed line). On the left, we show Spark's algorithm as it was originally, on the right we show Spark with support for TAMIFLEX. Nodes with rounded corners and double lines represent allocation sites. Nodes with rounded corners and single lines represent variables. Rectangular nodes represent methods in the call graph. `D` is the list of user-provided "dynamic classes". The methods `forName`, `clNewInst`, `ctorNewInst` and `mInvoke` extract the respective lists of reflective call sites in `m` from the log file, identifying `m` by its name and the given line number `i`.
We use special names for certain nodes in the pointer assignment graph. "*elements*" nodes denote the union of all elements of an array. "*retval*" nodes denote the return value of a method. "*rcvr*" nodes represent the method's receiver, i.e., "`this`". "arg_j" nodes represent argument variables of a method call at the side of the callee.

5.3 Improving Spark through Reflection Logs

We modified Spark so that it constructs its pointer assignment graph and call graph using the reflection log that TAMIFLEX produces. In result, users no longer have to specify dynamically loaded classes manually using the "`-dynamic-class`" option. This not only eases the burden on the user, it also improves soundness and precision of Spark in the following ways.

With respect to calls to `Method.invoke`, Spark can now handle these calls in a precise and, more importantly, sound way. (See bottom right of Figure 3.) When encountering a statement "`r = method.invoke(o,argArray)`" at line `i` within a method with name[6] `m`, Spark searches the log file for `Method.invoke` entries that were logged for this very source location. Spark then creates call-graph edges to the possible receiver methods of this call, as taken from the entries in the log file.

In the pointer-assignment graph, Spark connects `r` to the variable nodes that model the return values of these methods, and `o` to the nodes that model the receiver "`this`" within the methods. This models that `r` and `o` may point to the values represented by those nodes. To soundly model the argument hand-over between the call to `Method.invoke` and the possible callees, Spark further connects the node that models the contents of the `argArray` to every (reference-typed) parameter variable in any of the possible callees. In sync with Spark's usual handling of array elements, we do not distinguish between different array elements. Hence, within the method that was called reflectively it may appear to Spark that the method's arguments are aliased even when in reality they are not.

We handle `Class.forName`, `Class.newInstance` and `Constructor.newInstance` similarly (see Figure 3). In these cases, using the log file enhances precision: instead of creating edges and alloc nodes for all dynamic classes at every reflective call site, Spark can use the log file to determine for every dynamic call site $(\mathtt{m}, \mathtt{i})$ separately which classes may be loaded and which constructors may be called at that particular site. In particular, at calls to `Constructor.newInstance` we do not need to create edges to all possible constructors but only to the ones that were actually called.

Calls to constructors of abstract classes When modifying Spark to take into account TAMIFLEX's reflection log files, we initially found Spark complaining about attempts to generate alloc nodes for abstract classes. Creating such alloc nodes usually makes no sense because in Java allocating objects of an abstract class is not allowed: Java compilers prevent programmers from calling `new` on an abstract class. The Java runtime library further forbids the instantiation of abstract classes through reflection: invoking `newInstance` on a `Constructor` object of an abstract class results in an `InstantiationException` being thrown.

To our big surprise, though, we found that the Java runtime library itself indeed does successfully call constructors of abstract classes through reflection, when using serialization: both DaCapo benchmarks tradebeans and tradesoap deserialize certain data structures, which causes the runtime library to call the default constructor of the abstract class `java.util.AbstractSet`. The serialization interface appears to explicitly circumvent the usual restrictions of the reflection interface.

The explanation for this somewhat unconventional technique is that during de-serialization the Java runtime always initializes the de-serialized object, e.g. of type `HashSet`, by calling the default constructor of the object's first non-serializable super class; in this case this is `AbstractSet`. The serialization API then initializes all fields not defined by `AbstractSet` but by `HashSet` by reading the respective data values from the serialization input stream.

To cope with this subtlety, we modified Spark so that it allows for creating alloc nodes for abstract classes when being in "TAMIFLEX mode".

5.4 Phantom Classes

Early on, Soot included a concept of "phantom classes", representatives of classes for which no definition is known. These classes act as a sentinel, informing analyses that the portion of the program that Soot should regard ends here.

Soot creates phantom classes on demand, when encountering a reference to a class for which it cannot find any definition on its classpath. Likewise, when encountering a reference to an unknown field of a phantom class, Soot creates a phantom field with the correct signature. For method references Soot creates phantom methods with empty bodies.

Ideally, all of Soot's components should be able to properly deal with phantom classes, but in reality Soot's support for phantom classes has long been incomplete. Users of Soot frequently avoided this problem by disallowing Soot to use phantom classes in the first place. This is only possible, however, when instead giving Soot access to the definitions of all classes that the analyzed program references statically.

As we explained in Section 2, this is unrealistic in our scenario: the Play-out Agent only dumps classes that get loaded, but not every class that a program references statically will indeed be loaded at runtime. We therefore need a working mechanism like phantom classes to conduct static analysis

[6]Note that the log file holds only the name of the method that contains the reflective call site, not the full method signature. This is because TAMIFLEX creates the entry using a stack trace and Java's stack traces lack full signatures. In theory, method names without signatures may render Spark unable to infer the correct method when multiple overloaded methods with the same name exist. In practice, however, this is not problematic: Soot can usually infer the correct method using line-number information. In the few cases where no such information is available in the bytecode and the calling method is indeed overloaded, we soundly assume that the log-file entries could apply to any of the overloaded versions of the method.

in conjunction with TAMIFLEX. To make phantom classes work, we had to modify[7] Soot in the following ways.

