Understanding and Analyzing Java Reflection

YUE LI*, UNSW Sydney, Australia
TIAN TAN*, UNSW Sydney, Australia
JINGLING XUE, UNSW Sydney, Australia

Java reflection has been widely used in a variety of applications and frameworks. It allows a software system to inspect and change the behaviour of its classes, interfaces, methods and fields at runtime, enabling the software to adapt to dynamically changing runtime environments. However, this dynamic language feature imposes significant challenges to static analysis, because the behaviour of reflection-rich software is logically complex and statically hard to predict. As a result, existing static analysis tools either ignore reflection or handle it partially, resulting in missed, important behaviours, i.e., unsound results. Therefore, improving or even achieving soundness in (static) reflection analysis—an analysis that infers statically the behaviour of reflective code—will provide significant benefits to many analysis clients, such as bug detectors, security analyzers and program verifiers.

In this paper, we provide a comprehensive understanding of Java reflection through examining its underlying concept, API and real-world usage, and, building on this, we introduce a new static approach to resolving Java reflection effectively in practice. We have implemented our reflection analysis in an open-source tool, called SOLAR, and evaluated its effectiveness extensively with large Java programs and libraries. Our experimental results demonstrate that SOLAR is able to (1) resolve reflection more soundly than the state-of-the-art reflection analyses; (2) automatically and accurately identify the parts of the program where reflection is resolved unsoundly or imprecisely; and (3) guide users to iteratively refine the analysis results by using lightweight annotations until their specific requirements are satisfied.

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

Additional Key Words and Phrases: Java reflection, static analysis, points-to analysis

ACM Reference Format:

1 INTRODUCTION

Java reflection allows a software system to inspect and change the behaviour of its classes, interfaces, methods and fields at runtime, enabling the software to adapt to dynamically changing runtime environments. This dynamic language feature eases the development and maintenance of Java programs in many programming tasks by, for example, facilitating their flexible integration with the third-party code and their main behaviours to be configured according to a deployed runtime

*This work was mostly done while these authors were at UNSW Sydney. Both authors are now affiliated with Aarhus University, Denmark.

Authors’ addresses: Yue Li, Aarhus University, Denmark, yueli@cs.au.dk; Tian Tan, Aarhus University, Denmark, tiantan@cs.au.dk; Jingling Xue, UNSW Sydney, Australia, jingling@cs.au.dk.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

© 2019 Association for Computing Machinery.
1049-331X/2019/02-ART7 $15.00
https://doi.org/10.1145/nnnnnn.nnnnnnn

environment in a decoupled way. Due to such advantages, reflection has been widely used in a variety of Java applications and frameworks [32, 69].

Static analysis is widely recognized as a fundamental tool for bug detection [17, 43], security vulnerability analysis [1, 36], compiler optimization [14, 59], program verification [4, 12], and program debugging and understanding [34, 58]. However, when applying static analysis to Java programs, reflection poses a major obstacle [32, 33, 38, 51]. If the behavior of reflective code is not resolved well, much of the codebase will be rendered invisible for static analysis, resulting in missed, important behaviours, i.e., unsound analysis results [37]. Therefore, improving or even achieving soundness in (static) reflection analysis—an analysis that infers statically the behavior of reflective code—will provide significant benefits to all the client analyses as just mentioned above.

1.1 Challenges
Developing effective reflection analysis for real-world programs remains a hard problem, widely acknowledged by the static analysis community [37]:

“Reflection usage and the size of libraries/frameworks make it very difficult to scale points-to analysis to modern Java programs.” [64];

“Reflection makes it difficult to analyze statically.” [48];

“In our experience [18], the largest challenge to analyzing Android apps is their use of reflection ...” [2];

“Static analysis of object-oriented code is an exciting, ongoing and challenging research area, made especially challenging by dynamic language features, a.k.a. reflection.” [26].

There are three reasons on why it is hard to untangle this knotty problem:

- The Java reflection API is large and its common uses in Java programs are complex. It remains unclear how an analysis should focus on its effort on analyzing which of its reflection methods in order to achieve some analysis results as desired.

- The dynamic behaviours of reflective calls are mainly specified by their string arguments, which are usually unknown statically (e.g., with some string values being encrypted, read from configuration files, or retrieved from the Internet).

- The reflective code in a Java program cannot be analyzed alone in isolation. To resolve reflective calls adequately, a reflection analysis often works inter-dependently with a pointer analysis [32, 33, 38, 50, 51], with each being both the producer and consumer of the other. When some reflective calls are not yet resolved, the pointer information that is currently available can be over- or under-approximate. Care must be taken to ensure that the reflection analysis helps increase soundness (coverage) while still maintaining sufficient precision for the pointer analysis. Otherwise, the combined analysis would be unscalable for large programs.

As a result, most of the papers on static analysis for object-oriented languages, like Java, treat reflection orthogonally (often without even mentioning its existence). Existing static analysis tools either ignore reflection or handle it partially and ineffectively.

1.2 Previous Approaches
Initially, reflection analysis mainly relies on string analysis, especially when the string arguments to reflective calls are string constants, to resolve reflective targets, i.e., methods or fields reflectively accessed. Currently, this mainstream approach is still adopted by many static analysis tools for Java, such as SOOT, WALA, CHORD and DOOP. However, as described in Section 1.1, string analysis will fail in many situations where string arguments are unknown (Figure 5), resulting in limited soundness
and precision. As a static analysis, a (more) sound reflection analysis is one that allows (more) true reflective targets (i.e., targets that are actually accessed at runtime) to be resolved statically. In practice, any reflection analysis must inevitably make a trade-off among soundness, precision, scalability, and (sometimes) automation.

In addition, existing reflection analyses [2, 8, 20, 22, 28, 32, 35, 38, 51, 68] cannot answer two critical questions that are raised naturally, in practice: Q(1) how sound is a given reflection analysis and Q(2) which reflective calls are resolved unsoundly or imprecisely? We argue for their importance as follows:

- If Q(1) is unanswered, users would be unsure (or lose confidence) about the effectiveness of the analysis results produced. For example, a bug detector that reports no bugs may actually miss many bugs if some reflective calls are resolved unsoundly.
- If Q(2) is unanswered, users would not have an opportunity to contribute in improving the precision and soundness of the analysis results, e.g., by providing some user annotations. For some client analyses (e.g., verification), soundness is required.

1.3 Contributions

In this paper, we attempt to uncover the mysterious veil of Java reflection and change the informed opinion in the program analysis community about static reflection analysis: “Java reflection is a dynamic feature which is nearly impossible to handle effectively in static analysis”. Specifically, we make the following contributions:

- We provide a comprehensive understanding of Java reflection through examining its underlying concept (what it is), interface (how its API is designed), and real-world usage (how it is used in practice). As a result, we will provide the answers to several critical questions, which are somewhat related, including:
  - What is reflection, why is it introduced in programming languages, and how is Java reflection derived from the basic reflection concept?
  - Which methods of the Java reflection API should be analyzed carefully and how are they related, as the API is large and complex (with about 200 methods)?
  - How is reflection used in real-world Java programs and what can we learn from its common uses? We have conducted a comprehensive study about reflection usage in a set of 16 representative Java programs by examining their 1,423 reflective call sites. We report 7 useful findings to enable the development of improved practical reflection analysis techniques and tools in future research.

- We introduce a new static analysis approach, called SOLAR (soundness-guided reflection analysis), to resolve Java reflection effectively in practice. As shown in Figure 1, SOLAR has three unique advantages compared with previous work:
  - SOLAR is able to yield significantly more sound results than the state-of-the-art reflection analysis. In addition, SOLAR allows its soundness to be reasoned about when some reasonable assumptions are met.
  - SOLAR is able to accurately identify the parts of the program where reflection is analyzed unsoundly or imprecisely, making it possible for users to be aware of the effectiveness of their analysis results (as discussed in Section 1.2).
  - SOLAR provides a mechanism to guide users to iteratively refine the analysis results by adding lightweight annotations until their specific requirements are satisfied, enabling reflection to be analyzed in a controlled manner.
• We have implemented SOLAR in Doop [8] (a state-of-the-art pointer analysis tool for Java) and released it as an open-source tool. In particular, SOLAR can output its reflection analysis results with the format that is supported by Soot [63] (a popular framework for analyzing Java and Android applications), allowing Soot’s clients to use SOLAR’s results directly.

• We conduct extensive experiments on evaluating SOLAR’s effectiveness with large Java applications and libraries. Our experimental results provide convincing evidence on the ability of SOLAR in analyzing Java reflection effectively, in practice.

1.4 Organization
The rest of this paper is organized as follows. We will start by providing a comprehensive understanding of Java reflection in Section 2. Building on this understanding, we give an overview of SOLAR in Section 3 and introduce its underlying methodology in Section 4. Then, we formalize SOLAR in Section 5, describe its implementation in Section 6, and evaluate its effectiveness in Section 7. Finally, we discuss the related work in Section 8 and conclude in Section 9.

2 UNDERSTANDING JAVA REFLECTION
Java reflection is a useful but complex language feature. To gain a deep understanding about Java reflection, we examine it in three steps. First, we describe what Java reflection is, why we need it, and how it is proposed (Section 2.1). Second, we explain how Java reflection is designed to be used, i.e., its API (Section 2.2). Finally, we investigate comprehensively how it has been used in real-world Java applications (Section 2.3). After reading this section, the readers are expected to develop a whole picture about the basic mechanism behind Java reflection, understand its core API design, and capture the key insights needed for developing practical reflection analysis tools.

2.1 Concept
Reflection, which has long been studied in philosophy, represents one kind of human abilities for introspecting and learning their nature. Accordingly, a (non-human) object can also be endowed with the capability of such self-awareness. This arises naturally in artificial intelligence: “Here I am walking into a dark room. Since I cannot see anything, I should turn on the light”. As explained in [54], “such thought fragment reveals a self-awareness of behaviour and state, one that leads to a
change in that selfsame behaviour and state”, which allows an object to examine itself and leverage the meta-level information to figure out what to do next.

Similarly, when we enable programs to avail themselves of such reflective capabilities, reflective programs will also allow the programs to observe and modify properties of their own behaviour. Thus, let a program be self-aware — this is the basic motivation of the so-called computational reflection, which is also considered as the reflection used in the area of programming languages [15].

In the rest of this section, we will introduce what computational reflection is (Section 2.1.1), what reflective abilities it supports (Section 2.1.2), and how Java reflection is derived from it (Section 2.1.3).

2.1.1 Computational Reflection. Reflection, as a concept for computational systems, dates from Brian Smith’s doctoral dissertation [53]. Generally, as shown in Figure 2(a), a computational system is related to a domain and it answers questions about and/or support actions in the domain [39].

As described in [39], a computational system “incorporates internal structures representing the domain. These structures include data representing entities and relations in the domain and a program describing how these data may be manipulated”.

A computational system $S$ is said to be also a reflective system, as shown in Figure 2(b), if the following two conditions are satisfied:

- First, the system $S$ has its own representation, known as its self-representation or metasystem, in its domain as a kind of data to be examined and manipulated.
- Second, the system $S$ and its representation are causally connected: a change to the representation implies a change to the system, and vice versa.

The base system $S$ should be reified into its representation before its metasystem can operate. Then the metasystem examines and manipulates its behaviour using the reified representation. If any changes are made by the metasystem, then the effects will also be reflected in the behaviour of the corresponding base system.

2.1.2 Reflective Abilities. Generally, (computational) reflection is the ability of a program to examine and modify the structure and behavior of a program at runtime [23, 40]. Thus, it endows the program the capabilities of self-awareness and self-adapting. These two reflective abilities are known as introspection and intercession, respectively, and both require a reification mechanism to encode a program’s execution state as data first [15].

