Reflection Analysis for Java

Benjamin Livshits, John Whaley, and Monica S. Lam[⋆]

Computer Science Department
Stanford University
Stanford, CA 94305, USA
{livshits, jwhaley, lam}@cs.stanford.edu

Abstract. Reflection has always been a thorn in the side of Java static analysis tools. Without a full treatment of reflection, static analysis tools are both *incomplete* because some parts of the program may not be included in the application call graph, and *unsound* because the static analysis does not take into account reflective features of Java that allow writes to object fields and method invocations. However, accurately analyzing reflection has always been difficult, leading to most static analysis tools treating reflection in an unsound manner or just ignoring it entirely. This is unsatisfactory as many modern Java applications make significant use of reflection.

In this paper we propose a static analysis algorithm that uses points-to information to approximate the targets of reflective calls as part of call graph construction. Because reflective calls may rely on input to the application, in addition to performing reflection resolution, our algorithm also discovers all places in the program where user-provided specifications are necessary to fully resolve reflective targets. As an alternative to user-provided specifications, we also propose a reflection resolution approach based on type cast information that reduces the need for user input, but typically results in a less precise call graph.

We have implemented the reflection resolution algorithms described in this paper and applied them to a set of six large, widely-used benchmark applications consisting of more than 600,000 lines of code combined. Experiments show that our technique is effective for resolving most reflective calls without any user input. Certain reflective calls, however, cannot be resolved at compile time precisely. Relying on a user-provided specification to obtain a conservative call graph results in graphs that contain 1.43 to 6.58 times more methods that the original. In one case, a conservative call graph has 7,047 more methods than a call graph that does not interpret reflective calls. In contrast, ignoring reflection leads to missing substantial portions of the application call graph.

1 Introduction

Whole-program static analysis requires knowing the targets of function or method calls. The task of computing a program's call graph is complicated for a language like Java because of virtual method invocations and reflection. Past research has addressed the analysis of function pointers in C as well as virtual method calls in C++ and Java. Reflection, however, has mostly been neglected.

[⋆] This work was supported by NSF Grant No. 0326227 and an Intel Graduate Fellowship.

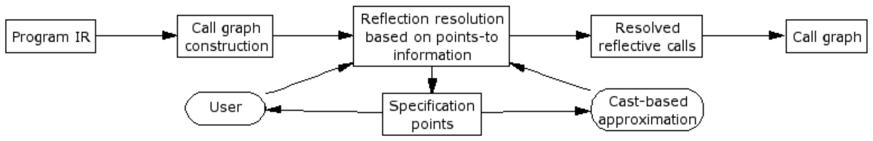

Fig. 1: Architecture of our static analysis framework.

Reflection in Java allows the developer to perform runtime actions given the descriptions of the objects involved: one can create objects given their class names, call methods by their name, and access object fields given their name [1]. Because names of methods to be invoked can be supplied by the user, especially in the presence of dynamic class loading, precise static construction of a call graph is generally undecidable. Even if we assume that all classes that may be used are available for analysis, without placing *any restrictions* of the targets of reflective calls, a sound (or conservative) call graph would be prohibitively large.

Many projects that use static analysis for optimization, error detection, and other purposes ignore the use of reflection, which makes static analysis tools *incomplete* because some parts of the program may not be included in the call graph and potentially *unsound*, because some operations, such as reflectively invoking a method or setting an object field, are ignored. Our research is motivated by the practical need to improve the coverage of static error detection tools [2–4]. The success of such tools in Java is predicated upon having a call graph available to the error detection tool. Unless reflective calls are interpreted, the tools run the danger of only analyzing a small portion of the available code and giving the developer a false sense of security when no bugs are reported. Moreover, when static results are used to reduce runtime instrumentation, all parts of the application that are used at runtime *must* be statically analyzed.

A recent paper by Hirzel, Diwan, and Hind proposes the use of dynamic instrumentation to collect the reflection targets discovered at run time [5]. They use this information to extend Andersen's context-insensitive, inclusion-based pointer analysis for Java into an online algorithm [6]. Reflective calls are generally used to offer a choice in the application control flow, and a dynamic application run typically includes only several of all the possibilities. However, analyses used for static error detection and optimization often require a *full* call graph of the program in order to achieve complete coverage.

In this paper we present a static analysis algorithm that uses points-to information to determine the targets of reflective calls. Often the targets of reflective calls can be determined precisely by analyzing the flow of strings that represent class names throughout the program. This allows us to precisely resolve many reflective calls and add them to the call graph. However, in some cases reflective call targets may depend on user input and require user-provided specifications for the call graph to be determined. Our algorithm determines all *specification points* — places in the program where user-provided specification is needed to determine reflective targets. The user is given the option to provide a specification and our call graph is complete with respect to the specifications provided [7].

Because providing reflection specifications can be time-consuming and error-prone, we also provide a conservative, albeit sometimes imprecise, approximation of targets of reflective calls by analyzing how type casts are used in the program. A common coding idiom consists of casting the result of a call to `Class.newInstance` used to create new objects to a more specific type before the returned object can be used. Relying on cast information allows us to produce a conservative call graph approximation without requiring user-provided reflection specifications in most cases. A flow diagram summarizing the stages of our analysis is shown in Figure 1.

Our reflection resolution approach hinges on three assumptions about the use of reflection: (a) all the class files that may be accessed at runtime are available for analysis; (b) the behavior of `Class.forName` is consistent with its API definition in that it returns a class whose name is specified by the first parameter, and (c) cast operations that operate on the results of `Class.newInstance` calls are correct. In rare cases when no cast information is available to aid with reflection resolution, we report this back to the user as a situation requiring specification.

1.1 Contributions

This paper makes the following contributions:

- We formulate a set of natural assumptions that hold in most Java applications and make the use of reflection amenable to static analysis.
- We propose a call graph construction algorithm that uses points-to information about strings used in reflective calls to statically find potential call targets. When reflective calls cannot be fully "resolved" at compile time, our algorithms determines a set of specification points — places in the program that require user-provided specification to resolve reflective calls.
- As an alternative to having to provide a reflection specification, we propose an algorithm that uses information about type casts in the program to statically approximate potential targets of reflective calls.
- We provide an extensive experimental evaluation of our analysis approach based on points-to results by applying it to a suite of six large open-source Java applications consisting of more than 600,000 lines of code combined. We evaluate how the points-to and cast-based analyses of reflective calls compare to a local intra-method approach. While all these analyses find at least one constant target for most `Class.forName` call sites, they only moderately increase the call graph size. However, the conservative call graph obtained with the help of a user-provided specification results is a call graph than is almost 7 times as big as the original. We assess the amount of effort required to come up with a specification and how cast-based information can significantly reduce the specification burden placed on the user.

1.2 Paper Organization

The rest of the paper is organized as follows. In Section 2, we provide background information about the use of reflection in Java. In Section 3, we lay out the simplifying assumptions made by our static analysis. In Sections 4 we describe our analysis approach. Section 5 provides a comprehensive experimental evaluation. Finally, in Sections 6 and 7 we describe related work and conclude.