First, Soot contains checks that throw an exception when attempting to access the super class of some class c and that super class has not been set. We had to modify this check so that no exception would be thrown if c is a phantom class. Second, Soot models both classes and interfaces using the same concept, "Soot classes". Phantom classes are just Soot classes with a special marker. When creating a phantom class, Soot creates this phantom class because it has no access to the definition of the class. Therefore, Soot cannot know if the phantom class actually represents a class or an interface. By default, Soot assumes that a phantom class represents a real class, not an interface. But Soot also contains checks that validate that an `interfaceinvoke` expression only invokes methods of interfaces, not classes. With interfaces represented as phantom classes, this check would fail: Soot sees a reference to a phantom *class*, not a "phantom interface". We solved this problem by modifying Soot to not perform the check when the target of an `interfaceinvoke` is a phantom class. Third, Soot contains a type-inference engine [3] for inferring the declared types of local variables. This is necessary because Java bytecode contains no such type information. The type-inference engine uses the class hierarchy to determine an appropriate declared type for variables that may hold object references of different types. Phantom classes have no super classes or super interfaces. Originally, this confused the type-inference algorithm, causing exceptions. We modified the inference engine to not take into account phantom classes when computing type constraints. This avoids the exceptions, nevertheless results in sound type assignment.

5.5 Incorrect references in dormant code

We also needed to make some minor adoptions to Soot to gracefully handle incorrect references in dormant code. Jython, for example, contains an `invokevirtual` bytecode that attempts to invoke a static method. If execution ever reached this bytecode, an exception would be thrown. Similarly, both the tradebeans and tradesoap benchmarks contain a reference to a non-existing method, presumably as a result of broken separate compilation. Soot normally refuses to accept programs that contain such errors. We modified Soot so that it would accept such programs in "TAMIFLEX mode". When encountering a method reference with "wrong staticness", this problem is just ignored, leaving the method call it is. When encountering a reference to a non-existing method, Soot creates this method on the target type but inserts a method body that will throw an `Error` when the program's execution ever reaches the respective call. This is similar to other Java compilers which often insert statements that throw an "unresolved compilation error" in such situations.

Reflective call sites in dormant code Another problem that we encountered with dormant code, i.e., code that appears reachable statically but does not actually execute on program runs, occurs when such dormant code itself contains a reflective call site. When Soot encounters such a piece of code during call-graph construction, it consults the log file for information about the possible call targets at this site. But when the code is dormant, the log file cannot contain an entry for this site.

By default, our extension to Spark simply ignores such reflective call sites, adding no edges to the pointer assignment graph or call graph. This is sound if the code is not only dormant but actually dead, i.e., will never execute on any program run. If this assumption does not hold, i.e., if runs exist that do execute the thought-to-be-dormant code, the omission of edges may yield unsound results.

To give some form of guarantee, we allow programmers to instruct Soot to insert guards at the reflective call sites in questions. Depending on the chosen option, when processing a reflective call site for which the log file contains no information, Soot inserts code that will either throw an `Error` at runtime, aborting the program run, or simply print a stack trace so that the program run can resume despite the error. This instrumentation will notify the programmer when executing the program in a way that deviates from the executions that were previously recorded.

6. Experiments

In this section we present experimental evidence that shows that programmers can effectively use TAMIFLEX and static-analysis tools like our improved version of Soot to conduct static whole-program analysis of programs that use reflection, custom class loaders, and runtime-generated classes. First, in Section 6.1 we show that the call graphs we obtain through Soot and TAMIFLEX are sound with respect to all program runs that the programmer recorded using TAMIFLEX's Play-out Agent. In Section 6.2 we show that programmers can collect stable log files even for programs that do not execute entirely deterministically, such as multi-threaded programs. Section 6.3 discusses the amount of code coverage that is required to obtain meaningful log files with TAMIFLEX. In Section 6.4 we show that applying TAMIFLEX to large programs with large inputs is not problematic: Both our agents induce a runtime overhead of usually below 10%. The Play-in Agent even induces virtually no overhead at all once all classes have been loaded. In Section 6.4 we discuss the time and memory requirements of Soot when run in "TAMIFLEX mode.". In Section 6.5, we provide a summary, in Section 6.6, we discuss threats to the validity of our experimental design.

[7] Parts of this modification were contributed by Syed Albiz and Patrick Lam from the University of Waterloo. We are grateful for their contribution.

6.1 Sound call graphs

The main reason for using a TAMIFLEX-generated log file in a static analysis is that the analysis can use this information to obtain a complete picture of the program's calling structure, i.e., a complete call graph. In general, when we speak of "complete" call graphs in the context of TAMIFLEX, we always refer to call graphs that are complete with respect to the recorded runs. For a static Spark-generated call graph to be sound in that sense, the graph must contain an edge $\mathtt{m} \rightarrow \mathtt{n}$ 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 uses as input to Spark.

Obtaining dynamic call graphs To test whether our modifications of Spark 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 runs. If the dynamic graphs contain the static ones, this confirms that that the static call graphs are sound in the above sense. But obtaining dynamic call graph is a non-trivial task in itself. For the purpose of this evaluation, we wrote a native JVMTI [25] 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` (or even `premain`) are called; in particular, the agent can record calls from native code back into Java bytecode. We believe that the dynamic call graphs that this JVMTI agent produces are as complete as possible without modifications to the underlying virtual machine.

Producing these call graphs incurs huge runtime overheads, ranging from 124x (batik) to 8587x (sunflow). Multi-threaded benchmarks suffer particularly from slowdown, as tracing requires synchronization upon every method call. This prevented us from recording dynamic call graphs for the tradebeans and tradesoap benchmarks: in both cases hard-coded timeouts within the exercised Apache DayTrader [14] cause the benchmarks to diverge from their usual execution path when being executed so slowly. (Note that users of TAMIFLEX do not require this agent at all. They only use the Java-based Play-in Agent and Play-out Agent, both of which incur far less overhead, as we will show in Section 6.4).