- **Introspection**: the ability of a program to observe, and consequently, reason about its own execution state.

Fig. 2. Computational vs. reflective computational systems.
• **Intercession**: the ability of a program to modify its own execution state or alter its own interpretation or meaning.

Providing full reflective abilities as shown above is challenging in practice, as this will introduce both implementation complexities and performance problems [10]. Thus, in modern programming languages like Java, reflective abilities are only partially supported [6, 19].

### 2.1.3 Java Reflection
Java reflection supports introspection and very limited intercession; in particular, an introspection step is usually followed by behaviour changes such as object creation, method invocation and attribute manipulation [9, 19]. Note that some other researchers hold a different view that Java reflection does not support intercession [6, 16], as they adopt a more strict definition of intercession, which implies the ability to modify the self-representation of a program.

Despite its limited reflective abilities, Java reflection is able to allow programmers to break the constraints of staticity and encapsulation, enabling the program to adapt to dynamically changing runtime environments. As a result, Java reflection has been widely used in real-world Java applications to facilitate flexibly different programming tasks, such as reasoning about control (i.e., about which computations to pursue next) [19], interfacing (e.g., interaction with GUIs or database systems) [21, 46], and self-activation (e.g., through monitors) [13].

Java reflection does not have a reify operation as described in Section 2.1.1 (Figure 2(b)) to turn the basic (running) system (including stack frames) into a representation (data structure) that is passed to a metasystem. Instead, a kind of metarepresentation, based on *metaobjects*, exists when the system starts running and persists throughout the execution of the system [19].

A metaobject is like the reflection in a mirror: one can adjust one’s smile (behaviour changes) by looking at oneself in a mirror (introspection). In Section 2.2, we will look at how Java reflection uses metaobjects and its API to facilitate reflective programming.

```java
A a = new A();
String cName, mName, fName = ...;
Class clz = Class.forName(cName);
Object obj = clz.newInstance();
Method mtd = clz.getDeclaredMethod(mName, A.class);
Object l = mtd.invoke(obj, a);
Field fld = clz.getField(fName);
X r = (X)fld.get(a);
fld.set(null, a);
```

Fig. 3. An example of reflection usage in Java.

### 2.2 Interface
We first use a toy example to illustrate some common uses of the Java reflection API (Section 2.2.1). We then delve into the details of its core methods, which are relevant to (and thus should be handled by) any reflection analysis (Section 2.2.2).

#### 2.2.1 An Example
There are two kinds of metaobjects: *Class* objects and member objects. In Java reflection, one always starts with a *Class* object and then obtain its member objects (e.g., *Method* and *Field* objects) from the *Class* object by calling its corresponding accessor methods (e.g., `getMethod()` and `getField()`).
In Figure 3, the metaobjects clz, mtd and fld are instances of the metaobject classes Class, Method and Field, respectively. Constructor can be seen as Method except that the method name "<init>" is implicit. Class allows an object to be created reflectively by calling newInstance(). As shown in line 4, the dynamic type of obj is the class (type) represented by clz (specified by cName). In addition, Class provides accessor methods such as getDeclaredMethod() in line 5 and getField() in line 7 to allow the member metaobjects (e.g., of Method and Field) related to a Class object to be introspected. With dynamic invocation, a Method object can be commanded to invoke the method that it represents (line 6). Similarly, a Field object can be commanded to access or modify the field that it represents (lines 8 and 9).

2.2.2 Core Java Reflection API. In reflection analysis, we are concerned with reasoning about how reflection affects the control and data flow information in the program. For example, if a target method (say m) that is reflectively invoked in line 6 in Figure 3 cannot be resolved statically, the call graph edge from this call site to method m (control flow) and the values passed interprocedurally from obj and a to this and the parameter of m (data flow), respectively, will be missing. Therefore, we should focus on the part of the Java reflection API that affects a pointer analysis, a fundamental analysis that statically resolves the control and data flow information in a program [27, 30–32, 38, 41, 42, 51, 52, 60–62].

It is thus sufficient to consider only the pointer-affecting methods in the Java reflection API. We can divide such reflective methods into three categories (Figure 4):

---

1We summarize and explain the core reflection API (25 methods) that is critical to static analysis. A more complete reflection API list (181 methods) is given in [26] without explanations though.
• *class-retrieving methods*, which create `Class` objects, e.g., `forName()` in line 3 in Figure 3.

• *member-retrieving methods*, which introspect and retrieve member metaobjects, i.e., `Method` (Constructor) and `Field` objects from a `Class` object, e.g., `getMethod()` in line 5 and `getField()` in line 7 in Figure 3.

• *reflective-action methods*, which affect the pointer information in the program reflectively, e.g., `newInstance()`, `invoke()`, `get()` and `set()` in lines 4, 6, 8 and 9 in Figure 3 for creating an object, invoking a method, accessing and modifying a field, respectively.

**Class-Retrieving Methods.** Everything in Java reflection begins with `Class` objects and they are returned by calling `class-retrieving` methods. There are many class-retrieving methods in the Java reflection API. In Figure 4, only the four most widely used ones are listed explicitly.

Note that `forName()` (loadClass()) returns a `Class` object representing a class that is specified by the value of its string argument. The `Class` object returned by `o.getClass()` and `A.class` represents the dynamic type (class) of `o` and `A`, respectively.

**Member-Retrieving Methods.** `Class` provides a number of accessor methods for retrieving its member metaobjects, i.e., the `Method` (Constructor) and `Field` objects. In addition, these member metaobjects can be used to introspect the methods, constructors and fields in their target class. Formally, these accessor methods are referred to here as the *member-retrieving* methods.

As shown in Figure 4, for each kind of member metaobjects, there are four member-retrieving methods. We take a `Method` object as an example to illustrate these methods, whose receiver objects are the `Class` objects returned by the class-retrieving methods.

• `getDeclaredMethod(String, Class[])` returns a `Method` object that represents a declared method of the target `Class` object with the name (formal parameter types) specified by the first (second) parameter (line 5 in Figure 3).

• `getMethod(String, Class[])` is similar to `getDeclaredMethod(String, Class[])` except that the returned `Method` object is public (either declared or inherited). If the target `Class` does not have a matching method, then its superclasses are searched first recursively (bottom-up) before its interfaces (implemented).

• `getDeclaredMethods()` returns an array of `Method` objects representing all the methods declared in the target `Class` object.

• `getMethods()` is similar to `getDeclaredMethods()` except that all the public methods (either declared or inherited) in the target `Class` object are returned.

**Reflective-Action Methods.** As shown in Figure 4, a total of nine reflective-action methods that can possibly modify or use (as their side effects) the pointer information in a program are listed. Accordingly, Table 1 explains how these methods affect the pointer information by giving their side effects on the pointer analysis.

In Figure 4, the first five reflective-action methods use four kinds of metaobjects as their receiver objects while the last four methods use `Class` or `Array` objects as their arguments. Below we briefly examine them in the order given in Table 1.

• The side effect of `newInstance()` is allocating an object with the type specified by its metaobject `clz` or `ctor` (say `A`) and initializing it via a constructor of `A`, which is the default constructor in the case of `Class::newInstance()` and the constructor specified explicitly in the case of `Constructor::newInstance()`.

• The side effect of `invoke()` is a virtual call when the first argument of `invoke()`, say `o`, is not null. The receiver object is `o` as shown in the “Side Effect” column in Table 1. When `o` is null, `invoke()` should be a static call.
Table 1. Nine reflective-action methods and their side effects on the pointer analysis, assuming that the target class of \textit{clz} and \textit{ctor} is \textit{A}, the target method of \textit{mtd} is \textit{m} and the target field of \textit{fld} is \textit{f}.

<table>
<thead>
<tr>
<th>Simplified Method</th>
<th>Calling Scenario</th>
<th>Side Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class::newInstance</td>
<td>o = clz.newInstance()</td>
<td>o = new A()</td>
</tr>
<tr>
<td>Constructor::newInstance</td>
<td>o = ctor.newInstance([arg1, ...])</td>
<td>o = new A[arg1, ...]</td>
</tr>
<tr>
<td>Method::invoke</td>
<td>a = mtd.invoke(o, [arg1, ...])</td>
<td>a = o.m(arg1, ...)</td>
</tr>
<tr>
<td>Field::get</td>
<td>a = fld.get(o)</td>
<td>a = o.f</td>
</tr>
<tr>
<td>Field::set</td>
<td>fld.set(o, a)</td>
<td>o.f = a</td>
</tr>
<tr>
<td>Proxy::newProxyInstance</td>
<td>o = Proxy.newProxyInstance(...)</td>
<td>o = new Proxy$*(...)</td>
</tr>
<tr>
<td>Array::newInstance</td>
<td>o = Array.newInstance(clz, size)</td>
<td>o = new A[size]</td>
</tr>
<tr>
<td>Array::get</td>
<td>a = Array.get(o, i)</td>
<td>a = o[i]</td>
</tr>
<tr>
<td>Array::set</td>
<td>Array.set(o, i, a)</td>
<td>o[i] = a</td>
</tr>
</tbody>
</table>

- The side effects of \texttt{get()} and \texttt{set()} are retrieving (loading) and modifying (storing) the value of a instance field, respectively, when their first argument, say \textit{o}, is not \texttt{null}; otherwise, they are operating on a static field.
- The side effect of \texttt{newProxyInstance()} is creating an object of a proxy class \texttt{Proxy$*$}, and this proxy class is generated dynamically according to its arguments (containing a \texttt{Class} object). \texttt{Proxy.newProxyInstance()} can be analyzed according to its semantics. A call to this method returns a \texttt{Proxy} object, which has an associated invocation handler object that implements the \texttt{InvocationHandler} interface. A method invocation on a \texttt{Proxy} object through one of its \texttt{Proxy} interfaces will be dispatched to the \texttt{invoke()} method of the object’s invocation handler.
- The side effect of \texttt{Array.newInstance()} is creating an array (object) with the component type represented by the \texttt{Class} object (e.g., \textit{clz} in Table 1) used as its first argument. \texttt{Array.get()} and \texttt{Array.set()} are retrieving and modifying an index element in the array object specified as their first argument, respectively.

2.3 Reflection Usage

The Java reflection API is rich and complex. We have conducted an empirical study to understand reflection usage in practice in order to guide the design and implementation of a sophisticated reflection analysis described in this paper. In this section, we first list the focus questions in Section 2.3.1, then describe the experimental setup in Section 2.3.2, and finally, present the study results in Section 2.3.3.

2.3.1 Focus Questions. We address the following seven focus questions in order to understand how Java reflection is used in the real world.

- Q1. Existing reflection analyses resolve reflection by analyzing statically the string arguments of class-retrieving and member-retrieving method calls. How often are these strings constants
and how often can non-constant strings be resolved by a simple string analysis that models string operations such as "+" and append()?

- Q2. Existing reflection analyses ignore the member-retrieving methods that return an array of member metaobjects. Is it necessary to handle such methods?
- Q3. Existing reflection analyses usually treat reflective method calls and field accesses as being non-static. Does this treatment work well in real-world programs? Specifically, how often are static reflective targets used in reflective code?
- Q4. In [38], intraprocedural post-dominating cast operations are leveraged to resolve newInstance() when its class type is unknown. This approach is still adopted by many reflection analysis tools. Does it generally work in practice?
- Q5. The Java reflection API contains many class-retrieving methods for returning Class objects. Which ones should be focused on by an effective reflection analysis?
- Q6. The core part of reflection analysis is to resolve all the nine reflective-action methods (Table 1) effectively. What are the reflective-action methods that are most widely used and how are the remaining ones used in terms of their relative frequencies?
- Q7. What are new insights on handling Java reflection (from this paper)?