2 Overview of Reflection in Java

In this section we first informally introduce the reflection APIs in Java. The most typical use of reflection by far is for creating new objects given the object class name. The most common usage idiom for reflectively creating an object is shown in Figure 2. Reflective APIs in Java are used for object creation, method invocation, and field access, as described below. Because of the space limitations, in this section we only briefly outline the relevant reflective APIs. Interested readers are encouraged to refer to our technical report for a complete treatment and a case study of reflection uses in our benchmarks applications [8].

Object Creation Object creation APIs in Java provide a way to programmatically create objects of a class, whose name is provided at runtime; parameters of the object constructor can be passed in as necessary. Obtaining a class given its name is most typically done using a call to one of the static functions `Class.forName(String, ...)` and passing the class name as the first parameter. We should point out that while `Class.forName` is the most common way to obtain a class given its name, it may not be the only method for doing so. An application may define a native method that implements the same functionality. The same observation applies to other standard reflective API methods.

The commonly used Java idiom `T.class`, where `T` is a class is translated by the compiler to a call to `Class.forName(T.getName())`. Since our reflection resolution algorithm works at the byte code level, `T.class` constructs do not require a special treatment. Creating an object with an empty constructor is achieved through a call to `newInstance` on the appropriate `java.lang.Class` object, which provides a runtime representation of a class.

Method Invocation Methods are obtained from a `Class` object by supplying the method signature or by iterating through the array of `Methods` returned by one of `Class` functions. `Methods` are subsequently invoked by calling `Method.invoke`.

Accessing Fields Fields of Java runtime objects can be read and written at runtime. Calls to `Field.get` and `Field.set` can be used to get and set fields containing objects. Additional methods are provided for fields of primitive types.

3 Assumptions About Reflection

This section presents assumptions we make in our static analysis for resolving reflection in Java programs. We believe that these assumptions are quite reasonable and hold for many real-life Java applications.

The problem of precisely determining the classes that an application may access is undecidable. Furthermore, for applications that access the network,

```
String className = ...;
Class c  = Class.forName(className);
Object o = c.newInstance();
T      t = (T) o;
```

Fig. 2: Typical use of reflection to create new objects.

the set of classes that may be accessed is *unbounded*: we cannot possibly hope to analyze all classes that the application may conceivably download from the net and load at runtime. Programs can also dynamically generate classes to be subsequently loaded. Our analysis assumes a closed world, as defined below.

Assumption 1. Closed world.
We assume that only classes reachable from the class path at analysis time can be used by the application at runtime.

In the presence of user-defined class loaders, it is impossible to statically determine the behavior of function `Class.forName`. If custom class loaders are used, the behavior of `Class.forName` can change; it is even possible for a malicious class loader to return completely unrelated classes in response to a `Class.forName` call. The following assumption allows us to interpret calls to `Class.forName`.

Assumption 2. Well-behaved class loaders.
The name of the class returned by a call to `Class.forName(className)` *equals* `className`.

To check the validity of Assumption 2, we have instrumented large applications to observe the behavior of `Class.forName`; we have never encountered a violation of this assumption. Finally, we introduce the following assumption that allows us to leverage type cast information contained in the program to constrain the targets of reflective calls.

Assumption 3. Correct casts.
Type cast operations that always operate on the result of a call to `newInstance` *are correct; they will always succeed without throwing a* `ClassCastException`.

We believe this to be a valid practical assumption: while it is possible to have casts that fail, causing an exception that is caught so that the instantiated object can be used afterwards, we have not seen such cases in practice. Typical `catch` blocks around such casts lead to the program terminating with an error message.

4 Analysis of Reflection

In this section, we present techniques for resolving reflective calls in a program. Our analysis consists of the following three steps:

1. We use a sound points-to analysis to determine all the possible sources of strings that are used as class names. Such sources can either be constant strings or derived from external sources. The pointer analysis-based approach *fully resolves* the targets of a reflective call if constant strings account for all the possible sources. We say that a call is *partially resolved* if the sources can be either constants or inputs and *unresolved* if the sources can only be inputs. Knowing which external sources may be used as class names is useful because users can potentially specify all the possible values; typical examples are return results of file read operations. We refer to program points where the input strings are defined as *specification points*.

2. Unfortunately the number of specification points in a program can be large. Instead of asking users to specify the values of every possible input string, our second technique takes advantage of casts, whenever available, to determine a conservative approximation of targets of reflective calls *that are not fully resolved*. For example, as shown in Figure 2, the call to `Class.newInstance`, which returns an `Object`, is always followed by a cast to the appropriate type before the newly created object can be used. Assuming no exception is raised, we can conclude that the new object must be a subtype of the type used in the cast, thus restricting the set of objects that may be instantiated.
3. Finally, we rely on user-provided specification for the remaining set of calls — namely calls whose source strings are not all constants — in order to obtain a conservative approximation of the call graph.

We start by describing the call graph discovery algorithm in Section 4.1 as well as how reflection resolution fits in with call graph discovery. Section 4.2 presents a reflection resolution algorithm based on pointer analysis results. Finally, Section 4.3 describes our algorithm that leverages type cast information for conservative call graph construction without relying on user-provided specifications.

4.1 Call Graph Discovery

Our static techniques to discover reflective targets are integrated into a context-insensitive points-to analysis that discovers the call graph on the fly [9]. As the points-to analysis finds the pointees of variables, type information of these pointees is used to resolve the targets of virtual method invocations, increasing the size of the call graph, which in turn is used to find more pointees. Our analysis of reflective calls further expands the call graph, which is used in the analysis to generate more points-to relations, leading to bigger call graphs. The discovery algorithm terminates when a fixpoint is reached and no more call targets or points-to relations can be found.

By using a points-to analysis to discover the call graph, we can obtain a more accurate call graph than by using a less precise technique such as class hierarchy analysis CHA [10] or rapid type analysis RTA [11]. We use a context-insensitive version of the analysis because context sensitivity does not seem to substantially improve the accuracy of the call graph [9, 12].

4.2 Pointer Analysis for Reflection

This section describes how we leverage pointer analysis results to resolve calls to `Class.forName` and track `Class` objects. This can be used to discover the types of objects that can be created at calls to `Class.newInstance`, along with resolving reflective method invocations and field access operations. Pointer analysis is also used to find specification points: external sources that propagate string values to the first argument of `Class.forName`.

Reflection and Points-to Information The programming idiom that motivated the use of points-to analysis for resolving reflection was first presented in Figure 2. This idiom consists of the following steps:

1. Obtain the name of the class for the object that needs to be created.
2. Create a `Class` object by calling the static method `Class.forName`.
3. Create the new object with a call to `Class.newInstance`.
4. Cast the result of the call to `Class.newInstance` to the necessary type in order to use the newly created object.

When interpreting this idiom statically, we would like to "resolve" the call to `Class.newInstance` in step 3 as a call to the default constructor `T()`. However, analyzing even this relatively simple idiom is nontrivial.