Comparing dynamic and static call graphs After recording dynamic call graphs with our JVMTI agent, we used Lhoták's call-graph differencing tool PROBE [26] to compare these dynamic graphs to the static call graphs that we compute with Soot and TAMIFLEX. The tool was of great help to us, as it not only shows a list of all methods that are reachable in one graph but not in the other, but also creates a ranked list of "critical" edges that lead to sub-components that only exist in one graph. The comparison with dynamic call graphs indeed helped us find an omission in our initial modifications to Spark; for calls to `Constructor.newInstance` we had forgotten to insert one of the necessary edges in the pointer-assignment graph (c.f. Figure 3). Our results confirm that our current version of Spark does indeed produce sound call graphs for all DaCapo benchmarks (with the exception of tradebeans and tradesoap, for which we do not know if the static call graphs are sound because we could not record dynamic graphs, as discussed above). Our implementation of TAMIFLEX, the JVMTI agent, all static and dynamic call graphs and the diff information produced by PROBE are available for future reference at the TAMIFLEX website.

6.2 Stability of log files

One important criterion for the feasibility of our approach is the stability of the reflection-log files that our Play-out Agent generates. The agent may, of course, produce different log files for different program inputs, but it should generate the same log files for the same inputs. However, even for the same inputs, programs can generate different program runs, due to concurrency.

To confirm that our implementation generates stable log files, we ran every benchmark 10 times, one iteration at a time, with the Play-out Agent enabled. After every single run, the agent's shutdown hook causes the benchmark to report the number of new entries added to the reflection log. Table 2 summarizes these numbers. As the results show, the benchmarks tomcat, tradebeans and tradesoap require multiple runs until they reach a plateau for which we have reason to believe that it is the fixed point for this input. All of these three benchmarks are multi-threaded. At the current time it remains unclear as to why exactly different schedules cause different classes to be loaded but we provide all log files on the TAMIFLEX website, for others to inspect.

6.3 Effect of input size

We next sought to determine how much the quality of a log file that TAMIFLEX produces depends on the code coverage of the program run that produces this log file. We model different levels of coverage through different input sizes.

Do larger inputs yield better coverage? The DaCapo "bach" release offers up to four input sizes for each benchmark: small, default, large and huge. Because "huge" only exists for a small subset of benchmarks, we restrict ourselves to the other three input sizes: "small," "default," and "large." By the names of these input sizes, one could expect that "large" always yields better code coverage than both "default" and "small," etc. To not merely rely on such assumptions, we measured the relative coverage that the different inputs yield. 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.

Figure 4 shows the result of this process as a set of twelve Venn diagrams [40] (again, as discussed above we cannot produce dynamic call graphs for tradesoap and tradebeans).