2.3.2 Experimental Setup. We have selected a set of 16 representative Java programs, including three popular desktop applications, javac-1.7.0, jEdit-5.1.0 and Eclipse-4.2.2 (denoted Eclipse4), two popular server applications, Jetty-9.0.5 and Tomcat-7.0.42, and all eleven DaCapo benchmarks (2006-10-MR2) [3]. Note that the DaCapo benchmark suite includes an older version of Eclipse (version 3.1.2). We exclude its bloat benchmark since its application code is reflection-free. We consider lucene instead of luindex and lusearch separately since these two benchmarks are derived from lucene with the same reflection usage.

We consider a total of 191 methods in the Java reflection API (version 1.6), including the ones mainly from package java.lang.reflect and class java.lang.Class.

We use Soot [63] to pinpoint the calls to reflection methods in the bytecode of a program. To understand the common reflection usage, we consider only the reflective calls found in the application classes and their dependent libraries but exclude the standard Java libraries. To increase the code coverage for the five applications considered, we include the jar files whose names contain the names of these applications (e.g., *jetty*.jar for Jetty) and make them available under the process-dir option supported by Soot. For Eclipse4, we use org.eclipse.core.runtime.adaptor.EclipseStarter to let Soot locate all the other jar files used.

We manually inspect the reflection usage in a program in a demand-driven manner, starting from its reflective-action methods, assisted by Open Call Hierarchy in Eclipse, by following their backward slices. For a total of 609 reflective-action call sites examined, 510 call sites for calling class-retrieving methods and 304 call sites for calling member-retrieving methods are tracked and studied. As a result, a total of 1,423 reflective call sites, together with some nearby statements, are examined in our study.

2.3.3 Results. Below we describe our seven findings on reflection usage as our answers to the seven focus questions listed in Section 2.3.1, respectively. We summarize our findings as individual remarks, which are expected to be helpful in guiding the development of practical reflection analysis techniques and tools in future research.

Q1. String Constants and String Manipulations. In class-retrieving methods, Class.forName() and loadClass() each have a String parameter to specify the target class. In member-retrieving methods, getDeclaredMethod(String,...) and getMethod(String,...) each return a Method
Fig. 5. Classification of the String arguments of two class-retrieving methods, forName() and loadClass(), and four member-retrieving methods, getMethod(), getDeclaredMethod(), getField() and getDeclaredField.

As shown in Figure 5, string constants are commonly used when calling the two class-retrieving methods (34.7% on average) and the four member-retrieving methods (63.1% on average). In the presence of string manipulations, many class/method/field names are unknown exactly. This is mainly because their static resolution requires precise handling of many different operations e.g., subString() and append(). In fact, many cases are rather complex and thus cannot be handled well by simply modeling the java.lang.String-related API. Thus, SOLAR does not currently handle string manipulations. However, the incomplete information about class/method/field names (i.e., partial string information) can be exploited beneficially [22, 51].

We also found that many string arguments are Unknown (55.3% for calling class-retrieving methods and 25.1% for calling member-retrieving methods, on average). These are the strings that may be read from, say, configuration files, command lines, or even Internet URLs. Finally, string constants are found to be more frequently used for calling the four member-retrieving methods than the two class-retrieving methods: 146 calls to getDeclaredMethod() and getMethod(), 27 calls to getDeclaredField() and getField() in contrast with 98 calls to forName() and loadClass(). This suggests that the analyses that ignore string constants flowing into some member-retrieving methods may fail to exploit such valuable information and thus become imprecise.

Remark 1. Resolving reflective targets by string constants does not always work. On average, only 49% reflective call sites (where string arguments are used to specify reflective targets) use string constants. In addition, fully resolving non-constant string arguments by string manipulation, although mentioned elsewhere [5, 38], may be hard to achieve, in practice.

Q2. Retrieving an Array of Member Objects. As introduced in Section 2.2.2, half of member-retrieving methods (e.g., getMethods()) return an array of member metaobjects. Although not as frequently used as the ones returning single member metaobject (e.g., getMethod()), they play an important role in introducing new program behaviours in some applications. For example, in the two Eclipse programs studied, there are four invoke() call sites called on an array of Method
Fig. 6. The percentage frequency distribution of reflective-action call sites on instance and static members.

objects returned from getMethods() and 15 fld. get() and f1d. set() call sites called on an array of Field objects returned by getDeclaredFields(). Through these calls, dozens of methods are invoked and hundreds of fields are modified reflectively. Ignoring such methods as in prior work [38] and tools (Bddbddb, WALA, SOOT) may lead to significantly missed program behaviours by the analysis.

**Remark 2.** In member-retrieving methods, get(Declared)Methods/Fields/Constructors(), which return an array of member metaobjects, are usually ignored by most of existing reflection analysis tools. However, they play an important role in certain applications for both method invocations and field manipulations.

**Q3. Static or Instance Members.** In the literature on reflection analysis [32, 38, 51, 68], reflective targets are mostly assumed to be instance members. Accordingly, calls to the reflective-action methods such as invoke(), get() and set(), are usually considered as virtual calls, instance field accesses, and instance field modifications, respectively (see Table 1 for details). However, in real programs, as shown in Figure 6, on average, 37% of the invoke() call sites are found to invoke static methods and 50% of the get()/set() call sites are found to access/modify static fields. Thus in practice, reflection analysis should distinguish both cases and also be aware of whether a reflective target is a static or instance member, since the approaches for resolving both cases are usually different.

**Remark 3.** Static methods/fields are invoked/accessed nearly as frequently as instance methods/-fields in Java reflection, even though the latter has received more attention in the literature. In practice, reflection analysis should distinguish the two cases and adopt appropriate approaches for handling them.

**Q4. Resolving newInstance() by Casts.** In Figure 3, when cName is not a string constant, the (dynamic) type of obj created by newInstance() in line 4 is unknown. For this case, Livshits et al. [38] propose to infer the type of obj by leveraging the cast operation that post-dominates intra-procedurally the newInstance() call site. If the cast type is A, the type of obj must be A or
one of its subtypes assuming that the cast operation does not throw any exceptions. This approach has been implemented in many analysis tools such as Wala, Bddbddb and Elf.

However, as shown in Figure 7, exploiting casts this way does not always work. On average, 28% of newInstance() call sites (obtained by manually inspecting all the related reflective code) have no such intra-procedural post-dominating casts. As newInstance() is the most widely used reflective-action method (see Q6), its unresolved call sites may significantly affect the soundness of the analysis, as discussed in Section 7.5.1. Hence, we need a better solution to handle newInstance().

![Figure 7. newInstance() resolution by leveraging intra-procedural post-dominating casts.](image)

**Remark 4.** Resolving newInstance() calls by leveraging their intra-procedural post-dominating cast operations fails to work for 28% of the newInstance() call sites found. As newInstance() affects critically the soundness of reflection analysis (Remark 6), a more effective approach for its resolution is required.

**Q5. Class-Retrieving Methods.** Figure 8 shows the percentage frequency distribution of eight class-retrieving methods. “Unknown” is included since we failed to find the class-retrieving methods for some reflective-action calls (e.g., invoke()) even by using Eclipse’s Open Call Hierarchy tool. For the first 12 programs, the six class-retrieving methods as shown (excluding “Unknown” and “Others”) are the only ones leading to reflective-action calls. For the last two, Jetty and Tomcat, “Others” stands for defineClass() in ClassLoader and getParameterTypes() in Method. Finally, getComponentType() is usually used in the form of getClass().getComponentType() for creating a Class object argument for Array.newInstance().

On average, Class.forName(), .class, getClass() and loadClass() are the top four most frequently used (48.1%, 18.0%, 17.0% and 9.7%, respectively). A class loading strategy can be configured in forName() and loadClass(). In practice, forName() is often used by the system class loader and loadClass() is usually overwritten in customer class loaders, especially in framework applications such as Tomcat and Jetty.

**Remark 5.** Reflection analysis should handle Class.forName(), getClass(), .class, and loadClass(), which are the four major class-retrieving methods for creating Class objects.
In addition, `getComponentType()` should also be modeled if `Array-related reflective-action methods are analyzed, as they are usually used together.

**Q6. Reflective-Action Methods.** Figure 9 depicts the percentage frequency distribution of all the nine reflective-action methods in all the programs studied. We can see that `newInstance()` and `invoke()` are the ones that are most frequently used (46.3% and 32.7%, respectively, on average). Both of them are handled by existing static analysis tools such as Doop, Soot, WALA and Bddbddb.

However, Field- and Array-related reflective-action methods, which are also used in many programs, are ignored by most of these tools. Their handling is often necessary. For example, Eclipse (org.eclipse.osgi.util.NLS) uses `Field.set()` to initialize a large number of (non-primitive) fields of all given classes. Some JDK code (e.g., `java.util.AbstractCollection`) uses `Array.newInstance()` to reflectively create a new non-primitive array whose type depends on the given argument.

As far as we know, Field- and Array-related reflective-action methods are handled only by ELF [32], SOLAR [33] and DOOP [51].

**Remark 6.** Reflection analysis should at least handle `newInstance()` and `invoke()` as they are the most frequently used reflective-action methods (79% on average), which will significantly affect a program’s behavior, in general; otherwise, much of the codebase may be invisible for analysis. Effective reflection analysis should also consider Field- and Array-related reflective-action methods, as they are also commonly used.
Q7. Self-Inferencing Property. As illustrated by the program given in Figure 3, the names of its reflective targets are specified by the *string arguments* (e.g., cName, mName and fName) at the class-retrieving and member-retrieving reflective calls. Therefore, string analysis has been a popular approach for static reflection analysis in the last decade. However, if the value of a string is unknown statically (e.g., read from external files or command lines), then the related reflective calls, including those to newInstance(), may have to be ignored, rendering the corresponding codebase or operations invisible to the analysis (note that conservatively estimating those unresolved reflective calls to have any effect would cause to invoke any methods, making the analysis too imprecise to be scalable). To improve precision, in this case, the last resort is to exploit the existence of some intra-procedurally post-dominating cast operations on a call to newInstance() in order to deduce the types of objects reflectively created (Q4).

However, in our study, we find that there are many other rich hints about the behaviors of reflective calls at their usage sites. Such hints can be and should be exploited to make reflection analysis more effective, even when some string values are partially or fully unknown. In the following, we first look at three real example programs to examine what these hints are and expose a so-called *self-inferencing property* inherent in these hints. Finally, we explain why self-inferencing property is naturally inherent in most Java reflection code and discuss its potential in making reflection analysis more effective.

**Example 2.1 (Reflective Method Invocation (Figure 10)).** The method name (the first argument of `getMethod()` in line 174) is statically unknown as part of it is read from command line `cmd`. However, the target method (represented by `method`) can be deduced from the second argument (`parameters`) of the corresponding reflective-action call `invoke()` in line 175. Here, `parameters` is an array of objects, with only one element (line 155). By querying the pointer analysis and also leveraging the type information in the program, we know that the type of the object pointed to by this is `FrameworkCommandInterpreter`, which has no subtypes. As a result, we can infer that
Application: Eclipse (v4.2.2)
Class: org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter

```java
123  public Object execute(String cmd) {
155    Object[] parameters = new Object[] {this};
167    for (int i = 0; i < size; i++) {
174      method = target.getClass().getMethod("_", cmd, parameterTypes);
175      retval = method.invoke(target, parameters); ...
228  }
```

Fig. 10. Self-inferencing property for a reflective method invocation, deduced from the number and dynamic types of the components of the one-dimensional array argument, parameters, at a `invoke()` call site.

the descriptor of the target method in line 175 must have only one argument and its actual type must be `FrameworkCommandInterpreter` or one of its supertypes.