The four steps shown above can be widely separated in the code and reside in different methods, classes, or jar libraries. The `Class` object obtained in step 2 may be passed through several levels of function calls before being used in step 3. Furthermore, the `Class` object can be deposited in a collection to be later retrieved in step 3. The same is true for the name of the class created in step 1 and used later in step 2. To determine how variables `className`, `c`, `o`, and `t` defined and used in steps 1–4 may be related, we need to know what runtime objects they may be referring to: a problem addressed by *points-to* analysis. Point-to analysis computes which objects each program variable may refer to.

Resolution of `Class.newInstance` of `Class.forName` calls is not the only thing made possible with points-to results: using points-to analysis, we also track `Method`, `Field`, and `Constructor` objects. This allows us to correctly resolve reflective method invocations and field accesses. Reflection is also commonly used to invoke the class constructor of a given class via calling `Class.forName` with the class name as the first argument. We use points-to information to determine potential targets of `Class.forName` calls and add calls to class constructors of the appropriate classes to the call graph.

The `bddbddb` Program Database In the remainder of this section we describe how pointer information is used for reflection resolution. We start by describing how the input program can be represented as a set of relations in `bddbddb`, a BDD-based program database [9, 13]. The program database and the associated constraint resolution tool allows program analyses to be expressed in a succinct and natural fashion as a set of rules in Datalog, a logic programming language. Points-to information is compactly represented in `bddbddb` with binary decision diagrams (BDDs), and can be accessed and manipulated efficiently with Datalog queries. The program representation as well as pointer analysis results are stored as relations in the `bddbddb` database. The domains in the database include invocation sites I, variables V, methods M, heap objects named by their allocation site H, types T, and integers Z.

The source program is represented as a number of input relations. For instance, relations *actual* and *ret* represent parameter passing and method returns, respectively. In the following, we say that predicate $A(x_1, \ldots, x_n)$ is true if tuple $(x_1, \ldots, x_n)$ is in relation A. Below we show the definitions of Datalog relations used to represent the input program:

actual: $I \times Z \times V$. $\mathit{actual}(i, z, v)$ means that variable v is zth argument of the method call at i.

ret: $I \times V$. $\mathit{ret}(i, v)$, means that variable v is the return result of the method call at i.

assign: $V \times V$. $assign(v_1, v_2)$ means that there is an implicit or explicit assignment statement $v_1 = v_2$ in the program.
load: $V \times F \times V$. $load(v_1, f, v_2)$ means that there is a load statement $v_2 = v_1.f$ in the program.
store: $V \times F \times V$. $store(v_1, f, v_2)$ means that there is a store statement $v_1.f = v_2$ in the program.
string2class: $H \times T$. $string2class(s, t)$ means that string constant s is the string representation of the name of type t.
calls: $I \times M$ is the invocation relation. $calls(i, m)$ means that invocation site i may invoke method m.

Points-to results are represented with the relation *vP*:

vP: $V \times H$ is the variable points-to relation. $vP(v, h)$ means that variable v may point to heap object h.

A Datalog query consists of a set of rules, written in a Prolog-style notation, where a predicate is defined as a conjunction of other predicates. For example, the Datalog rule $D(w, z) :- A(w, x), B(x, y), C(y, z)$. says that "$D(w, z)$ is true if $A(w, x)$, $B(x, y)$, and $C(y, z)$ are all true."

Reflection Resolution Algorithm The algorithm for computing targets of reflective calls is naturally expressed in terms of Datalog queries. Below we define Datalog rules to resolve targets of `Class.newInstance` and `Class.forName` calls. Handling of constructors, methods, and fields proceed similarly.

To compute reflective targets of calls to `Class.newInstance`, we define two Datalog relations. Relation *classObjects* contains pairs $\langle i, t\rangle$ of invocations sites $i \in I$ calling `Class.forName` and types $t \in T$ that may be returned from the call. We define *classObjects* using the following Datalog rule:

$$\begin{aligned} classObjects(i, t) \ :- \ & calls(i, \text{“}\texttt{Class.forName}\text{”}), \\ & actual(i, 1, v), vP(v, s), string2class(s, t). \end{aligned}$$

The Datalog rule for *classObjects* reads as follows. Invocation site i returns an object of type t if the call graph relation *calls* contains an edge from i to "`Class.forName`", parameter 1 of i is v, v points to s, and s is a string that represents the name of type t.

Relation *newInstanceTargets* contains pairs $\langle i, t\rangle$ of invocation sites $i \in I$ calling `Class.newInstance` and classes $t \in T$ that may be reflectively invoked by the call. The Datalog rule to compute *newInstanceTargets* is:

$$\begin{aligned} newInstanceTargets(i, t) \ :- \ & calls(i, \text{“}\texttt{Class.newInstance}\text{”}), \\ & actual(i, 0, v), vP(v, c), \\ & vP(v_c, c), ret(i_c, v_c), classObjects(i_c, t). \end{aligned}$$

The rule reads as follows. Invocation site i returns a new object of type t if the call graph relation *calls* contains an edge from i to `Class.newInstance`, parameter 0 of i is v, v is aliased to a variable v_c that is the return value of invocation site i_c, and i_c returns type t. Targets of `Class.forName` calls are resolved and calls to the appropriate class constructors are added to the invocation relation *calls*:

$$calls(i, m) \ :- \ classObjects(i, t), m = t + \text{“}.<\texttt{clinit}>\text{”}.$$

```
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet4AddressImpl
loadImpl() @ 43 InetAddress.java:1231       => java.net.Inet6AddressImpl
...
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.ISO_8859_15
lookup() @ 86 AbstractCharsetProvider.java:126 => sun.nio.cs.MS1251
...
tryToLoadClass() @ 29 DataFlavor.java:64       => java.io.InputStream
...
```

Fig. 3: A fragment of a specification file accepted by our system. A string identifying a call site to `Class.forName` is mapped to a class name that that call may resolve to.

(The "+" sign indicates string concatenation.) Similarly, having computed relation *newInstanceTargets*(i, t), we add these reflective call targets invoking the appropriate type constructor to the call graph relation *calls* with the rule below:

$$\mathit{calls}(i, m) \ :-\ \mathit{newInstanceTargets}(i, t), m = t + \text{“}.<\texttt{init}>\text{”}.$$

Handling `Constructor` and Other Objects Another technique of reflective object creation is to use `Class.getConstructor` to get a `Constructor` object, and then calling `newInstance` on that. We define a relation *constructorTypes* that contains pairs $\langle i, t\rangle$ of invocations sites $i \in I$ calling `Class.getConstructor` and types $t \in T$ of the type of the constructor:

$$\begin{aligned}\mathit{constructorTypes}(i, t) \ :-\ & \mathit{calls}(i, \text{“}\texttt{Class.getConstructor}\text{”}),\\ & \mathit{actual}(i, 0, v), \mathit{vP}(v, h), \mathit{classObjects}(h, t).\end{aligned}$$

Once we have computed *constructorTypes*, we can compute more *newInstanceTargets* as follows:

$$\begin{aligned}\mathit{newInstanceTargets}(i, t) \ :-\ & \mathit{calls}(i, \text{“}\texttt{Class.newInstance}\text{”}),\\ & \mathit{actual}(i, 0, v), \mathit{vP}(v, c), \mathit{vP}(v_c, c), \mathit{ret}(i_c, v_c),\\ & \mathit{constructorTypes}(i_c, t).\end{aligned}$$

This rule says that invocation site i calling "`Class.newInstance`" returns an object of type t if parameter 0 of i is v, v is aliased to the return value of invocation i_c which calls "`Class.getConstructor`", and the call to i_c is on type t.