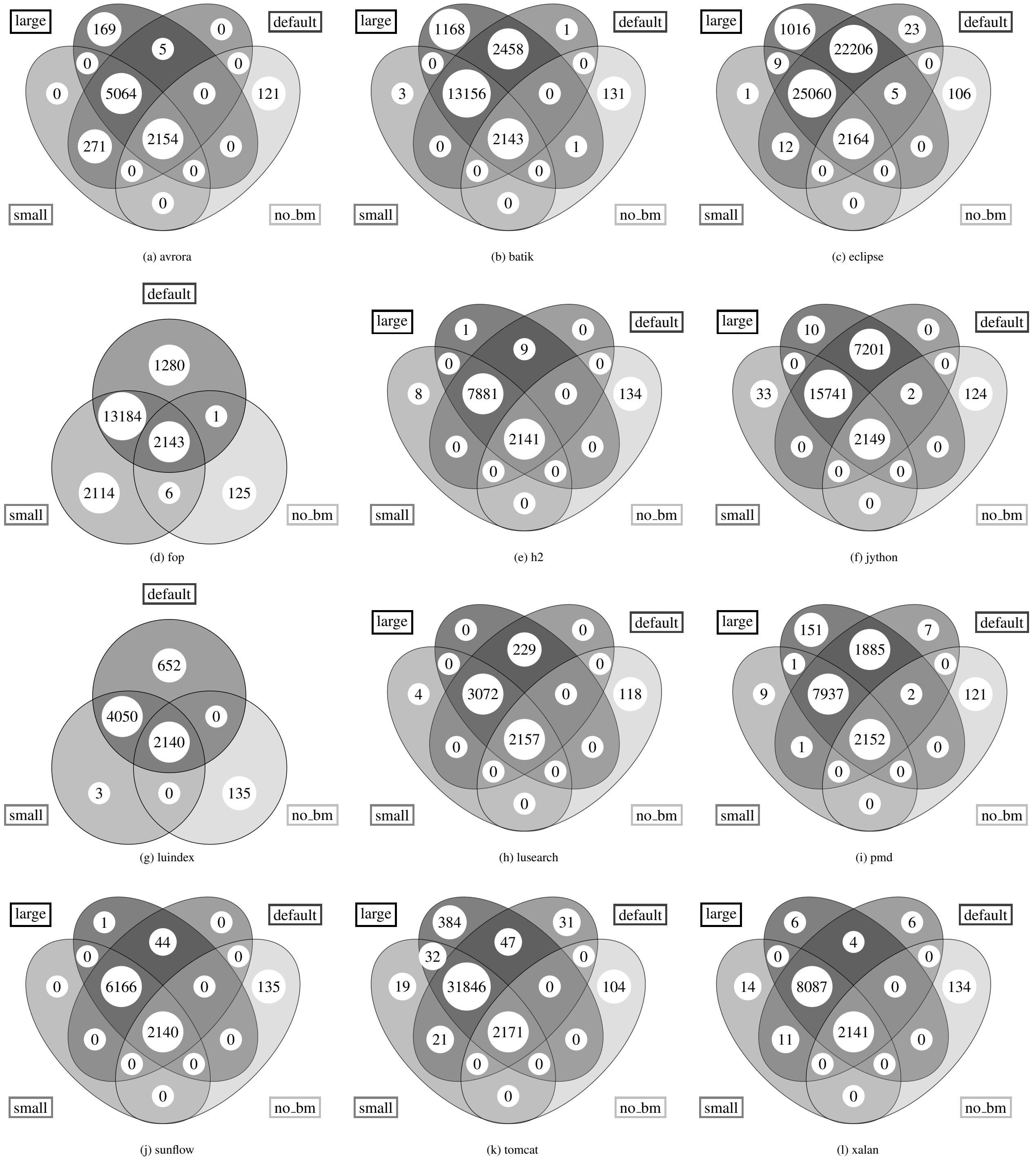

Figure 4: Venn diagrams [40] showing the number of reachable methods shared by the dynamic call graphs at different input sizes (small / default / large) and with a run of the DaCapo suite without selecting a benchmark, merely printing the harness' usage message (no_bm); fop and luindex have no large inputs

	small										default										large									
Run #	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9
avrora	36										36										30									
batik	55										64										64									
eclipse	387										593										595									
fop	321										295														n/a					
h2	54										54										54									
jython	161										267										267									
luindex	77										86														n/a					
lusearch	52										56										56									
pmd	71										77										79									
sunflow	61										61										61									
tomcat	568		2								596			2		2					598								2	
tradebeans	2803		10								2807			10							2807									
tradesoap	3076				6						3104			10							3106					10			6	
xalan	201										201										201									

Table 2: Number of new reflection-log-file entries discovered in each of 10 runs

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 deliberately diverges from the benchmark's normal execution. As the figure shows, most benchmarks have a large overlap between all three input sizes. For instance, avrora has 5064 methods that are reached no matter what input size is chosen. The number of methods that is covered by all four configuration (including "no_bm") is virtually constant for all benchmarks, at around 2150. These are methods reachable from the bare benchmarking harness. Similarly, there appears to be an almost constant number of methods that is only reachable on the "no_bm" run: around 125. Often there is also indeed a significant number of methods covered by "default" and "large", which is not covered by "small." Batik, for instance has 2458 such methods but it has only three methods that "small" covers but the other configurations do not. An interesting outlier is fop; its "small" input appears to induce a call graph that differs significantly from those induced by the "default" runs. (The small input causes fop to generate a PDF file, while "default" causes fop to generate a PS file.) We can also see that, while "default" is often a super-set of "small," the "large" configurations often do not seem to add much coverage. For instance, in h2 there is only a single method that the benchmark reaches on a "large" input but not on the other inputs. To conclude, the assumption that "larger" inputs yield better coverage appears generally correct, but there are exceptions: for instance, the number of methods covered by fop-small is larger than the number of methods covered by fop-default.

Impact of input size on quality of log files An easy metric for the quality of a reflection log file is the number of reflective call sites that it covers. The more call sites it covers the more call sites Spark can model soundly using the information in the log file. In Table 3 we show the number of reflective call sites covered by the log file for every benchmark configuration. We were pleasantly surprised to see the number of covered reflective call sites is *not* heavily correlated with the code coverage of the respective benchmark run. For many benchmarks, the number of covered sites does not increase at all for larger inputs, or only increases slightly. For avrora, the small and default inputs even reach more reflective call sites than the large input. Considering Figure 4a, this is not surprising: the small and default inputs cover 271 methods that the large input does not cover.

Even when a larger input does not cover more reflective call sites, the larger input could yield more varied kinds of reflective calls (to more targets) at these call sites. But looking back at Table 2, we can see that also the increase in total log-file entries seems unproblematic: only for few benchmarks does the number of generated entries differ significantly among different input sizes. We conclude that even program runs that expose relatively bad code coverage do nevertheless frequently cover many reflective call sites. Therefore, users of TamiFlex will not necessarily have to aim for complete code coverage to produce at least "almost sound" analysis results.

Impact of input size on number of phantom classes In Section 5, we explained how we use Soot's concept of phantom classes to deal with dormant code that appears to be live only from a static point of view. Fewer phantom classes mean more input information for the static analysis. Therefore, having to model fewer classes as phantom classes would be desirable. How does a change in code coverage impact the number of phantom classes? In Table 4 we give the number of phantom classes that Soot generates

	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

Table 3: Number of reflective call sites in log file

	small	default	large
avrora	152	152	154
batik	420	407	383
eclipse	765	554	554
fop	510	522	n/a
h2	96	95	95
jython	461	401	401
luindex	71	57	n/a
lusearch	78	78	78
pmd	217	197	175
sunflow	85	84	84
tomcat	271	271	270
tradebeans	2809	2807	2808
tradesoap	2799	2798	2798
xalan	204	204	203

Table 4: Number of phantom classes

for every benchmark configuration. First we can see that the number of phantom classes is by no means negligible; especially tradebeans and tradesoap statically reference many such "missing" classes. In addition, the numbers show that there is a slight tendency for larger workloads to result in fewer phantom classes: runs on larger inputs are likely to load classes that runs on smaller inputs fail to load (and therefore fail to dump). However, overall the impact of the input size on the number of phantom classes is only very small. This confirms the intuition that we formed above, i.e., that small inputs can already reach a long way.

6.4 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. It is therefore important to consider the runtime overhead that both agents incur. To quantify this runtime overhead, we used the DaCapo benchmark suite for what it was designed for: runtime-performance evaluation. The evaluation was done on 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: one without any agents (acting as a baseline), one with the Play-out Agent, and one with the Play-in Agent enabled. During each invocation, the benchmark performed two iterations of its default workload and we report the runtime of both iterations. Figure 5 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. One reason for this is that 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.

Figure 5 shows that TAMIFLEX incurs little overhead[8] 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 (cf. Table 5).