Application: Eclipse (v4.2.2)
Class: org.eclipse.osgi.framework.internal.core.Framework

```java
1652  public static Field getField(Class clazz, ...) {
1653    Field[] fields = clazz.getDeclaredFields(); ...
1654    for (int i = 0; i < fields.length; i++) {
1658      return fields[i]; } ...
1662  }
1682  private static void forceContentHandlerFactory(...) {
1683    Field factoryField = getField(URLConnection.class, ...);
1687    java.net.ContentHandlerFactory factory =
1689      (java.net.ContentHandlerFactory) factoryField.get(null); ...
1709  }
```

Fig. 11. Self-inferencing property for a reflective field access, deduced from the type casted on the returned value of, and the `null` argument used at, a `get()` call site.

Example 2.2 (Reflective Field Access (Figure 11)). In line 1683, `factoryField` is obtained as a `Field` object from an array of `Field` objects created in line 1653 for all the fields in `URLConnection`. In line 1687, the object returned from `get()` is cast to `java.net.ContentHandlerFactory`. Based on its cast operation and `null` argument, we know that the call to `get()` may only access the `static` fields of `URLConnection` with the type `java.net.ContentHandlerFactory`, its supertypes or its subtypes. Without the self-inferencing property at the `get()` call site, all the fields in `URLConnection` must be assumed to be accessed conservatively by an analysis.

Example 2.3 (Reflective Field Modification (Figure 12)). Like the case in Figure 11, the field object in line 290 is also read from an array of field objects created in line 302. This code pattern appears one more time in line 432 in the same class, i.e., `org.eclipse.osgi.util.NLS` (not shown here). According to the two arguments, `null` and `value`, provided at `set()` (line 290), we can deduce
That the target field (to be modified in line 290) is static (from null) and its declared type must be \texttt{java.lang.String} or one of its supertypes (from the type of value).

**Definition 2.4 (Self-Inferencing Property).** For each reflective-action call site, its self-inferencing property comprises the information that can be used to infer its reflective targets, which consists of (1) all the information of its arguments (including receiver object), namely the number of arguments, their types, and (2) the possible downcasts on its returned values, and (3) the possible string values statically resolved at its corresponding class-retrieving and member-retrieving call sites.

We argue that the self-inferencing property is inherent in most Java reflection code due to the characteristics of object-oriented programming and the Java reflection API. For example, the declared type of the object reflectively returned by \texttt{get()} and \texttt{invoke()} or created by \texttt{newInstance()} is always \texttt{java.lang.Object}. Therefore, the returned object must be first cast to a specific type before it is used as a regular object, except when its dynamic type is \texttt{java.lang.Object} or it will be used only as an receiver for the methods inherited from \texttt{java.lang.Object}; otherwise, the compilation would fail. As another example, the descriptor of a target method reflectively called at \texttt{invoke()} must be consistent with what is specified by its second argument (e.g., parameters in line 175 of Figure 10); otherwise, exceptions would be thrown at runtime. These constraints should be exploited to enable resolving reflection in a disciplined way.

The self-inferencing property not only helps resolve reflective calls more effectively when the values of string arguments are partially known (e.g., when either a class name or a member name is known), but also provides an opportunity to resolve some reflective calls even if the string values are fully unknown. For example, in some Android apps, class and method names for reflective calls are encrypted for benign or malicious obfuscation, which "makes it impossible for any static analysis to recover the reflective call" [48]. However, this appears to be too pessimistic in our setting, because, in addition to the string values, some other self-inferencing hints are possibly available to facilitate reflection resolution. For example, given \texttt{(A)\texttt{invoke(o, \ldots)}}), the class type of the target method can be inferred from the dynamic type of \texttt{o} (by pointer analysis), and the declared return type and descriptor of the target method can also be deduced from \texttt{A} and \texttt{\ldots}, respectively, as discussed above.
Remark 7. Self-inferencing property is inherent in most Java reflection code, but has not been fully exploited in analyzing reflection before. We will show how this property can be leveraged in different ways (for analyzing different kinds of reflective methods as shown in Sections 4.2 and 4.3) to make reflection analysis significantly more effective.

3 OVERVIEW OF SOLAR

We first introduce the design goal of, challenges faced by, and insights behind Solar in Section 3.1. We then present an overview of the Solar framework including its basic working mechanism and the functionalities of its components in Section 3.2.

3.1 Goals, Challenges and Insights

Design Goals. As already discussed in Section 1.3, Solar is designed to resolve reflection as soundly as possible (i.e., more soundly or even soundly when some reasonable assumptions are met) and accurately identify the reflective calls that are resolved unsoundly.

Challenges. In addition to the challenges described in Section 1.1, we must also address another critical problem: it is hard to reason about the soundness of Solar and identify accurately which parts of the reflective code have been resolved unsoundly.

If one target method at one reflective call is missed by the analysis, it may be possible to identify the statements that are unaffected and thus still handled soundly. However, the situation will deteriorate sharply if many reflective calls are resolved unsoundly. In the worst case, all the other statements in the program may be handled unsoundly. To play safe, the behaviors of all statements must be assumed to be under-approximated in the analysis, as we do not know which value at which statement has been affected by the unsoundly resolved reflective calls.

Let us consider the program in Figure 10 as an example. If the invoke() call site (at line 175) cannot be resolved soundly, many runtime behaviors will be missed by static analysis. When its target methods are invoked reflectively, the objects pointed to by parameters, target and retval could all be modified, affecting potentially the other parts of the program, which may also finally change the value of target and its associated reflective behaviors. If there are many such unsoundly resolved reflective calls distributed in different parts of the program, then more unsoundness will end up being “propagated” throughout the program, making the analysis hard to know which parts of the program have been actually soundly analyzed.

Insights. To achieve the design goals of Solar, we first need to ensure that as few reflective calls are resolved unsoundly as possible. This will reduce the propagation of unsoundness to as few statements as possible in the program. As a result, if Solar reports that some analysis results are sound (unsound), then they will be likely sound (unsound). This is the key to enabling Solar to achieve practical precision in terms of both soundness reasoning and unsoundness identification.

To resolve most or even all reflective calls soundly, Solar needs to maximally leverage the available information (the string values at reflective calls are inadequate as they are often unknown statically) in the program to help resolve reflection. Meanwhile, Solar should resolve reflection precisely. Otherwise, Solar may be unscalable due to too many false reflective targets introduced. In Sections 4.2 and 4.3, we will describe how Solar leverages the self-inferencing property in a program (Definition 2.4) to analyze reflection with good soundness and precision.

Finally, Solar should be aware of the conditions under which a reflective target cannot be resolved. In other words, we need to formulate a set of soundness criteria for different reflection methods based on different resolution strategies adopted. If the set of criteria is not satisfied, Solar
can mark the corresponding reflective calls as the ones that are resolved unsoundly. Otherwise, Solar can guarantee the soundness of the reflection analysis under some reasonable assumptions (Section 4.1).

3.2 The Solar Framework

Figure 13 gives an overview of Solar. Solar consists of four core components: an inference engine for discovering reflective targets, an interpreter for soundness and precision, a locator for unsound and imprecise calls, and a Probe (a lightweight version of Solar). In the rest of this section, we first introduce the basic working mechanism of Solar and then briefly explain the functionality of each of its components.

3.2.1 Working Mechanism. Given a Java program, the inference engine resolves and infers the reflective targets that are invoked or accessed at all reflective-action method call sites in the program, as soundly as possible. There are two possible outcomes. If the reflection resolution is scalable (under a given time budget), the interpreter will proceed to assess the quality of the reflection resolution under soundness and precision criteria. Otherwise, Probe, a lightweight version of Solar, would be called upon to analyze the same program again. As Probe resolves reflection less soundly but much more precisely than Solar, its scalability can be usually guaranteed. We envisage providing a range of Probe variants with different trade-offs among soundness, precision and scalability, so that the scalability of Probe can be always guaranteed.

If the interpreter confirms that the soundness criteria are satisfied, Solar reports that the reflection analysis is sound. Otherwise, the locator will be in action to identify which reflective calls are resolved unsoundly. In both cases, the interpreter will also report the reflective calls that are resolved unsoundly and (ranked) imprecise reflective calls & The related calls with hints for the reflective targets.

Fig. 13. Overview of Solar.
imprecisely if the precision criteria are violated. This allows potential precision improvements to be made for the analysis.

The locator not only outputs the list of reflective calls that are resolved unsoundly or imprecisely in the program but also pinpoints the related class-retrieving and member-retrieving method calls of these "problematic" calls, which contain the hints to guide users to add annotations, if possible. Figure 14 depicts an example output.

As will be demonstrated in Section 7, for many programs, SOLAR is able to resolve reflection soundly under some reasonable assumptions. However, for certain programs, like other existing reflection analyses, SOLAR is unscalable. In this case, PROBE (a lightweight version of SOLAR whose scalability can be guaranteed as explained above) is applied to analyze the same program. Note that PROBE is also able to identify the reflective calls that are resolved unsoundly or imprecisely in the same way as SOLAR. Thus, with some unsound or imprecise reflective calls identified by PROBE and annotated by users, SOLAR will re-analyze the program, scalably after one or more iterations of this "probing" process. As discussed in Section 7, the number of such iterations is usually small, e.g., only one is required for most of the programs evaluated.

For some programs, users may choose not to add annotations to facilitate reflection analysis. Even in this case, users can still benefit from the SOLAR approach, for two reasons. First, PROBE is already capable of producing good-quality reflection analysis results, more soundly than string analysis. Second, users can understand the quality of these results by inspecting the locator’s output, as discussed in Section 1.2.

3.2.2 Basic Components. Their functionalities are briefly explained below.

Reflective Target Inference Engine. We employ two techniques to discover reflective targets: collective inference for resolving reflective method invocations (invoke()) and field accesses/modifications (get() / set()) and lazy heap modeling for handling reflective object creation (newInstance()).
Both techniques exploit the self-inferencing property found in our reflection usage study (Section 2.3.3) to resolve reflection in a disciplined manner with good soundness and precision. We will explain their approaches in Sections 4.2 and 4.3, respectively, and further formalize them in Section 5.4.

**Soundness and Precision Interpreter.** Solar currently adopts a simple but practical scheme to measure precision, in terms of the number of targets resolved at a reflective-action call site. Solar allows users to specify a threshold value in advance to define the imprecision that can be tolerated for each kind of reflective-action calls, forming its precision criteria. Suppose 100 is given as the threshold value for `invoke()`. An `invoke()` call site will be reported as being imprecisely resolved if more than 100 targets have been found at the call site. If no threshold is provided, Solar will report all the reflective-action sites (with their resolved targets), starting from the most imprecise one to the most precise one, and let the users decide which ones are resolved imprecisely.

The soundness criteria are formulated in Section 5.5 in terms of conditions under which various inference rules (adopted by the inference engine) can be applied soundly.

**Unsound and Imprecise Call Locator.** A reflective-action reflective call is identified as being imprecisely resolved if the number of resolved targets is higher than permitted by its corresponding precision criterion. Similarly, a reflective-action reflective call is marked as being unsoundly resolved if its corresponding soundness criterion is violated.

To facilitate user annotations for an imprecisely or unsoundly resolved reflective-action reflective call, the locator also pinpoints its corresponding class-retrieving and member-retrieving method call sites. It can be difficult to understand the semantics of a reflective-action call by reading just the code at its vicinity. Often, more hints about its semantics are available at or around its class-retrieving and member-retrieving method call sites, which may reside in different methods or even classes in the program.