In a similar manner, we can add support for `Class.getConstructors`, along with support for reflective field, and method accesses. The specification of these are straightforward and we do not describe them here. Our actual implementation completely models all methods in the Java Reflection API. We refer the reader to a technical report we have for more details [8].

Specification Points and User-Provided Specifications Using a points-to analysis also allows us to determine, when a non-constant string is passed to a call to `Class.forName`, the *provenance* of that string. The *provenance* of a string is in essence a backward data slice showing the flow of data to that string. Provenance allows us to compute *specification points*—places in the program where external sources are read by the program from a configuration file, system properties, etc. For each specification point, the user can provide values that may be passed into the application.

We compute the provenance by propagating through the assignment relation *assign*, aliased loads and stores, and string operations. To make the specification points as close to external sources as possible, we perform a simple analysis of strings to do backward propagation through string concatenation operations. For brevity, we only list the `StringBuffer.append` method used by the Java compiler to expand string concatenation operations here; other string operations work in a similar manner. The following rules for relation *leadsToForName* detail provenance propagation:

$$\begin{array}{lll}
\mathit{leadsToForName}(v, i) & :- & \mathit{calls}(i, \text{“}\texttt{Class.forName}\text{”}), \mathit{actual}(i, 1, v).\\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{assign}(v_1, v_2).\\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i),\\
 & & \mathit{load}(v_3, f, v_1), \mathit{vP}(v_3, h_3), \mathit{vP}(v_4, h_3), \mathit{store}(v_4, f, v_2).\\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{ret}(i_2, v_1),\\
 & & \mathit{calls}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 0, v_2).\\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{ret}(i_2, v_1),\\
 & & \mathit{calls}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2).\\
\mathit{leadsToForName}(v_2, i) & :- & \mathit{leadsToForName}(v_1, i), \mathit{actual}(i_2, 0, v_1),\\
 & & \mathit{calls}(i_2, \text{“}\texttt{StringBuffer.append}\text{”}), \mathit{actual}(i_2, 1, v_2).
\end{array}$$

To compute the specification points necessary to resolve `Class.forName` calls, we find endpoints of the *leadsToForName* propagation chains that are *not* string constants that represent class names. These will often terminate in the return result of a call to `System.getProperty` in the case of reading from a system property or `BufferedReader.readLine` in the case of reading from a file. By specifying the possible values at that point that are appropriate for the application being analyzed, the user can construct a complete call graph.

Our implementation accepts specification files that contain a simple textual map of a specification point to the constant strings it can generate. A specification point is represented by a method name, bytecode offset, and the relevant line number. An example of a specification file is shown in Figure 3.

4.3 Reflection Resolution Using Casts

For some applications, the task of providing reflection specifications may be too heavy a burden. Fortunately, we can leverage the type cast information present in the program to automatically determine a conservative approximation of possible reflective targets. Consider, for instance, the following typical code snippet:

```
1.   Object o = c.newInstance();
2.   String s = (String) o;
```

The cast in statement 2 *post-dominates* the call to `Class.newInstance` in statement 1. This implies that all execution paths that pass through the call to `Class.newInstance` must also go through the cast in statement 2 [14]. For statement 2 not to produce a runtime exception, `o` must be a subclass of `String`. Thus, only subtypes of `String` can be created as a result of the call to `newInstance`.

More generally, if the result of a `newInstance` call is *always* cast to type t, we say that only subtypes of t can be instantiated at the call to `newInstance`.

Relying on cast operations can possibly be unsound as the cast may fail, in which case, the code will throw a `ClassCastException`. Thus, in order to work, our cast-based technique relies on Assumption 3, the correctness of cast operations.

Preparing Subtype Information We rely on the closed world Assumption 2 described in Section 3 to find the set of all classes possibly used by the application. The classes available at analysis time are generally distributed with the application. However, occasionally, there are classes that are generated when the application is compiled or deployed, typically with the help of an Ant script. Therefore, we generate the set of possible classes *after* deploying the application.

We pre-process all resulting classes to compute the subtyping relation $\mathit{subtype}(t_1, t_2)$ that determines when t_1 is a subtype of t_2. Preprocessing even the smallest applications involved looking at many thousands of classes because we consider all the default jars that the Java runtime system has access to. We run this preprocessing step off-line and store the results for easy access.

Using Cast Information We integrate the information about cast operations directly into the system of constraints expressed in Datalog. We use a Datalog relation *subtype* described above, a relation *cast* that holds the cast operations, and a relation *unresolved* that holds the unresolved calls to `Class.forName`. The following Datalog rule uses cast operations applied to the return result v_{ret} of a call i to `Class.newInstance` to constrain the possible types t_c of `Class` objects c returned from calls sites i_c of `Class.forName`:

$$\begin{aligned}\mathit{classObjects}(i_c, t) \ :- \ & \mathit{calls}(i, \text{“}\texttt{Class.newInstance}\text{”}), \mathit{actual}(i, 0, v), \mathit{vP}(v, c), \\ & \mathit{ret}(i, v_{ret}), \mathit{cast}(\_, t_c, v_{ret}), \mathit{subtype}(t, t_c), \\ & \mathit{unresolved}(i_c), \mathit{vP}(v_c, c), \mathit{ret}(i_c, v_c).\end{aligned}$$

Information propagates both forward and backward—for example, casting the result of a call to `Class.newInstance` constrains the `Class` object it is called upon. If the same `Class` object is used in another part of the program, the type constraint derived from the cast will be obeyed.

Problems with Using Casts Casts are sometimes inadequate for resolving calls to `Class.newInstance` for the following reasons. First, the cast-based approach is inherently imprecise because programs often cast the result of `Class.newInstance` to a very wide type such as `java.io.Serializable`. This produces a lot of *potential* subclasses, only some of which are relevant in practice. Second, as our experiments show, not all calls to `Class.newInstance` have post-dominating casts, as illustrated by the following example.