What makes logging these reflective calls expensive are the calls to `Thread.getStackTrace` that the Play-out Agent performs (cf. Section 4.2.1). Although TAMIFLEX requires only the topmost stack frames, the calls construct a representation of the call stack in its entirety.[9] Table 5 shows the total number of reflective calls encountered and the average depth of the call stack at the reflective calls. In the case of tradesoap, `Thread.getStackTrace` constructs more than

[8] We could not measure the runtime nor average stack height of tomcat because of a known infinite recursion in tomcat that causes the call stack to grow indefinitely until the respective thread dies. (For more info see tracker item 2934521 in DaCapo's bug tracker.) On our benchmarking machine, this error causes the virtual machine to signal a segmentation fault when one of our agents is enabled. We suspect that the virtual machine does not handle the error correctly. All the other numbers that we do report for tomcat were taken on a different machine on which the error causes no segmentation fault. Unfortunately we could not use this other machine to measure runtimes because it is a time-sharing environment unsuitable for performance evaluation. Even on this machine, an average stack height is not meaningful when a stack overflow occurs.

[9] At the expense of portability, one could improve performance by using JVMTI [25] to selectively access the stack.

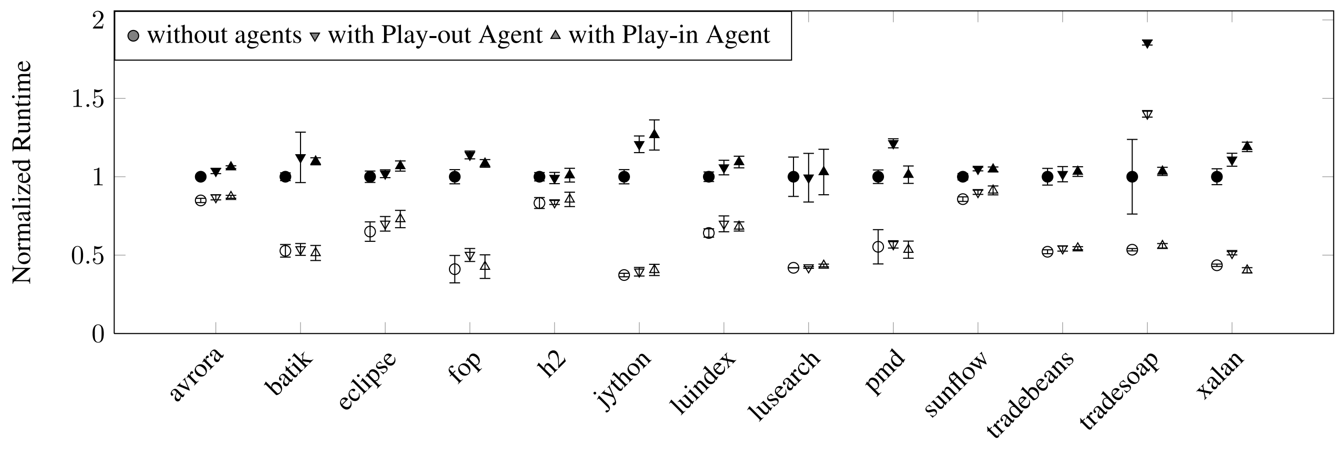

Figure 5: Normalized runtime[8] (arithmetic mean ± standard deviation) on the programs from the DaCapo benchmark suite (size: default) for the first (■) and second (□) iteration.

	Number	Stack height
avrora	38	14.68
batik	212	39.95
eclipse	2730	42.58
fop	3449	50.53
h2	68	22.38
jython	1511	76.11
luindex	90	16.96
lusearch	312	11.08
pmd	29021	29.60
sunflow	88	29.18
tomcat	17594	n/a[8]
tradebeans	51268	58.09
tradesoap	684486	32.17
xalan	27633	15.43

Table 5: Number of reflective calls and average stack height at these calls (size: default)

	Number	Volume [MiB]
avrora	1039	5.71
batik	2129	12.25
eclipse	3157	23.90
fop	1848	11.42
h2	1006	6.34
jython	3026	18.32
luindex	782	4.72
lusearch	729	4.33
pmd	1520	9.17
sunflow	965	6.02
tomcat	2752	18.02
tradebeans	8078	51.38
tradesoap	8207	51.99
xalan	1264	7.50

Table 6: Classes dumped by Play-out Agent (size: default)

22 million stack frames for the 684486 reflective calls made, at an average stack height of 32.17, during a single iteration of the benchmark. The majority of the calls target the accessor methods of a single class, the `DataHoldingBean`. 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%.

The Play-out Agent incurs runtime overhead not only when logging reflective calls but also when dumping class files. Figure 5 does not account for this overhead, as the agent keeps the class files in memory and only dumps them when the VM is about to shut down, i.e., after the benchmark itself has finished and has reported its runtime. The time that it takes to dump the class files is, however, negligible in practice. In Table 6 we show the amount of classes that the Dumper dumps for each benchmark run. Dumping the 8207 class files loaded when running tradesoap amounts to 51.99 MiB of data and prolongs the runtime of the overall benchmark invocation by a mere 1.7 seconds.

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.