**Example 3.1.** Figure 14 illustrates Solar’s output for a real program. In the figure, the output is shown at the left-hand side and the related reflective code about the unsoundly resolved `invoke()` call is shown at the right-hand side. In the “Unsound List” in the output, the `invoke()` (reflective-action method) call site in method `getValue` is the unsoundly resolved call identified. Its class-retrieving method (`forName()`) and member-retrieving method (`getMethods()`) call sites, which are located in the constructor of class `org.hsqldb.Function`, are also highlighted. At the right-hand side of the figure, we can see that the hints for annotations are available around the class-retrieving and member-retrieving call sites (e.g., lines 169, 184 and 185) rather than the reflective-action call site (line 352). Based on those hints, users can easily annotate for this unsoundly resolved `invoke()` by, e.g., finding out the values of `cn` (line 169) and `mn` (line 185).

We will further explain how Solar identifies unsoundly resolved reflective calls in Section 4.4 and how users are guided to add annotations in Section 4.5.

**Probe.** Probe is a lightweight version of Solar that weakens the power of its inference engine. Probe changes its inference strategies in both collective inference and lazy heap modeling, by resolving reflection more precisely but less soundly. Thus, the scalability of Probe can be usually guaranteed as fewer false reflective targets are introduced. We will formalize Probe based on the formalism of Solar in Section 5.7.

4 THE SOLAR METHODOLOGY

We first define precisely a set of assumptions made (Section 4.1). Then we examine the methodologies of collective inference (Section 4.2) and lazy heap modeling (Section 4.3) used in Solar’s inference
engine. Finally, we explain how SOLAR identifies unsoundly resolved reflective calls (Section 4.4) and how doing so helps guide users to add lightweight annotations to facilitate a subsequent reflection analysis (Section 4.5).

4.1 Assumptions
There are four reasonable assumptions. The first one is commonly made on static analysis [55] and the next two are made previously on reflection analysis for Java [38]. SOLAR adds one more assumption to allow reflective allocation sites to be modeled lazily. Under the four assumptions, it becomes possible to reason about the soundness and imprecision of SOLAR.

Assumption 1 (Closed-World). Only the classes reachable from the class path at analysis time can be used during program execution.

This assumption is reasonable since we cannot expect static analysis to handle all classes that a program may download from the Internet and load at runtime. In addition, Java native methods are excluded as well.

Assumption 2 (Well-Behaved Class Loaders). The name of the class returned by a call to Class.forName(cName) equals cName.

This assumption says that the class to be loaded by Class.forName(cName) is the expected one specified by the value of cName, thus avoiding handling the situation where a different class is loaded by, e.g., a malicious custom class loader. How to handle custom class loader statically is still an open hard problem. Note that this assumption also applies to loadClass(), another class-retrieving method shown in Figure 4.

Assumption 3 (Correct Casts). Type cast operations applied to the results of calls to reflective-action methods are correct, without throwing a ClassCastException.

This assumption has been recently demonstrated as practically valid through extensive experiments in [26].

Assumption 4 (Object Reachability). Every object o created reflectively in a call to newInstance() flows into (i.e., will be used in) either (1) a type cast operation . . . = (T) v or (2) a call to a reflective-action method, get(v), set(v, . . .) or invoke(v, . . .), where v points to o, along every execution path in the program.

Cases (1) and (2) represent two kinds of usage points at which the class types of object o will be inferred lazily. Specifically, case (1) indicates that o is used as a regular object, and case (2) says that o is used reflectively, i.e., flows to the first argument of different reflective-action calls as a receiver object. This assumption does not cover only one rare situation where o is created but never used later. As validated in Section 7.2, Assumption 4 is found to hold for almost all reflective allocation sites in the real code.

4.2 Collective Inference
Figure 15 gives an overview of collective inference for handling reflective method invocations and field accesses/modifications. Essentially, we see how the reflective-action method calls invoke(), get() and set() are resolved. A Class object C is first created for the target class named cName. Then a Method (Field) object M (F) representing the target method (field) named mName (fName) in the target class of C is created. Finally, at some reflective call sites, e.g., invoke(), get() and set(), the target method (field) is invoked (accessed) on the target object o, with the arguments, . . . or t.
Solar works as part of a pointer analysis, with each being both the producer and consumer of the other. By exploiting the self-inferencing property (Definition 2.4) inherent in the reflective code, Solar employs the following two component analyses:

**Target Propagation (Marked by Solid Arrows)** Solar resolves the targets (methods or fields) of reflective calls, `invoke()`, `get()` and `set()`, by propagating the names of their target classes and methods/fields (e.g., those pointed by `cName`, `mName` and `fName` if statically known) along the solid lines into the points symbolized by circles.

**Target Inference (Marked by Dashed Arrows)** By using Target Propagation alone, a target member name (blue circle) or its target class type (red circle) at a reflective call site may be missing, i.e., unknown, due to the presence of input-dependent strings (Figure 5). If the target class type (red circle) is missing, Solar will infer it from the dynamic type(s) of the target object `o` (obtained by pointer analysis) at `invoke()`, `get()` or `set()` (when `o != null`). If the target member name (blue circle) is missing, Solar will infer it from (1) the dynamic types of the arguments of the target call, e.g., ... of `invoke()` and `t` of `set()`, and/or (2) the downcast on the result of the call, such as `(A)` at `invoke()` and `get()`.

**Example 4.1.** Let us illustrate Target Inference by considering `t = (A) f1.get(o)` in Figure 15. If a target field name `fName` is known but its target class type (i.e., red circle of `f1`) is missing, we can infer it from the types of all pointed-to objects `o’` by `o`. If `B` is one such a type, then a potential target class of `o` is `B` or any of its supertypes. If the target class type of `f1` is `B` but a potential target field name (i.e., blue circle of `f1`) is missing, we can deduce it from the downcast `(A)` to resolve the call to `t = o.f`, where `f` is a member field in `B` whose type is `A` or a supertype or subtype of `A`. A supertype is possible because a field (whose declared type is this supertype) may point to an object of type `A` or a subtype of `A`.

In Figure 15, if `getMethods()` or `getFields()` is called as a member-retrieving method instead, then an array of `Method` or `Field` objects will be returned so that Target Propagation from it is implicitly performed by pointer analysis. All the other methods in `Class` for introspecting methods/fields/constructors are handled similarly.
Resolution Principles. To balance soundness, precision and scalability in a disciplined manner, collective inference resolves the targets at a reflective-action method call site (Figure 15) if and only if one of the following three conditions is met:

- Both its target class type (red circle) and target member name (blue circle) are made available by target propagation (solid arrow) or target inference (dashed arrow).
- Only its target class type (red circle) is made available by target propagation (solid arrow) or target inference (dashed arrow).
- Only its target member name (blue circle) is made available by both target propagation (solid arrow) and target inference (dashed arrow).

In practice, the first condition is met by many calls to `invoke()`, `get()` and `set()`. In this case, the number of spurious targets introduced can be significantly reduced due to the simultaneous enforcement of two constraints (the red and blue circles).

To increase the inference power of Solar, as explained in Section 3.1, we will also resolve a reflective-action call under one of the other two conditions (i.e., when only one circle is available). The second condition requires only its target class type to be inferable, as a class (type) name that is prefixed with its package name is usually unique. However, when only its target member name (blue circle) is inferable, we insist both its name (solid arrow) and its descriptor (dashed arrow) are available. In a large program, many unrelated classes may happen to use the same method name. Just relying only on the name of a method in the last condition may cause imprecision.

If a reflective-action call does not satisfy any of the above three conditions, then Solar will flag it as being unsoundly resolved, as described in Section 4.4.

4.3 Lazy Heap Modeling

As shown in Section 2.3.3, reflective object creation, i.e., `newInstance()` is the most widely used reflective-action method. Lazy heap modeling (LHM), illustrated in Figure 16, is developed to facilitate its target inference and the soundness reasoning for Solar.

There are three cases. Let us consider Cases (II) and (III) first (to ease understanding, we discuss Case (I) after explaining these two cases). Usually, an object, say o, created by `newInstance()` will be used later either regularly or reflectively as shown in Cases (II) and (III), respectively. In Case (II), since the declared type of o is `java.lang.Object`, o is first cast to a specific type before used for calling methods or accessing fields as a regular object. Thus, o will flow to some cast operations. In Case (III), o is used in a reflective way, i.e., as the first argument of a call to a reflective-action method, `invoke()`, `get()` or `set()`, on which the target method (field) is called (accessed). This appears to be especially commonly used in Android apps.

For these two cases, we can leverage the information at o’s usage sites to infer its type lazily and also make its corresponding effects (on static analysis) visible there. As for the (regular) side effect that may be made by o along the paths from `newInstance()` call site to its usage sites, we use Case (I) to cover this situation.

Now, we examine these Cases (I) – (III), which are highlighted in Figure 16, one by one, in more detail. If cName at c = `Class.forName(cName)` is unknown, Solar will create a `Class` object c” that represents this unknown class and assign it to c. On discovering that c1 points to a c” at an allocation site i (v = c1.newInstance()), Solar will create an abstract object o”i of an unknown type for the site to mark it as being unresolved so far. Subsequently, o”i will flow into Cases (I) – (III).

In Case (I), the returned type of o”i is declared as `java.lang.Object`. Before o”i flows to a cast operation, the only side effect that can be made by this object is to call some methods declared in
java.lang.Object. In terms of reflection analysis, only the two pointer-affecting methods shown in Figure 16 need to be considered. SOLAR handles both soundly, by returning (1) an unknown string for `v1.toString()` and (2) an unknown `Class` object for `v1.getClass()`. Note that `clone()` cannot be called on `v1` of type `java.lang.Object` (without a downcast being performed on `v1` first).

Let us consider Cases (II) and (III), where each statement, say $S_x$, is called an LHM point, containing a variable $x$ into which $o_u$ flows. In Figure 16, we have $x \in \{v2, v3, v4\}$. Let $lhms(S_x)$ be the set of class types discovered for the unknown class $u$ at $S_x$ by inferring from the cast operation at $S_x$ as in Case (II) or the information available at a call to (C) `m1.invoke(v4, args)` (e.g., on $C$, $m1$ and $args$) as in Case (III). For example, given $S2_{v2}$: $A a = (A) v2$, $lhms(S2_{v2})$ contains $A$ and its subtypes. To account for the side effect of $v = c1.newInstance()$ at $S_x$ lazily, we add (conceptually) a statement, $x = new T()$, for every $T \in lhms(S_x)$, before $S_x$. Thus, $o_u$ is finally split into and thus aliased with $n$ distinct abstract objects, $o_1^{T_1},..., o_n^{T_n}$, where $lhms(S_x) = \{T_1, ..., T_n\}$, such that $x$ will be made to point to all these new abstract objects.

Figure 16 illustrates lazy heap modeling for the case when neither $A$ nor $B$ has subtypes and the declaring class for $m1$ is discovered to be $D$ (i.e., $c2$ in Figure 16 represents class $D$), which has one subtype $B$. Thus, SOLAR will deduce that $lhms(S2_{v2}) = \{A\}$, $lhms(S3_{v3}) = \{B\}$ and $lhms(S4_{v4}) = \{B, D\}$. Note that in Case (II), $o_u$ will not flow to $a$ and $b$ due to the cast operations.

As `java.lang.Object` contains no fields, all field accesses to $o_u$ will only be made on its lazily created objects. Therefore, if the same concrete object represented by $o_u$ flows to both $S_{x1}$ and $S_{x2}$,
then \( \text{lh}(S_{x_1}) \cap \text{lh}(S_{x_2}) \neq \emptyset \). To handle the alias information soundly, for each type inferred at LHM points w.r.t. a given newInstance() call site, we only create one object lazily. This implies that \( x_1 \) and \( x_2 \) will point to a common object lazily created. For example, in Figure 16, \( v_3 \) and \( v_4 \) points to \( o_{B_i} \) since \( \text{lh}(S_{v_3}) \cap \text{lh}(S_{v_4}) = \{o_{B_i}^u\} \). As a result, the alias relation between \( x_1.f \) and \( x_2.f \) is correctly maintained, where \( f \) is a field of \( o_{B_i}^u \).