Example 1. As shown in Figure 4, one of our benchmark applications, `freetts`, places the object returned by `Class.newInstance` into a vector `voiceDirectories` (line 5). Despite the fact that the objects are subsequently cast to type `VoiceDirectory[]` on line 8, intraprocedural post-dominance is not powerful enough to take this cast into account. □

Using cast information significantly reduces the need for user-provided specification in practice. While the version of the analysis that does not use cast

```
1.      UniqueVector voiceDirectories = new UniqueVector();
2.      for (int i = 0; i < voiceDirectoryNames.size(); i++) {
3.          Class c = Class.forName((String) voiceDirectoryNames.get(i),
4.                                              true, classLoader);
5.          voiceDirectories.add(c.newInstance());
6.      }
7.
8.      return (VoiceDirectory[]) voiceDirectories.toArray(new
9.                      VoiceDirectory[voiceDirectories.size()]);
```

Fig. 4: A case in `freetts` where our analysis is unable to determine the type of objects instantiated on line 5 using casts.

information can be made fully sound with user specification as well, we chose to only provide a specification for the cast-based version.

5 Experimental Results

In this section we present a comprehensive experimental evaluation of the static analysis approaches presented in Section 4. In Section 5.1 we describe our experimental setup. Section 5.2 presents an overview our experimental results. Section 5.3 presents our baseline local reflection analysis. In Sections 5.4 and 5.5 we discuss the effectiveness of using the points-to and cast-based reflection resolution approaches, respectively. Section 5.6 describes the specifications needed to obtain a sound call graph approximation. Section 5.7 compares the overall sizes of the call graph for the different analysis versions presented in this section.

5.1 Experimental Setup

We performed our experiments on a suite of six large, widely-used open-source Java benchmark applications. These applications were selected among the most popular Java projects available on SourceForge. We believe that real-life applications like these are more representative of how programmers use reflection than synthetically created test suites, or SPEC JVM benchmarks, most of which avoid reflection altogether.

Benchmark	**Description**	**Line count**	**File count**	**Jars**	**Available classes**
`jgap`	genetic algorithms package	32,961	172	9	62,727
`freetts`	speech synthesis system	42,993	167	19	62,821
`gruntspud`	graphical CVS client	80,138	378	10	63,847
`jedit`	graphical text editor	144,496	427	1	62,910
`columba`	graphical email client	149,044	1,170	35	53,689
`jfreechart`	chart drawing library	193,396	707	6	62,885
Total		**643,028**	**3,021**	**80**	**368,879**

Fig. 5: Summary of information about our benchmarks. Applications are sorted by the number of lines of code in column 3.

	None	Local			Points-to				Casts				Sound		
Benchmark	**T**	**T**	**FR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**PR**	**UR**	**T**	**FR**	**UR**
`jgap`	27	27	19	8	28	20	1	7	28	20	4	4	89	85	4
`freetts`	30	30	21	9	30	21	0	9	34	25	4	5	81	75	6
`gruntspud`	139	139	112	27	142	115	5	22	232	191	19	22	220	208	12
`jedit`	156	156	137	19	161	142	3	16	178	159	12	7	210	197	12
`columba`	104	105	89	16	105	89	2	14	118	101	10	7	173	167	6
`jfreechart`	104	104	91	13	104	91	1	12	149	124	10	15	169	165	4

Fig. 6: Results of resolving `Class.forName` calls for different analysis versions.

Summary of information about the applications is provided in Figure 5. Notice that the traditional lines of code size metric is somewhat misleading in the case of applications that rely on large libraries. Many of these benchmarks depend of massive libraries, so, while the application code may be small, the full size of the application executed at runtime is quite large. The last column of the table in Figure 5 lists the number of classes available by the time each application is deployed, including those in the JDK.

We ran all of our experiments on an Opteron 150 machine equipped with 4GB or memory running Linux. JDK version 1.4.2_08 was used. All of the running times for our preliminary implementation were in tens of minutes, which, although a little high, is acceptable for programs of this size. Creating subtype information for use with cast-based analysis took well under a minute.

5.2 Evaluation Approach

We have implemented five different variations of our algorithms: None, Local, Points-to, Casts, and Sound and applied them to the benchmarks described above. None is the base version that performs no reflection resolution; Local performs a simple local analysis, as described in Section 5.3. Points-to and Casts are described in Sections 4.2 and 4.3, respectively.

Version Sound is augmented with a user-provided specification to make the answer conservative. We should point out that only the Sound version provides results that are fully sound: None essentially assumes that reflective calls have no targets. Local only handles reflective calls that can be fully resolved within a single method. Points-to and Casts only provide targets for reflective calls for which either string or cast information constraining the possible targets is available and unsoundly assumes that the rest of the calls have no targets.

Figure 6 summarizes the results of resolving `Class.forName` using all five analysis versions. `Class.forName` calls represent by far the most common kind of reflective operations and we focus on them in our experimental evaluation. To reiterate the definitions in Section 4, we distinguish between:

– *fully resolved calls* to `Class.forName` for which all potential targets are class name constants,
– *partially resolved calls*, which have at least one class name string constant propagating to them, and
– *unresolved calls*, which have no class name string constants propagating to them, only non-constant external sources requiring a specification.

The columns subdivide the total number of calls (T) into fully resolved calls (FR), partially resolved (PR), and unresolved (UR) calls. In the case of

LOCAL analysis, there are no partially resolved calls — calls are either fully resolved to constant strings or unresolved. Similarly, in the case of SOUND analysis, all calls are either fully resolved or unresolved, as further explained in Section 5.5.

5.3 Local Analysis for Reflection Resolution (LOCAL)

To provide a baseline for comparison, we implemented a local intra-method analysis that identifies string constants passed to `Class.forName`. This analysis catches only those reflective calls that can be resolved completely within a single method. Because this technique does not use interprocedural points-to results, it cannot be used for identification of specification points. Furthermore, because for method invocations and field accesses the names of the method or field are typically *not* locally defined constants, we do not perform resolution of method calls and field accesses in LOCAL.

A significant percentage of `Class.forName` calls can be fully resolved by local analysis, as demonstrated by the numbers in column 4, Figure 6. This is partly due to the fact that it is actually quite common to call `Class.forName` with a constant string parameter for side-effects of the call, because doing so invokes the class constructor. Another common idiom contributing the number of calls resolved by local analysis is `T.class`, which is converted to a call to `Class.forName` and is *always* statically resolved.

5.4 Points-to Information for Reflection Resolution (POINTS-TO)

Points-to information is used to find targets of reflective calls to `Class.forName`, `Class.newInstance`, `Method.invoke`, etc. As can be seen from Figure 6, for all of the benchmarks, POINTS-TO information results in more resolved `Class.forName` calls and fewer unresolved ones compared to LOCAL.

Specification Points Quite frequently, some sort of specification is required for reflective calls to be fully resolved. Points-to information allows us to provide the user with a list of specification points where inputs needs to be specified for a conservative answer to be obtained. Among the specification points we have encountered in our experiments, calls to `System.getProperty` to retrieve a system variable and calls to `BufferedReader.readLine` to read a line from a file are quite common. Below we provide a typical example of providing a specification.