Runtime and memory requirements of Soot The only worrisome result that we could find was the increased runtime and memory consumption of Soot and Spark when producing call graphs based on TamiFlex trace files. The problem is that, for programs like the DaCapo benchmarks that use reflection and custom class loaders heavily, Soot creates much larger call graphs when used with TamiFlex

	small	default	large
avrora	1:50	1:52	2:05
batik	12:10	12:10	12:46
eclipse	11:34	18:18	17:51
fop	24:47	23:36	n/a
h2	2:42	2:43	3:06
jython	41:59	1:01:47	1:00:54
luindex	1:53	1:52	n/a
lusearch	1:48	2:04	2:01
pmd	2:51	2:51	2:56
sunflow	7:00	6:42	6:47
tomcat	23:57	23:41	25:70
tradebeans	1:13:56	1:12:00	1:12:42
tradesoap	1:15:40	1:13:41	1:15:32
xalan	3:08	2:50	2:47

Table 7: Runtimes of Soot in TAMIFLEX mode (h:mm:ss)

than without. This is actually a good thing: large portions of the call graph that Soot would miss without TAMIFLEX are likely to be reached at runtime. These call graphs would therefore be unsound. On the other hand, creating and processing larger call graphs (and pointer-assignment graphs) requires more time and memory. Table 7 shows the runtime of Soot and Spark in TAMIFLEX mode, i.e., including the construction of a call graph and points-to sets and including the time that it takes to write transformed classes back to disk. (In our case the classes were not actually transformed but they could have been had we enabled any of the optimizations in Soot.) We ran Soot on a compute server with 12 CPUs of type Quad-Core AMD Opteron 8356, clocked at 2.3GhZ. (Note though, that Soot is single-threaded). As virtual machine we used the 64bit Linux version of IBM's J9 1.6.0 SR7 because Sun's HotSpot is known to have problems with large heap sizes: HotSpot frequently leads to segmentation faults when used with Spark, likely caused by a programming error in the virtual machine itself, as Spark is implemented in pure Java. We did not measure the memory requirements of Soot explicitly but 6GB were insufficient to run Soot on all benchmark configurations while 10GB sufficed. It appears that memory-efficient representations of call graphs and pointer-assignment graphs like the ones in Paddle [28] and bddbddb [41] gain importance through these results.

6.5 Summary

To summarize our experimental evaluation, we have shown that programmers can use TAMIFLEX in combination with Spark to obtain call graphs that are sound with respect to all program runs that TAMIFLEX recorded. In addition, we showed that the quality of the log files that TAMIFLEX produces does not depend much on the code coverage of the programs in our benchmark suite: even on small inputs, the benchmark runs will often visit most reflective call sites, causing TAMIFLEX to record information about these sites. Similarly, the input size has no large impact on the the number of classes that Soot needs to model as phantom classes, although there is a slight tendency that better code coverage yields less phantom classes. The runtime overhead of TAMIFLEX's agents is generally low. In particular, there appears to be no perceivable overhead after the first iteration of any of the DaCapo benchmarks. The only unfavourable side effect of using TAMIFLEX appears to be that constructing sound call graphs and points-to sets based on TAMIFLEX log files requires a significant amount of computation time and memory, but this is just due to the nature of these analyses and nothing that TAMIFLEX itself could address.

6.6 Threats to validity

The internal validity of our experiments is high. In general, 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 could only find one minor problem with this approach. To perform comparisons between the static call graphs computed with Soot and TAMIFLEX and those generated by the JVMTI agent (cf. Section 6.1), we use the PROBE call-graph differencing tool [26]. This tool, however, is unable to distinguish between signatures that only differ in their return type. Due to this limitation, PROBE combines these methods in its internal call-graph model, although our JVMTI agent and TAMIFLEX can tell the methods apart. But this problem, which did not exist before Java 5 introduced methods with covariant return types, is negligible for two reasons: First, at most two dozen methods found in the generated log files have a covariant return type. Second, all these methods are synthetic "bridge" methods [20, Section 15.12.4.5] which simply call the method they overwrite. As PROBE treats these two methods as equal, it treats the call to the overwritten method as a recursive call. While strictly speaking incorrect, this does not affect reachability in the call-graph at large. Both Soot and TAMIFLEX correctly distinguish methods with covariant return types.

The external validity of our experiments is threatened by our choice of benchmarks programs, and in particular by the chosen benchmark runs. It appears widely accepted that the DaCapo benchmarks give an accurate picture of Java applications with industrial relevance. Nevertheless, we argue about the stability of TAMIFLEX's log files only by comparing three runs (small/default/large) for each benchmark. This is unproblematic in the scenario where researchers use TAMIFLEX and Soot to analyze benchmarks such as DaCapo only. However, further experiments are required to show that TAMIFLEX can just as effectively be used in a broader context where many more program runs would need to be considered. In the near future, we plan to evaluate TAMIFLEX in combination with test-coverage tools.

7. Related Work

We compare our work with online analyses, Bruno Dufour's "blended" analysis, static analyses for resolving reflection, approaches for partial evaluation, Tatsubori's approach on enforced meta-programming layers and PRuby, a hybrid dynamic/static approach to type inference. We also refer to an article on the static-analysis tool Coverity, in which the authors describe problems similar to the ones that TAMIFLEX solves. Coverity solves these problems through tracing on the operating-system level.

Online analysis Hirzel et al. [23, 24] 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. Andersen's points-to-analysis algorithm consists of two phases: finding constraints that model the program's semantics, and propagating these constraints until reaching a fixed point. An online approach requires multiple extensions to the first part: constraints cannot be computed in a single pass but rather have to be discovered during program execution. An online algorithm must therefore use abstractions that allow for such online updates. Hirzel et al. solve this problem using a specialized constraint graph.

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, e.g., assume that call graphs and points-to sets soundly model all possible executions. It appears non-trivial to convert such algorithms so 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 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 complete for all possible executions. Also, Hirzel's et al.'s approach is bound to Jikes, while TAMIFLEX can be used with any Java 6 compliant virtual machine.

Blended analysis Dufour [15] 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 has 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. This is similar to TAMIFLEX's Play-out Agent, although in a more specialized setting. The authors' analysis is deliberately unsound, as it aims to find the performance problems present in a given program run. The authors do not address the problem that TAMIFLEX's Play-in Agent addresses: re-inserting offline-transformed classes into the original system. Therefore, the author's approach also does not need to normalize randomized class files in the way TAMIFLEX does.