### Example 4.2

In Figure 17, SOLAR will model the newInstance() call in line 3 lazily (as \( \text{cName1} \) in line 10, then \( \text{cName} \) in line 2, are statically unknown) by returning an object \( o_{B_i}^u \) of an unknown type \( u \). Note that \( o_{B_i}^u \) flows into two kinds of usage points: the cast operation in line 12 and the invoke() call in line 15. In the former case, SOLAR will infer \( u \) to be \( A \) and its subtypes in line 12. In the latter case, SOLAR will infer \( u \) based on the information available in line 15 by distinguishing three cases. (1) If \( \text{cName2} \) is known, then SOLAR deduces \( u \) from the known class in \( \text{cName2} \). (2) If \( \text{cName2} \) is unknown but \( \text{mName2} \) is known, then SOLAR deduces \( u \) from the known method name in \( \text{mName2} \) and the second argument new Object[] \{b, c\} of the invoke() call site. (3) If both \( \text{cName2} \) and \( \text{mName2} \) are unknown (given that the types of \( o_{B_i}^u \) are already unknown), then SOLAR will flag the invoke() call in line 15 as being unsoundly resolved, detected automatically by verifying one of the soundness criteria, i.e., Condition (4) in Section 5.5.

**Discussion.** Under Assumption 4, we need only to handle the three cases in Figure 16 to establish whether a newInstance() call has been modeled soundly or not. The rare exception (which breaks Assumption 4) is that \( o_{B_i}^u \) is created but never used later (where no hints are available). To achieve soundness in this rare case, the corresponding constructor (of the dynamic type of \( o_{B_i}^u \)) must be annotated to be analyzed statically unless ignoring it will not affect the points-to information to be obtained. Again, as validated in Section 7.2, Assumption 4 is found to be very practical.

### 4.4 Unsound Call Identification

Intuitively, we mark a reflective-action reflective call as being unsoundly resolved when SOLAR has exhausted all its inference strategies to resolve it, but to no avail. In addition to Case (3) in Example 4.2, let us consider another case in Figure 16, except that \( c2 \) and \( \text{mName} \) are assumed to be unknown. Then \( m1 \) at \( s4 \): \( m1.\text{invoke}(v_4, \arg) \) will be unknown. SOLAR will mark it as unsoundly resolved, since just leveraging \( \arg \) alone to infer its target methods may cause SOLAR to be too imprecise to scale (Section 4.2).

The formal soundness criteria that are used to identify unsoundly resolved reflective calls are defined and illustrated in Section 5.5.
4.5 Guided Lightweight Annotation

As shown in Example 3.1, SOLAR can guide users to the program points where hints for annotations are potentially useful for unsoundly or imprecisely resolved reflective calls. As these "problematic" call sites are the places in a program where reflective-action methods are invoked, we can hardly extract the information there to know the names of the reflective targets, as they are specified at the corresponding class-retrieving and member-retrieving call sites (also called the annotation sites), which may not appear in the same method or class (as the "problematic call sites"). Thus, SOLAR is designed to automatically track the flows of metaobjects from the identified "problematic" call sites in a demand-driven way to locate all the related annotation sites.

In SOLAR, annotations are only added for the unsoundly resolved reflective-action call sites which are few and identified accurately. As a result, the number of required annotations (for achieving soundness) is significantly less than that required in [38], which simply asks for annotations when the string argument of a reflective call is statically unknown. This is further validated in Section 7.3.

5 FORMALISM

We formalize SOLAR, as illustrated in Figure 13, for RefJava, which is Java restricted to a core subset of its reflection API. SOLAR is flow-insensitive but context-sensitive. However, our formalization is context-insensitive for simplicity. We first define RefJava (Section 5.1), give a road map for the formalism (Section 5.2) and present some notations used (Section 5.3). We then introduce a set of rules for formulating collective inference and lazy heap modeling in SOLAR’s inference engine (Section 5.4). Based on these rules, we formulate a set of soundness criteria (Section 5.5) that enables reasoning about the soundness of SOLAR (Section 5.6). Finally, we describe how to instantiate Probe from SOLAR (Section 5.7), and handle static class members (Section 5.8).

5.1 The RefJava Language

RefJava consists of all Java programs (under Assumptions 1 – 4) except that the Java reflection API is restricted to the seven core reflection methods: one class-retrieving method Class.forName(), two member-retrieving methods getMethod() and getFiedl(), and four reflective-action methods for reflective object creation newInstance(), reflective method invocation invoke(), reflective fields access get() and modification set().

The soundness of pointer analysis for non-reflective Java (defined in terms of the the statements in Figure 20) is well established in the literature [51, 57]. RefJava is an extension with the above seven reflective methods added.

Our formalism is designed to allow its straightforward generalization to the entire Java reflection API. As is standard, a Java program is represented only by five kinds of statements in the SSA form, as shown in Figure 20. For simplicity, we assume that all the members (fields or methods) of a class accessed reflectively are its instance members, i.e., o ≠ null in get(o), set(o, t) and invoke(o,. . . ) in Figure 15. We will formalize how to handle static members in Section 5.8.

5.2 Road Map

As depicted in Figure 18, SOLAR’s inference engine, which consists of five components, works together with a pointer analysis. The arrow ←→ between a component and the pointer analysis means that each is both a producer and consumer of the other.

Let us take an example to see how this road map works. Consider the reflective-action call t = f1.get(o) in Figure 15. If cName and fName are string constants, Propagation will create a Field object (pointed to by f1) carrying its known class and field information and pass it to Target Search (①). If cName or fName is not a constant, a Field object marked as such is created and passed to
Inference (2), which will infer the missing information and pass a freshly generated Field object enriched with the missing information to Target Search (3). Then Target Search maps a Field object to its reflective target $f$ in its declaring class (4). Finally, Transformation turns the reflective call $t = f1.get(o)$ into a regular statement $t = o.f$ and passes it to the pointer analysis (5). Note that Lazy Heap Modeling handles newInstance() based on the information discovered by Propagation (a) or Inference (b).

5.3 Notations

In this paper, a field signature consists of the field name and descriptor (i.e., field type), and a field is specified by its field signature and the class where it is defined (declared or inherited). Similarly, a method signature consists of the method name and descriptor (i.e., return type and parameter types) and, a method is specified by its method signature and the class where it is defined.

We will use the notations given in Figure 19. CO, FO and MO represent the set of Class, Field and Method objects, respectively. In particular, $c^t$ denotes a Class object of a known class $t$ and $c^u$ denotes a Class object of an unknown class $u$. As illustrated in Figure 16, we write $o^t_i$ to represent an abstract object created at an allocation site $i$ if it is an instance of a known class $t$ and $o^u_i$ of (an unknown class type) otherwise. For a Field object, we write $f^t_i$ if it is a field defined in a known class $t$ and $f^u_i$ otherwise, with its signature being $s$. In particular, we write $f^u_s$ for $f^s_u$ in the special case when $s$ is unknown, i.e., $s.tf = s.nzf = u$. Similarly, $m^t_s$, $m^u_s$, $m^t_u$ and $m^u_u$ are used to represent Method objects. We write $m^u_s$ for $m^t_s$ when $s$ is unknown (with the return type $s.tr$ being irrelevant, i.e., either known or unknown), i.e., $s.nzm = s.nzp = u$.

5.4 The Inference Engine of Solar

We present the inference rules used by all the components in Figure 18, starting with the pointer analysis and moving to the five components of Solar. Due to their cyclic dependencies, the reader is invited to read ahead sometimes, particularly to Section 5.4.6 on LHM, before returning back to the current topic.

5.4.1 Pointer Analysis. Figure 20 gives a standard formulation of a flow-insensitive Andersen’s pointer analysis for REFJAVA. There are five basic statements: new, assignment, load, store and...
two cases. If $i$ is used to resolve the virtual dispatch of method $o_i$ (unknown type) can only be made to the abstract objects of known types created lazily from $t_i$. At this allocation site, labeled by $x$, can be accessed. In Java, as explained in Section 4.3, the field accesses to $o_i$ can be made to the abstract objects of known types created lazily from $o_i$ at LHM points.

Method call, as shown in Figure 20. $pt(x)$ represents the points-to set of a pointer $x$. An array object is analyzed with its elements collapsed to a single field, denoted $arr$. For example, $x[i] = y$ can be seen as $x.arr = y$. In [A-New], $o_i^t$ uniquely identifies the abstract object created as an instance of $t$ at this allocation site, labeled by $i$. In [A-Ld] and [A-St], only the fields of an abstract object $o_i^t$ of a known type $t$ can be accessed. In Java, as explained in Section 4.3, the field accesses to $o_i^u$ (of an unknown type) can only be made to the abstract objects of known types created lazily from $o_i^u$ at LHM points.

\[
i : x = \text{new } t() \quad [\text{A-New}] \quad \frac{x = y f \ o_i^t \in pt(y)}{pt(o_i^t.f) \subseteq pt(x)} \quad [\text{A-Ld}] \quad \frac{x.f = y \ o_i^t \in pt(x)}{pt(y) \subseteq pt(o_i^t.f)} \quad [\text{A-St}]\]

\[
x = y.m(\text{arg}_1, \ldots, \text{arg}_n) \quad o_i^t \in pt(y) \quad m' = \text{dispatch}(o_i^t, m) \quad [\text{A-CALL}] \quad \frac{x \in pt(m'_{\text{this}})}{pt(m'_{\text{ret}}) \subseteq pt(x)} \quad \forall 1 \leq k \leq n : pt(\text{arg}_k) \subseteq pt(m'_{p_k})\]

Fig. 19. Notations. Here $\bar{X} = X \cup \{u\}$, where $u$ is an unknown class type or an unknown field/method signature. A superscript '*' marks a domain that contains $u$.

In [A-Call] (for non-reflective calls), like the one presented in [57], the function $\text{dispatch}(o_i^t, m)$ is used to resolve the virtual dispatch of method $m$ on the receiver object $o_i^t$ to be $m'$. There are two cases. If $o_i^t \in pt(y)$, we proceed normally as before. For $o_i^u \in pt(y)$, it suffices to restrict $m$ to \{toString(), getClass\}, as illustrated in Figure 16 and explained in Section 4.3. We assume

\[\text{toString()} \in \text{dispatch}(o_i^u, m)\]
that \( m' \) has a formal parameter \( m'_{hi} \) for the receiver object and \( m'_{p1}, \ldots, m'_{pn} \) for the remaining parameters, and a pseudo-variable \( m'_{ref} \) is used to hold the return value of \( m' \).

5.4.2 Propagation. Figure 21 gives the rules for handling forName(), getMethod() and getField() calls. Different kinds of Class, Method and Field objects are created depending on whether their string arguments are string constants or not. For these rules, \( \mathcal{SC} \) denotes a set of string constants and the function toClass creates a Class object \( c' \), where \( t \) is the class specified by the string value returned by \( \text{val}(o_i) \) (with \( \text{val} : \mathbb{H} \rightarrow \text{java.lang.String} \)).

\[
\begin{align*}
\text{Class} & \quad c = \text{Class.forName}(cName) \quad o_i^{\text{String}} \in \text{pt}(cName) \quad \text{[P-FORNAME]} \\
\text{Method} & \quad m = c'.\text{getMethod}(mName, \ldots) \quad o_i^{\text{String}} \in \text{pt}(mName) \quad c^- \in \text{pt}(c') \quad \text{[P-GETMTH]} \\
\text{Field} & \quad f = c'.\text{getField}(fName) \quad o_i^{\text{String}} \in \text{pt}(fName) \quad c^- \in \text{pt}(c') \quad \text{[P-GETFLD]}
\end{align*}
\]

By design, \( c', f'_s \) and \( m'_i \) will flow to Target Search but all the others, i.e., \( c^u, f'^u, f'_u, m^u \) and \( m'_u \) will flow to Inference, where the missing information is inferred. During Propagation, only the name of a method/field signature \( s \) (\( s.\text{n}_m \) or \( s.\text{n}_f \)) can be discovered but its other parts are unknown: \( s.\text{t}_r = s.\text{p} = s.\text{t}_f = u \).