Example 2. This example describes resolving reflective targets of a call to `Class.newInstance` in `javax.xml.transform.FactoryFinder` in the JDK in order to illustrate the power and limitation of using points-to information. Class `FactoryFinder` has a method `Class.newInstance` shown in Figure 7. The call to `Class.newInstance` occurs on line 9. However, the exact class instantiated at runtime depends on the `className` parameter, which is passed into this function. This function is invoked from a variety of places with the `className` parameter being read from initialization properties files, the console, etc. In only one case, when `Class.newInstance` is called from another function `find` located in another file, is the `className` parameter a string constant.

This example makes the power of using points-to information apparent — the `Class.newInstance` target corresponding to the string constant is often difficult to

```
private static Object newInstance(String className,
                ClassLoader classLoader) throws ConfigurationError {
   try {
       Class spiClass;
       if (classLoader == null) {
           spiClass = Class.forName(className);
       }
       ...
       return spiClass.newInstance();
   } catch (...)
       ...
   }
```

Fig. 7: Reflection resolution using points-to results in `javax.xml.transform.FactoryFinder` in the JDK.

find by just looking at the code. The relevant string constant was passed down through several levels of method calls located in a different file; it took us more that five minutes of exploration with a powerful code browsing tool to find this case in the source. Resolving this `Class.newInstance` call also requires the user to provide input for four specification points: along with a constant class name, our analysis identifies two specification points, which correspond to file reads, one access of system properties, and another read from a hash table. □

In most cases, the majority of calls to `Class.forName` are fully resolved. However, a small number of unresolved calls are potentially responsible for a large number of specification points the user has to provide. For Points-to, the average number of specification points per invocation site ranges from 3 for `freetts` to 9 for `gruntspud`. However, for `jedit`, the average number of specification points is 422. Specification points computed by the pointer analysis-based approach can be thought of as "hints" to the user as to where provide specification.

In most cases, the user is likely to provide specification at program input points where he knows what the input strings may be. This is because at a reflective call it may be difficult to tell what all the constant class names that flow to it may be, as illustrated by Example 2. Generally, however, the user has a choice. For problematic reflective calls like those in `jedit` that produce a high number of specification points, a better strategy for the user may be to provide reflective specifications at the `Class.forName` *calls themselves* instead of laboriously going through all the specification points.

5.5 Casts for Reflection Resolution (Casts)

Type casts often provide a good first static approximation to what objects can be created at a given reflective creation site. There is a pretty significant increase in the number of `Class.forName` calls reported in Figure 6 in a few cases, including 93 newly discovered `Class.forName` calls in `gruntspud` that apprear due to a bigger call graph when reflective calls are resolved. In all cases, the majority of `Class.forName` calls have their targets at least partially resolved. In fact, as many as 95% of calls are resolved in the case of `jedit`.

As our experience with the Java reflection APIs would suggest, most `Class.newInstance` calls are post-dominated by a cast operation, often located within only a few lines of code of the `Class.newInstance` call. However, in our experiments, we have identified a number of `Class.newInstance` call sites, which were not dominated by a cast of any sort and therefore the return result of `Class.newInstance` could not be constrained in any way. As it turns out, most of these unconstrained `Class.newInstance` call sites are located in the JDK and `sun.*` sources, Apache libraries, etc. Very few were found in application code.

The high number of unresolved calls in the JDK is due to the fact that reflection use in libraries tends to be highly generic and it is common to have "`Class.newInstance` wrappers" — methods that accept a class name as a string and return an object of that class, which is later cast to an appropriate type in the caller method. Since we rely on *intraprocedural* post-dominance, resolving these calls is beyond our scope. However, since such "wrapper" methods are typically called from multiple invocation sites and different sites can resolve to different types, it is unlikely that a precise approximation of the object type returned by `Class.newInstance` is possible in these cases at all.

Precision of Cast Information Many reflective object creation sites are located in the JDK itself and are present in all applications we have analyzed. For example, method `lookup` in package `sun.nio.cs.AbstractCharsetProvider` reflectively creates a subclass of `Charset` and there are 53 different character sets defined in the system. In this case, the answer is precise because all of these charsets can conceivably be used depending on the application execution environment. In many cases, the cast approach is able to *uniquely* pinpoint the target of `Class.newInstance` calls based on cast information. For example, there is only one subclass of class `sun.awt.shell.ShellFolderManager` available to `gruntspud`, so, in order for the cast to succeed, it must be instantiated.

In general, however, the cast-based approach provides an imprecise upper bound on the call graph that needs to be analyzed. Because the results of `Class.newInstance` are occasionally cast to very wide types, such as `java.lang.Cloneable`, many potential subclasses can be instantiated at the `Class.newInstance` call site. The cast-based approach is likely to yield more precise results on applications that use Java generics, because those applications tend to use more narrow types when performing type casts.

5.6 Achieving a Sound Call Graph Approximation (Sound)

Providing a specification for unresolved reflective calls allows us to achieve a sound approximation of the call graph. In order to estimate the amount of effort required to come up with a specification for unresolved reflective calls, we decided to start with Points-to and add a reflection specification until the result became sound. Because providing a specification allows us to discover more of the call graph, two or three rounds of specification were required as new portions of the program became available. In practice, we would start without a specification and examine all unresolved calls and specification points corresponding to them. Then we would come up with a specification and feed it back to the call graph construction algorithm until the process converges.

Coming up with a specification is a difficult and error-prone task that requires looking at a large amount of source code. It took us about ten hours to incrementally devise an appropriate specification and ensure its completeness

	Starting with Strings					**Starting with Casts**				
Benchmark	**Specs**	**Sites**	**Libs**	**App**	**Types/site**	**Specs**	**Sites**	**Libs**	**App**	**Types/site**
`jgap`	1,068	25	21	4	42.72	16	2	2	0	8.0
`freetts`	964	16	14	2	60.25	0	4	3	1	0.0
`gruntspud`	1,014	27	26	1	37.56	18	4	4	0	4.5
`jedit`	1,109	21	19	2	52.81	63	3	2	1	21.0
`columba`	1,006	22	21	1	45.73	16	2	2	0	8.0
`jfreechart`	1,342	21	21	0	63.90	18	4	4	0	4.5

Fig. 8: User-provided specification statistics.

by rerunning the call graph construction algorithm. After providing a reflection specification stringing with Points-to, we then estimate how much of the user-provided specification can be avoided if we were to rely on type casts instead.

Specification Statistics The first part of Figure 8 summarizes the effort needed to provide specifications to make the call graph sound. The second column shows the number of specifications of the form *reflective call site => type*, as exemplified by Figure 3. Columns 3–5 show the number of reflection calls sites covered by each specification, breaking them down into sites that located within library vs application code. As can be seen from the table, while the number of invocation sites for which specifications are necessary is always around 20, only a few are part of the application. Moreover, in the case of `jfreechart`, *all* of the calls requiring a specification are part of the library code.

Since almost all specification points are located in the JDK and library code, specification can be shared among different applications. Indeed, there are only 40 *unique* invocation sites requiring a specification across all the benchmarks. Column 6 shows the average number of types specified per reflective call site. Numbers in this columns are high because most reflective calls within the JDK can refer to a multitude of implementation classes.