Reflection analysis Livshits, Whaley and Lam [4] 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.) As we discussed in Section 5, Spark uses the same mechanism for calls to `Class.forName` but not for calls to `Method.invoke`. The author's approach further analyzes type casts to narrow down the possible type of objects created by `Class.newInstance`: assuming that casts do not fail, an expression such as `o = (C) Class.newInstance(..)` will always assign an object of type `C` (or a subtype) to `o`. However, existing pointer analysis frameworks like Spark implement this idea already: Spark will narrow down the possible type of `o` to any subtype of `C` for any expression of the form `o = (C) <exp>`, irrespective of the use of reflection. The author's approach assumes that all class files that the program may load at runtime are accessible offline. We do not make this assumption; instead TAMIFLEX's Play-out Agent gathers these class files for us.

Partial evaluation Similar to Livshits et al., Braux and Noyé [10] propose a static approach to handling reflection at compile-time. The author's solution propagates type information through the program's abstract syntax tree. In some cases this information may be sufficient to substitute dynamically loaded classes by concrete types and calls to the reflection API by concrete method calls. However, unlike TAMIFLEX, the authors focus on lowering the runtime overhead involved with creating and manipulating reflective objects of type `Class`, `Method` or `Field`. For instance, their approach can replace the code from Figure 6a by the one in Figure 6b if the type of `anObj` can be inferred and happens to declare fields `x` and `y`.

Enforced meta-programming layers Tatsubori [38] introduces "Enforced meta-programming layers" (EMPL), a runtime notification mechanism for reflective calls. Tatsubori identified a problem similar to the static-analysis problem that we described: application middle-ware layers may use a custom class loader `l` to re-write, at load time, calls to a certain method `m()`. But when the application happens

```
Field []   fields  = anObj.getClass (). getFields ();
for( int  i=0;  i<fields . length ;  i++) {
    System.out. println ( fields [ i ]. getName() +
      ":  "+  fields [ i ]. get(anObj)  );
}
```

(a) Code using reflection

```
System.out. println (”x:  ”+anObj.x);
System.out. println (”y:  ”+anObj.y);
```

(b) Replaced code without reflection

Figure 6: Code replacement through partial evaluation

to call m through reflection or happens to load classes that call m through a class loader that is not a child class loader of l then there is no way for l to detect and rewrite these calls. Tatsubori therefore introduces EMPL as a mechanism to systematically discover such calls nonetheless, and that way make them available to rewriting class loaders. At application start-up time, EMPL rewrites the application's class loaders and the reflection interface of the Java runtime library such that these classes notify callbacks within EMPL of any loaded class and any reflective method call. EMPL then provides programmers with a meta-programming interface to handle such events in a uniform way.

EMPL's intent is similar to that of TAMIFLEX: making program analyses and transformations aware of reflection and dynamically loaded classes. EMPL however targets program transformers that rewrite classes at load time, while TAMIFLEX targets static program analyses. It also appears that some of the problems that Tatsubori describes were already solved through the `java.lang.instrument` API that we use for TAMIFLEX. For instance, using this API, programmers can rewrite every class, no matter which class loader loads it. When Tatsubori published his work in March 2004, this API did not yet exist.

PRuby With PRuby [19], Furr et al. propose a static-type inference system for the Ruby programming language. PRuby's analysis consists of three steps. The first step uses run-time instrumentation to gather application specific run-time profiles. The second step replaces dynamic features with statically analyzable alternatives, based on the profile. Importantly, PRuby adds instrumentation to safely handle cases when subsequent runs do not match the profile. In a last step, DRuby performs static type inference on the transformed code to enforce type safety.

Coverity In a recent article [6], the makers of Coverity [12] explain some of the difficulties that they encountered from making a research tool for bug finding ready for the market. One of the problems they mention:

> "Law: You can't check code you don't see. It seems too trite to note that checking code requires first finding it... until you try to do so consistently on many large code bases. Probably the most reliable way to check a system is to grab its code during the build process; the build system knows exactly which files are included in the system and how to compile them. This seems like a simple task. Unfortunately, it's often difficult to understand what an ad hoc, homegrown build system is doing well enough to extract this information, a difficulty compounded by the near-universal absolute edict: 'No, you can't touch that.' By default, companies refuse to let an external force modify anything; you cannot modify their compiler path, their broken makefiles (if they have any), or in any way write or reconfigure anything other than your own temporary files. Which is fine, since if you need to modify it, you most likely won't understand it."

The Coverity group eventually solved this problem (in a C-based setting) by intercepting system calls during the program's build process. In TAMIFLEX, the Play-out Agent solves the same problem in a Java-based setting: it writes all loaded code into a flat directory.

8. Conclusion

During the past decade, the notion of a realistic Java program has changed. Industrial Java applications have enormously grown in size and therefore researchers have proposed many new algorithms to improve the scalability of static analyses. However, few researchers have considered the problems that reflection and custom class loaders pose for static analyses. While ten years ago there may have been few Java programs that use these dynamic features, today's industrial Java applications use reflection and custom class loaders as the rule, not the exception, improve maintenance, integrate with other tools and frameworks, or to facilitate distributed computing.

In this work we have presented TAMIFLEX, an integrated solution to taming reflection in static analysis. For the first time, TAMIFLEX allows researchers to automatically construct sound call graphs and points-to information for 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 statically and re-insert the offline-transformed classes into the original application on the fly.

We have proven the feasibility of our approach by applying it to version 9.12-bach of the DaCapo benchmark suite, a realistic cross-section of the current state of the art in Java programming. Our results show that researchers can effectively use TAMIFLEX to create sound call graphs for all DaCapo benchmarks, despite their use of reflection, custom class loaders and dynamic class generation.