5.4.3 Collective Inference. Figure 22 gives nine rules to infer reflective targets at the reflective-action calls \( x = (A) \text{.invoke}(y, \text{args}), x = (A) \text{.get}(y), f.\text{set}(y, x) \) where \( A \) indicates a post-dominating cast on their results. If \( A = \text{Object} \), then no such cast exists. These rules fall into three categories. In [I-InvTp], [I-GetTp] and [I-SetTp], we use the types of the objects pointed to by \( y \) to infer the class type of a method/field. In [I-InvSig], [I-GetSig] and [I-SetSig], we use the information available at a call site (excluding \( y \)) to infer the descriptor of a method/field signature. In [I-InvS2T], [I-GetS2T] and [I-SetS2T], we use a method/field signature to infer the class type of a method/field.

Some notations used are in order. As is standard, \( t :: t' \) holds when \( t \) is \( t' \) or a subtype of \( t' \). In [I-InvSig], [I-GetSig], [I-InvS2T] and [I-GetS2T], \( :: \) is used to take advantage of the post-dominating cast (\( A \)) during inference when \( A \) is not \( \text{Object} \). By definition, \( u :: \text{Object} \) holds. If \( t' \) is not \( \text{Object} \), then \( t :: t' \) holds if and only if \( t :: t' \) or \( t :: t' \) holds. In [I-InvSig] and [I-InvS2T], the information on args is also exploited, where args is an array of type \( \text{Object}[] \), only when it can be analyzed exactly element-wise by an intra-procedural analysis. In this case, suppose that \( \text{args} \) is an array of \( n \) elements. Let \( A_i \) be the set of types of the objects pointed to by its \( i \)-th element, \( \text{args}[i] \). Let
where c1 with the given signature s is ignored if defined if s
M(s.t, s.n, s.p)

Let us modify the reflective allocation site in line 3 (Figure 17) to c1.newInstance()

Example 5.1. Let us modify the reflective allocation site in line 3 (Figure 17) to c1.newInstance(). By applying [L-KwTp] (introduced

P_i = \{t' \mid t \in A_i, t <: t'\}. Then Ptp(args) = P_0 \times \cdots \times P_{n-1}. Otherwise, Ptp(args) = \emptyset, implying that args is ignored as it cannot be exploited effectively during inference.

To maintain precision in [I-InvS2T], [I-GetS2T] and [I-SetS2T], we use a method (field) signature to infer its classes when both its name and descriptor are known. In [I-InvS2T], the function M(s.t, s.n, s.p) returns the set of classes where the method with the specified signature s is defined if s.n != u and s.p != u, and \emptyset otherwise. The return type of the matching method is ignored if s.t = u. In [I-GetS2T] and [I-SetS2T], \mathcal{F}(s.n.f, s.t.f) returns the set of classes where the field with the given signature s is defined if s.n.f != u and s.t.f != u, and \emptyset otherwise.

Let us illustrate our rules by considering two examples in Figures 17 and 23.

Fig. 22. Rules for Collective Inference.
later in Figure 25) to the modified allocation site, SOLAR will create a new object $o^A_t$, which will flow to line 10, so that $o^A_t \in pt(v)$. Suppose both cName2 and mName2 point to some unknown strings. When [P-GetMtd] is applied to c.getMethod(mName, ...) in line 7, a Method object, say, $m_0^A$ is created and eventually assigned to m in line 14. By applying [I-InvTp] to m.invoke(v, args) in line 15, where $o_0^A \in pt(v)$, SOLAR deduces that the target method is a member of class A. Thus, a new object $m_0^A$ is created and assigned to pt(m). Given args = new Object[]{b, c}, $Ptp(args)$ is constructed as described earlier. By applying [I-InvSig] to this invoke() call, SOLAR will add all new Method objects $m_0^A$ to pt(m) such that $s.p \in Ptp(args)$, which represent the potential target methods called reflectively at this site.

Example 5.2. In Figure 23, hd is statically unknown but the string argument of getMethod() is "handle", a string constant. By applying [P-ForName], [P-GetMtd] and [L-UkwTp] (Figure 25) to the forName(), getMethod() and newInstance() calls, respectively, we obtain $c^u \in pt(c)$, $m^u \in pt(m)$ and $o^u \in pt(h)$, where s indicates a signature with a known method name (i.e., "handle"). Since the second argument of the invoke() call can also be exactly analyzed, SOLAR will be able to infer the classes t where method "handle" is defined by applying [I-InvS2T]. Finally, SOLAR will add all inferred Method objects $m_0^A$ to pt(m) at the invoke() call site. Since neither the superscript nor the subscript of $m_0^A$ is u, the inference is finished and the inferred $m_0^A$ will be used to find out the reflective targets (represented by it) in Target Search (Section 5.4.4).

5.4.4 Target Search. For a Method object $m_0^A$ in a known class t (with s being possibly u), we define $MTD : M \otimes T \rightarrow P(T)$ to find all the methods matched:

$$MTD(m_0^A) = \bigcup_{t} mtdLookUp(t', s.t_r, s.n_m, s.p)$$

where mtdLookUp is the standard lookup function for finding the methods according to a declaring class t' and a signature s except that (1) the return type s.t_r is also considered in the search (for better precision) and (2) any u that appears in s is treated as a wild card during the search.

Similarly, we define $FLD : F \otimes T \rightarrow P(T)$ for a Field object $f_0^A$:

$$FLD(f_0^A) = \bigcup_{t} fldLookUp(t', s.t_f, s.n_f)$$

to find all the fields matched, where fldLookUp plays a similar role as mtdLookUp. Note that both $MTD(m_0^A)$ and $FLD(f_0^A)$ also need to consider the super types of t (i.e., the union of the results for all t' where t < t', as shown in the functions) to be conservative due to the existence of member inheritance in Java.

Fig. 23. A simplified real code of Example 5.2 for illustrating inference rule [I-InvS2T]
5.4.5 Transformation. Figure 24 gives the rules used for transforming a reflective call into a regular statement, which will be analyzed by the pointer analysis.

\[
\begin{align*}
\frac{x = m.invoke(y, args) \quad m^f \in pt(m) \quad m' \in MTD(m^f) \quad o^- \in pt(args)}{\forall 1 \leq k \leq n : \{o_{j'}^k\} \subseteq pt(arg_k)} & \quad t'' \text{ is declaring type of } m'_{pk} \quad t' \ll t''} & & [T-Inv] \\
\frac{x = f.get(y) \quad f^f \in pt(f) \quad f \in FLD(f^f)}{x = y.f} & & [T-Get] \\
\frac{f.set(y, x) \quad f^f \in pt(f) \quad f \in FLD(f^f)}{y.f = x} & & [T-Set]
\end{align*}
\]

Fig. 24. Rules for Transformation.

Let us examine [T-Inv] in more detail. The second argument args points to a one-dimensional array of type Object[], with its elements collapsed to a single field arr during the pointer analysis, unless args can be analyzed exactly intra-procedurally in our current implementation. Let arg1, ..., argn be the n freshly created arguments to be passed to each potential target method m' found by Target Search. Let m'_{p1}, ..., m'_{pn} be the n parameters (excluding this) of m', such that the declaring type of m'_{pk} is t''. We include argj to pt(argk) only when t' \ll t'' holds in order to filter out the objects that cannot be assigned to m'_{pk}. Finally, the reflective target method found can be analyzed by [A-Call] in Figure 20.

5.4.6 Lazy Heap Modeling. In Figure 25, we give the rules for lazily resolving a newInstance() call, as explained in Section 4.3.

\[
\begin{align*}
i : o = c'.newInstance() \quad c^f \in pt(c') \quad \{o_i^f\} \subseteq pt(o) & & [L-KwTp] \\
i : o = c'.newInstance() \quad c^f \in pt(c') \quad \{o_i^f\} \subseteq pt(o) & & [L-UkwTp] \\
A \ a = (A) \times \ \{o_i^f\} \subseteq pt(a) \quad o_i^f \in pt(x) \quad t \ll A & & [L-Cast] \\
x = m.invoke(y, \ldots) \quad o_i^f \in pt(y) \quad m^f \in pt(m) \quad t' \ll t & & [L-Inv] \\
x = f.get(y) \quad \{o_i^f\} \subseteq pt(y) \quad o_i^f \in pt(y) \quad \{o_i^f\} \subseteq pt(y) & & [L-GSet]
\end{align*}
\]

Fig. 25. Rules for Lazy Heap Modeling.

In [L-KwTp], for each Class object c' pointed to by c', an object, o_i^f, is created as an instance of this known type at allocation site i straightaway. In [L-UkwTp], as illustrated in Figure 16, o_i^f is
created to enable LHM if \( c' \) points to a \( c'' \) instead. Then its lazy object creation happens at its Case (II) by applying [L-CAST] (with \( o'^{\prime} \) blocked from flowing from \( x \) to \( a \)) and its Case (III) by applying [L-INV] and [L-GSET]. Note that in [L-CAST], \( A \) is assumed not to be \( \text{Object} \).

5.5 Soundness Criteria

RefJava includes four reflective-action methods as described in Section 5.1. Solar is sound if their calls are resolved soundly under Assumptions 1 – 4. Due to Assumption 4 and the illustration in Figure 16, there is no need to consider \( \text{newInstance()} \) since it will be soundly resolved if \( \text{invoke()} \), \( \text{get()} \) and \( \text{set()} \) are soundly resolved. For convenience, we define:

\[
\text{AllKwn}(v) = \exists \exists a_i'' \in \text{pt}(v)
\]

which means that the dynamic type of every object pointed to by \( v \) is known.

Recall the nine rules given for resolving (A) \( m.\text{invoke}(y, \text{args}) \), (A) \( f.\text{get}(y) \) and \( f.\text{set}(y, x) \) in Figure 22. For the Method (Field) objects \( m'_o^{\prime} (f'_s) \) with known classes \( t \), these targets can be soundly resolved by Target Search, except that the signatures \( s \) can be further refined by applying [I-InvSig], [I-GetSig] and [I-SetSed].

For the Method (Field) objects \( m''_o (f'_s) \) with unknown class types \( u \), the targets accessed are inferred by applying the remaining six rules in Figure 22. Let us consider a call to (A) \( m.\text{invoke}(y, \text{args}) \). Solar attempts to infer the missing classes of its Method objects in two ways, by applying [I-InvTr] and [I-InvS2T]. Such a call is soundly resolved if the following condition holds:

\[
\text{SC}(m.\text{invoke}(y, \text{args})) = \text{AllKwn}(y) \lor \forall m''_o \in \text{pt}(m) : s.n_m \neq u \land PTp(args) \neq \emptyset
\]

If the first disjunct holds, applying [I-InvTr] to \( \text{invoke()} \) can over-approximate its target methods from the types of all objects pointed to by \( y \). Thus, every Method object \( m''_o \in \text{pt}(m) \) is refined into a new one \( m'_o \) for every \( o'_i \in \text{pt}(y) \).