The second part of Figure 8 estimates the specification effort required if were were to start with a cast-based call graph construction approach. As can be seen from columns 8–10, the number of `Class.forName` calls that are not constrained by a cast operation is quite small. There are, in fact, only 14 unique invocation sites — or about a third of invocation sites required for Points-to. This suggests that the the effort required to provide a specification to make Casts sound is considerably smaller than our original effort that starts with Points-to.

Specification Difficulties In some cases, determining meaningful values to specify for `Class.forName` results is quite difficult, as shown by the example below.

Example 3. One of our benchmark applications, `jedit`, contains an embedded Bean shell, a Java source interpreter used to write editor macros. One of the calls to `Class.forName` within `jedit` takes parameters extracted from the Bean shell macros. In order to come up with a conservative superset of classes that may be invoked by the Bean shell interpreter for a given installation of `jedit`, we parse the scripts that are supplied with `jedit` to determine imported Java classes they have access to. (We should note that this specification is only sound for the default configuration of `jedit`; new classes may need to be added to the specification if new macros become available.) It took us a little under an hour to develop appropriate Perl scripts to do the parsing of 125 macros supplied with `jedit`. The `Class.forName` call can instantiate a total of 65 different types. □

We should emphasize that the conservativeness of the call graph depends on the conservativeness of the user-provided specification. If the specification missed potential relations, they will be also omitted from the call graph. Furthermore, a specification is typically only conservative for a given configuration of an application: if initialization files are different for a different program installation, the user-provided specification may no longer be conservative.

Remaining Unresolved Calls Somewhat surprisingly, there are *still* some `Class.forName` calls that are not fully resolved given a user-provided specification, as can be seen from the last column in Figure 6. In fact, this is not a specification flaw: no valid specification is *possible* for those cases, as explained below.

Example 4. The audio API in the JDK includes method `javax.sound.sampled.AudioSystem.getDefaultServices`, which is not called in Java version 1.3 and above. A `Class.forName` call within that method resolves to constant `com.sun.media.sound.DefaultServices`, however, this class is absent in post-1.3 JDKs. However, since this method represents dead code, our answer is still sound. Similarly, other unresolved calls to `Class.forName` located within code that is not executed for the particular application configuration we are analyzing refer to classes specific to MacOS and unavailable on Linux, which is the platform we performed analysis on. In other cases, classes were unavailable for JDK version 1.4.2_08, which is the version we ran our analysis on. □

5.7 Effect of Reflection Resolution on Call Graph Size

Figure 9 compares the number of classes and methods across different analysis versions. Local analysis does not have any significant effect on the number of methods or classes in the call graph, even though most of the calls to

Classes						
Benchmark	None	Local	Points-to	Casts	Sound	
`jgap`	264	264	268	276	1,569	5.94
`freetts`	309	309	309	351	1,415	4.58
`gruntspud`	1,258	1,258	1,275	2,442	2,784	2.21
`jedit`	1,660	1,661	1,726	2,152	2,754	1.66
`columba`	961	962	966	1,151	2,339	2.43
`jfreechart`	884	881	886	1,560	2,340	2.65

Methods						
Benchmark	None	Local	Points-to	Casts	Sound	
`jgap`	1,013	1,014	1,038	1,075	6,676	6.58
`freetts`	1,357	1,358	1,358	1,545	5,499	4.05
`gruntspud`	7,321	7,321	7,448	14,164	14,368	1.96
`jedit`	11,230	11,231	11,523	13,487	16,003	1.43
`columba`	5,636	5,642	5,652	6,199	12,001	2.13
`jfreechart`	5,374	5,374	5,392	8,375	12,111	2.25

Fig. 9: Number of classes and methods in the call graph for different analysis versions.

`Class.forName` can be resolved with local analysis. This is due to the fact that the vast majority of these calls are due to the use of the `T.class` idiom, which typically refer to classes that are already within the call graph. While these trivial calls are easy to resolve, it is the analysis of the other "hard" calls with a lot of potential targets that leads to a substantial increase in the call graph size.

Using POINTS-TO increases the number of classes and methods in the call graph only moderately. The biggest increase in the number of methods occurs for `jedit` (293 methods). Using CASTS leads to significantly bigger call graphs, especially for `gruntspud`, where the increase in the number of methods compared to NONE is almost two-fold.

The most noticeable increase in call graph size is observed in version SOUND. Compared to NONE, the average increase in the number of classes is 3.2 times the original and the average increase for the number of methods is 3 times the original. The biggest increase in the number of methods occurs in `gruntspud`, with over 7,000 extra methods added to the graph.

Figure 9 also demonstrate that the lines of code metric is not always indicative of the size of the final call graph — programs are listed in the increasing order of line counts, yet, `jedit` and `gruntspud` are clearly the biggest benchmarks if we consider the method count. This can be attributed to the use of large libraries that ship with the application in binary form as well as considering a much larger portion of the JDK in version SOUND compared to version NONE.

6 Related Work

General treatments of reflection in Java are given in Forman and Forman [1] and Guéhéneuc et al. [15]. The rest of the related work falls into the following broad categories: projects that explicitly deal with reflection in Java and other languages; approaches to call graph construction in Java; and finally, static and dynamic analysis algorithms that address the issue of dynamic class loading.

6.1 Reflection and Metadata Research

The metadata and reflection community has a long line of research originating in languages such as Scheme [16]. We only mention a few relevant projects here. The closest static analysis project to ours we are aware of is the work by Braux and Noyé on applying partial evaluation to reflection resolution for the purpose of optimization [17]. Their paper describes extensions to a standard partial evaluator to offer reflection support. The idea is to "compile away" reflective calls in Java programs, turning them into regular operations on objects and methods, given constraints on the concrete types of the object involved. The type constraints for performing specialization are provided by hand.

Our static analysis can be thought of as a tool for inferring such constraints, however, as our experimental results show, in many cases targets of reflective calls cannot be uniquely determined and so the benefits of specialization to optimize program execution may be limited. Braux and Noyé present a description of how their specialization approach may work on examples extracted from the JDK, but lacks a comprehensive experimental evaluation. In related work for languages other than Java, Ruf explores the use of partial evaluation as an optimization technique in the context of CLOS [18].

Specifying reflective targets is explicitly addressed in Jax [19]. Jax is concerned with reducing the size of Java applications in order to reduce download time; it reads in the class files that constitute a Java application, and performs a whole-program analysis to determine the components of the application that must be retained in order to preserve program behavior. Clearly, information about the true call graph is necessary to ensure that no relevant parts of the application are pruned away. Jax's approach to reflection is to employ user-provided specifications of reflective calls. To assist the user with writing complete specification files, Jax relies on dynamic instrumentation to discover the missing targets of reflective calls. Our analysis based on points-to information can be thought of as a tool for determining where to insert reflection specifications.