Acknowledgments

We thank the DaCapo developers for all the work involved with developing and continuously maintaining and supporting 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. Much gratitude also goes to Syed Albiz and Patrick Lam for helping modify Soot to correctly handle dormant code. Last but not least we wish to thank Ondřej Lhoták for providing and supporting PROBE. This call-graph viewing and differencing tool proved enormously useful. This work was supported by the Center for Advanced Security Research Darmstadt (www.cased.de).

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. ISSN 0018-8670.

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

[3] B. Bellamy, P. Avgustinov, O. de Moor, and D. Sereni. Efficient local type inference. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 475–492. ACM, 2008. ISBN 978-1-60558-215-3.

[4] J. W. Benjamin Livshits and M. S. Lam. Reflection analysis for Java. In *Third Asian Symposium on Programming Languages and Systems*, November 2005.

[5] W. C. Benton and C. N. Fischer. Interactive, scalable, declarative program analysis: from prototype to implementation. In *PPDP '07: Proceedings of the 9th ACM SIGPLAN international conference on Principles and practice of declarative programming*, pages 13–24. ACM, 2007. ISBN 978-1-59593-769-8.

[6] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM (CACM)*, 53(2): 66–75, 2010. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1646353.1646374.

[7] 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*, pages 169–190, Oct. 2006.

[8] E. Bodden, P. Lam, and L. Hendren. Object representatives: a uniform abstraction for pointer information. In *Visions of Computer Science - BCS International Academic Conference*. British Computing Society, Sept. 2008.

[9] E. Bodden, P. Lam, and L. Hendren. Finding Programming Errors Earlier by Evaluating Runtime Monitors Ahead-of-Time. In *FSE*, pages 36–47, Nov. 2008.

[10] M. Braux and J. Noyé. Towards partially evaluating reflection in java. In *PEPM '00: Proceedings of the 2000 ACM SIGPLAN workshop on Partial Evaluation and semantics-based Program Manipulation*, pages 2–11, New York, NY, USA, 1999. ACM.

[11] É. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. In *Proceedings of the ASF (ACM SIGOPS France) Journées Composants 2002 : Systèmes à composants adaptables et extensibles (Adaptable and extensible component systems)*, 2002.

[12] Coverity. Coverity static-analysis tool. `http://www.coverity.com/`.

[13] DaCapoWebsite. Dacapo website. `http://dacapobench.org/`.

[14] DayTrader. Apache DayTrader benchmark. `http://cwiki.apache.org/GMOxDOC20/daytrader.html`.

[15] B. Dufour, B. G. Ryder, and G. Sevitsky. Blended analysis for performance understanding of framework-based applications. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 118–128, New York, NY, USA, 2007. ACM.

[16] M. B. Dwyer and R. Purandare. Residual dynamic typestate analysis: Exploiting static analysis results to reformulate and reduce the cost of dynamic analysis. In *ASE*, pages 124–133, May 2007.

[17] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *PLDI*, pages 242–256, 1994.

[18] S. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. In *ISSTA*, pages 133–144, July 2006.

[19] M. Furr, J. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*. ACM, Oct. 2009.

[20] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java(TM) Language Specification*. Addison-Wesley Professional, 3rd edition, 2005. ISBN 0321246780.

[21] M. W. Hall and K. Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(3):227–242, 1992. ISSN 1057-4514.

[22] R. Halpert, C. Pickett, and C. Verbrugge. Component-based lock allocation. In *PACT*, pages 353–364. IEEE Computer Society, September 2007.

[23] M. Hirzel, A. Diwan, M. Hind, M. Hirzel, A. Diwan, and M. Hind. Pointer analysis in the presence of dynamic class loading. In *ECOOP*, pages 96–122, 2004.

[24] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *TOPLAS*, 29(2):11, 2007. ISSN 0164-0925.

[25] JVMTI. Java Virtual Machine Tool Interface (JVM TI). Version 6. `http://java.sun.com/javase/6/docs/technotes/guides/jvmti/`.

[26] O. Lhoták. Comparing call graphs. In *PASTE*, pages 37–42, New York, NY, USA, 2007. ACM.

[27] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. In *CC*, volume 2622 of *LNCS*, pages 153–169, Apr. 2003.

[28] O. Lhoták and L. Hendren. Evaluating the benefits of context-sensitive points-to analysis using a bdd-based implementation. *TOSEM*, 18(1):1–53, 2008. ISSN 1049-331X.

[29] G. C. Murphy, D. Notkin, W. G. Griswold, and E. S. Lan. An empirical study of static call graph extractors. *TOSEM*, 7(2): 158–191, 1998. ISSN 1049-331X.

[30] A. C. Myers. JFlow: practical mostly-static information flow control. In *POPL*, pages 228–241, New York, NY, USA, 1999. ACM.

[31] N. A. Naeem and O. Lhoták. Typestate-like analysis of multiple interacting objects. In *OOPSLA*, pages 347–366, Oct. 2008.

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

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

[34] S. Sagiv, T. W. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *TOPLAS*, 20(1):1–50, 1998.

[35] ShutdownHooks. Shutdown hooks API, Sun Microsystems. `http://java.sun.com/j2se/1.5.0/docs/guide/lang/hook-design.html`.

[36] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, pages 387–400, June 2006.

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

[38] M. Tatsubori. Living with reflection. In *6th JSSST Workshop on Programming and Programming Languages, Gamagohri, Aichi, Japan*, Mar. 2004.

[39] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *CASCON*, page 13. IBM Press, 1999.

[40] J. Venn. On the diagrammatic and mechanical representation of propositions and reasonings. *Philosophical Magazine and Journal of Science (5th series)*, 9(59):1–18, July 1880.

[41] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144, New York, NY, USA, 2004. ACM.

[42] G. Xu and A. Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *ISSTA*, pages 225–236, New York, NY, USA, 2008. ACM.