6.2 Call Graph Construction

A lot of effort has been spent of analyzing function pointers in C as well as virtual method calls in C++ and Java. We briefly mention some of the highlights of call graph construction algorithms for Java here. Grove et al. present a parameterized algorithmic framework for call graph construction [12, 20]. They empirically assess a multitude of call graph construction algorithms by applying them to a suite of medium-sized programs written in Cecil and Java. Their experience with Java programs suggests that the effect of using context sensitivity for the task of call graph construction in Java yields only moderate improvements.

Tip and Palsberg propose a propagation-based algorithm for call graph construction and investigate the design space between existing algorithms for call graph construction such as 0-CFA and RTA, including RA, CHA, and four new ones [7]. Sundaresan et al. go beyond the tranditional RTA and CHA approaches in Java and and use type propagation for the purpose of obtaining a more precise call graph [21]. Their approach of using variable type analysis (VTA) is able to uniquely determine the targets of potentially polymorphic call sites in 32% to 94% of the cases. Agrawal et al. propose a demand-driven algorithm for call graph construction [22]. Their work is motivated by the need for just-in-time or dynamic compilation as well as program analysis used as part of software development environments. They demonstrate that their demand-driven technique has the same accuracy as the corresponding exhaustive technique. The reduction in the graph construction time depends upon the ratio of the cardinality of the set of influencing nodes to the set of all nodes.

6.3 Dynamic Analysis Approaches

Our work is motivated to a large extend by the need of error detection tool to have a static approximation of the true conservative call graph of the application. This largely precludes dynamic analysis that benefits optimizations such as method inlining and connectivity-based garbage collection.

A recent paper by Hirzel, Diwan, and Hind addresses the issues of dynamic class loading, native methods, and reflection in order to deal with the full complexity of Java in the implementation of a common pointer analysis [5]. Their approach involves converting the pointer analysis [6] into an online algorithm: they add constraints between analysis nodes as they are discovered at runtime. Newly generated constraints cause re-computation and the results are propagated to analysis clients such as a method inliner and a garbage collector at runtime. Their approach leverages the class hierarchy analysis (CHA) to update the call graph. Our technique uses a more precise pointer analysis-based approach to call graph construction.

7 Conclusions

This paper presents the first static analysis for call graph construction in Java that addresses reflective calls. Our algorithm uses the results of a points-to analysis to determine potential reflective call targets. When the calls cannot be fully resolved, user-provided specification is requested. As an alternative to providing specification, type cast information can be used to provide a conservative approximation of reflective call targets.

We applied our static analysis techniques to the task of constructing call graphs for six large Java applications, some consisting of more than 190,000 lines of code. Our evaluation showed that as many as 95% of reflective `Class.forName` could at least partially be resolved to statically determined targets with the help of points-to results and cast information *without* providing any specification.

While most reflective calls are relatively easy to resolve statically, *precisely* interpreting some reflective calls requires a user-provided specification. Our pointer analysis-based approach also identified specification points — places in the program corresponding to file and system property read operations, etc., where user input is needed in order to obtain a full call graph. Our evaluation showed that the construction of a specification that makes the call graph conservative is a time-consuming and error-prone task. Fortunately, our cast-based approach can drastically reduce the specification burden placed on the user by providing a conservative, albeit potentially imprecise approximation of reflective targets.

Our experiments confirmed that ignoring reflection results in missing significant portions of the call graph, which is not something that effective static analysis tools can afford. While the local and points-to analysis techniques resulted in only a moderate increase in call graph size, using the cast-based approach resulted in call graphs with as many as 1.5 times more methods than the original call graph. Furthermore, providing a specification resulted in much larger conservative call graphs that were almost 7 times bigger than the original. For instance, in one our benchmark, an additional 7,047 methods were discovered in the conservative call graph version that were not present in the original.

References

1. Forman, I.R., Forman, N.: Java Reflection in Action. Manning Publications (2004)
2. Koved, L., Pistoia, M., Kershenbaum, A.: Access rights analysis for Java. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2002) 359 – 372

3. Reimer, D., Schonberg, E., Srinivas, K., Srinivasan, H., Alpern, B., Johnson, R.D., Kershenbaum, A., Koved, L.: SABER: Smart Analysis Based Error Reduction. In: Proceedings of International Symposium on Software Testing and Analysis. (2004) 243 – 251
4. Weimer, W., Necula, G.: Finding and preventing run-time error handling mistakes. In: Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004) 419 – 431
5. Hirzel, M., Diwan, A., Hind, M.: Pointer analysis in the presence of dynamic class loading. In: Proceedings of the European Conference on Object-Oriented Programming, Systems, Languages, and Applications. (2004) 96–122
6. Andersen, L.O.: Program analysis and specialization for the C programming language. PhD thesis, University of Copenhagen (1994)
7. Tip, F., Palsberg, J.: Scalable propagation-based call graph construction algorithms. ACM SIGPLAN Notices **35** (2000) 281–293
8. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java, `http://suif.stanford.edu/~livshits/papers/tr/reflection_tr.pdf`. Technical report, Stanford University (2005)
9. Whaley, J., Lam, M.: Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In: Proceedings of the ACM Conference on Programming Language Design and Implementation. (2004) 131 – 144
10. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy analysis. Lecture Notes in Computer Science **952** (1995) 77–101
11. Bacon, D.F.: Fast and Effective Optimization of Statically Typed Object-Oriented Languages. PhD thesis, University of California at Berkeley (1998)
12. Grove, D., Chambers, C.: A framework for call graph construction algorithms. ACM Trans. Program. Lang. Syst. **23** (2001) 685–746
13. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proceedings of the ACM Symposium on Principles of Database Systems. (2005) 1 – 12
14. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley (1986)
15. Guéhéneuc, Y.G., Cointe, P., Ségura-Devillechaise, M.: Java reflection exercises, correction, and FAQs. `http://www.yann-gael.gueheneuc.net/Work/Teaching/Documents/Practical-ReflectionCourse.doc.pdf` (2002)
16. Thiemann, P.: Towards partial evaluation of full Scheme. In: Reflection '96. (1996)
17. Braux, M., Noyé, J.: Towards partially evaluating reflection in Java. In: Proceedings of the ACM Workshop on Partial Evaluation and Semantics-based Program Manipulation. (1999) 2–11
18. Ruf, E.: Partial evaluation in reflective system implementations. In: Workshop on Reflection and Metalevel Architecture. (1993)
19. Tip, F., Laffra, C., Sweeney, P.F., Streeter, D.: Practical experience with an application extractor for Java. ACM SIGPLAN Notices **34** (1999) 292–305
20. Grove, D., DeFouw, G., Dean, J., Chambers, C.: Call graph construction in object-oriented languages. In: Proceedings of the ACM Conference on Object-oriented Programming, Systems, Languages, and Applications. (1997) 108–124
21. Sundaresan, V., Hendren, L., Razafimahefa, C., Vallée-Rai, R., Lam, P., Gagnon, E., Godin, C.: Practical virtual method call resolution for Java. ACM SIGPLAN Notices **35** (2000) 264–280
22. Agrawal, G., Li, J., Su, Q.: Evaluating a demand driven technique for call graph construction. In: Computational Complexity. (2002) 29–45