If the second disjunct holds, then [I-InvS2T] comes into play. Its targets are over-approximated based on the known method names \( s.n_m \) and the types of the objects pointed to by \( \text{args} \). Thus, every Method object \( m''_o \in \text{pt}(m) \) is refined into a new one \( m'_o \), where \( s.t_r \leftarrow A \) and \( s.p \in PTp(args) \neq \emptyset \). Note that \( s.t_r \) is leveraged only when it is not \( u \). The post-dominating cast (A) is considered not to exist if \( A = \text{Object} \). In this case, \( u \leftarrow \text{Object} \) holds (only for \( u \)).

Finally, the soundness criteria for \( \text{get()} \) and \( \text{set()} \) are derived similarly:

\[
\text{SC}((A) f.\text{get}(y)) = \text{AllKwn}(y) \lor \forall f''_s \in \text{pt}(f) : s.n_f \neq u \land A \neq \text{Object}
\]

\[
\text{SC}((A) f.\text{set}(y, x)) = \text{AllKwn}(y) \lor \forall f''_s \in \text{pt}(f) : s.n_f \neq u \land \text{AllKwn}(x)
\]

In (5), applying [I-GetTr] ([I-GetS2T]) resolves a \( \text{get()} \) call soundly if its first (second) disjunct holds. In (6), applying [I-SetTr] ([I-SetS2T]) resolves a \( \text{set()} \) call soundly if its first (second) disjunct holds. By observing [T-Set], we see why \( \text{AllKwn}(x) \) is needed to reason about the soundness of [I-SetS2T].

5.6 Soundness Proof

We prove the soundness of Solar for RefJava subject to our soundness criteria (4) – (6) under Assumptions 1 – 4. We do so by taking advantage of the well-established soundness of Andersen’s pointer analysis (Figure 20) stated below.

**Lemma 5.3.** Solar is sound for RefJava with its reflection API ignored.

If we know the class types of all targets accessed at a reflective call but possibly nothing about their signatures, Solar can over-approximate its target method/fields in Target Search. Hence, the following lemma holds.
Lemma 5.4. Solar is sound for RefJava_c, the set of all RefJava programs in which cName is a string constant in every Class.forName(cName) call.

Proof Sketch. By [P-ForName], the Class objects at all Class.forName(cName) calls are created from known class types. By Lemma 5.3, this has four implications. (1) LHM is not needed. For the rules in Figure 25, only [L-KwTp] is relevant, enabling every c'.newInstance() call to be resolved soundly as a set of regular new t() calls, for all Class objects c' ∈ pt(c'). (2) In [P-GetMtd], the Method objects m' of all class types t accessed at a getMethod() call are created, where m' symbolizes over-approximately all target methods in MTD(m'). The same sound approximation for getField() is made by [P-GetField]. (3) For the nine rules given in Figure 22, only [I-InvSig], [I-GetSig] and [I-SetSig] are applicable, since ∄m_u ∈ m, ∄f_u ∈ f and ∄o_u ∈ y, in order to further refine their underlying method or field signatures. (4) In Figure 24, the set of reflective targets at each call site is over-approximated. By applying Lemma 5.3 again and noting Assumptions 1 – 3, Solar is sound for RefJava_c.

Solar is sound subject to (4) – (6) under Assumptions 1 – 4.

Theorem 5.5. Solar is sound for RefJava if SC(c) holds at every reflective call c of the form (A) m.invoke(y, args), (A) f.get(y) or f.set(y, x).

Proof Sketch. For reasons of symmetry, we focus only on a call, c, to (A) m.invoke(y, args). SC(c) is given in (4). If its first disjunct is true, then all abstract objects flowing into y are created either from known classes soundly by [L-KwTp] or lazily from unknown classes as illustrated in Figure 16 by applying initially [L-UkwTp] and later [L-Cast] (for Case (II)) and [L-Inv] (for Case (III)), but soundly under Assumption 4. If its second disjunct is true, then Solar can always infer the missing class types t in a Method object m'_t pointed to by pt(m) to over-approximate the set of its target methods as MTD(m'_t). This takes us back to a situation equivalent to the one established in the proof of Lemma 5.4. Thus, Solar is sound for RefJava if SC(c) holds at every reflective call c of the form (A) m.invoke(y, args).

5.7 Probe

We instantiate Probe, as shown in Figure 13, from Solar as follows. To make Probe resolve reflection precisely, we refrain from performing Solar’s LHM by retaining [L-UkwTp] but ignoring [L-Cast], [L-GetTp] and [L-Inv] and abandon some of Solar’s sophisticated inference rules by disabling [I-InvS2T], [I-GetS2T] and [I-SetS2T].

In Target Search, Probe will restrict itself to only Method objects m'_t and Field objects f'_s, where the signature s is at least partially known.

5.8 Static Class Members

To handle static class members, our rules can be simply modified. In Figure 22, y = null. [I-InvTp], [I-GetTp] and [I-SetTp] are not needed (by assuming pt(null) = ∅). In the soundness criteria stated in (4) – (6), the first disjunct is removed in each case. [I-InvS2T], [I-GetS2T] and [I-SetS2T] are modified with o'_t ∈ pt(y) being replaced by y = null. The rules in Figure 24 are modified to deal with static members. In Figure 25, [L-GetS] and [L-Inv] are no longer relevant. The other rules remain applicable. The static initializers for the classes in the closed world are analyzed. This can happen at, say loads/stores for static fields as is the standard but also when some classes are discovered in [P-ForName], [L-Cast], [L-GSet] and [L-Inv].
6 IMPLEMENTATION

Solar, as shown in Figure 18, works together with a pointer analysis. We have implemented Solar on top of Doop [8], a state-of-the-art pointer analysis framework for Java. Solar is implemented in the Datalog language. Presently, Solar consists of 303 Datalog rules written in about 2800 lines of code.

We release Solar as an open-source tool at http://www.cse.unsw.edu.au/~corg/solar. Now Solar has been augmented to output its reflection analysis results with the format that is supported by Soot [63] (a popular framework for analyzing Java and Android applications), which enables Soot’s clients to use Solar’s results directly.

Below we extend our formalism for RefJava to handle the other methods in the Java reflection API, divided into class-retrieving, member-retrieving and reflective-action methods. For details, we refer to the open-source implementation of Solar.

Class-Retrieving Methods. We divide the class-retrieving methods into six groups, depending how a Class object is obtained, by (1) using a special syntax (A.class), (2) calling Object.getClass(), (3) using a full-qualified string name (in Class.forName() and ClassLoader.loadClass()), (4) calling proxy API Proxy.getProxyClass(), (5) calling, e.g., Class.getComponentType(), on a metaobject, and (6) calling, e.g., sun.reflect.Reflection.getCallerClass() to introspect an execution context. According to Section 2.3.3, the last three are infrequently used. Our current implementation handles the class-retrieving methods in (1) – (4) plus Class.getComponentType() in (5) since the latter is used in array-related reflective-action methods.

Member-Retrieving Methods. In addition to the two included in RefJava, there are ten more member-retrieving methods as given in Figure 4. To show how to handle the other ten methods based on our existing formalism (Section 5), we take getDeclaredMethod() and getMethods() as examples as the others can be dealt with similarly. For a Method object resolved to be \( m' \) at a getDeclaredMethod() call by Propagation, the search for building \( MTD(m'_t) \) is confined to class \( t \) (the only change in the rules). Otherwise, the search is done as described in Section 5.4.4.

For \( ms = c'.getMethods() \), which returns an array of Method objects, its analysis is similarly done as \( m = c'.getMethod(mName) \) in Figure 21. We first create a placeholder array object \( ms_{ph} \) so that \( ms_{ph} \in pt(ms) \).

As getMethods() is parameterless, we then insert a new Method object \( m'_u \) (\( m'_u \in pt(ms_{ph,arr}) \)) for every \( c' (c'' \in pt(c') \). The underlying pointer analysis will take care of how eventually an element of such an array flows into, say \( m \), in \( m.invoke() \). Then the rules in Figure 22 are applicable.

Reflective-Action Methods. In addition to the four in RefJava, the following four reflective-action methods are also analyzed. Constructor.newInstance() is handled as Class.newInstance() except that its array argument is handled exactly as in invoke(). Its call sites are also modeled lazily. Array.newInstance(), Array.get() and Array.set() are handled as in Table 1. Presently, Solar does not handle Proxy.newProxyInstance() in its implementation. However, it can be analyzed according to its semantics as described in Section 2.2.2. Recently, Fourtounis et al. [20] introduce a static analysis approach to effectively handle more complex Java proxy cases.

7 EVALUATION

We evaluate Solar by comparing it (primarily) with two state-of-the-art reflection analyses for Java, DoopR [51] and Elf [32]. In particular, our evaluation addresses the following research questions (RQs):

- RQ1. How well does Solar achieve full automation without any annotations?
• RQ2. How well does Solar reason about the soundness and identify the unsoundly or imprecisely resolved (or “problematic”) reflective calls?
• RQ3. How well does Solar make the trade-offs among soundness, precision and scalability in terms of reflection analysis? In other words, can Solar resolve reflection more soundly than Doop and Elf with acceptable precision and efficiency?

Below we describe our experimental setup, revisit our assumptions, and answer these RQs in the given order.

7.1 Experimental Setup
As all the three reflection analyses, Doop, Elf and Solar, are implemented in Doop [8] (a state-of-the-art pointer analysis framework for Java) and run with a pointer analysis, we compare them by running each together with the same context-sensitive pointer analysis option provided by Doop: selective-2-type-sensitive+heap [25].

Doop denotes the reflection analysis proposed in [51] and all its provided reflection resolution options are turned on including -enable-reflection, -enable-reflection-substring-analysis, -enable-reflection-use-based-analysis and -enable-reflection-invent-unknown-objects, in the experiment. Elf (version 0.3) is the reflection analysis proposed in [32] with its default setting.

We use the LogicBlox Datalog engine (v3.9.0) on a Xeon E5-2650 2GHz machine with 64GB of RAM. We consider 7 large DaCapo benchmarks (2006-10-MR2) and 4 real-world applications, avrora-1.7.115 (a simulator), checkstyle-4.4 (a checker), freecs-1.3.20111225 (a server) and findbugs-1.2.1 (a bug detector), under a large reflection-rich Java library, JDK 1.6.0_45. The other 4 small DaCapo benchmarks are excluded since reflection is much less used.

7.2 Assumptions
When analyzing the real code in the experiment, we accommodate Assumptions 1 – 4 as follows. For Assumption 1, we rely on Doop’s pointer analysis to simulate the behaviors of Java native methods. Dynamic class loading is assumed to be resolved separately [49]. To simulate its effect, we create a closed world for a program, by locating the classes referenced with Doop’s fact generator and adding additional ones found through program runs under TamiFlex [5]. For the DaCapo benchmarks, their three standard inputs are used. For avrora and checkstyle, their default test cases are used. For findbugs, one Java program is developed as its input since no default ones are available. For freecs, a server requiring user interactions, we only initialize it as the input in order to ensure repeatability. For a class in the closed-world of a program, Solar analyzes its static initializer at the outset if it is dynamically loaded and proceeds as discussed in Section 5.8 otherwise.

Assumptions 2 and 3 are taken for granted.

As for Assumption 4, we validate it for all reflective allocation sites where oᵢᵤ is created in the application code of the 10 programs that can be analyzed scalably. This assumption is found to hold at 75% of these sites automatically by performing a simple intra-procedural analysis. We have inspected the remaining 25% interprocedurally and found only two violating sites (in eclipse and checkstyle), where oᵢᵤ is never used. In the other sites inspected, oᵢᵤ flows through only local variables with all the call-chain lengths being at most 2. This shows that Assumption 4 generally holds in practice.
7.3 RQ1: Automation and Annotations

Figure 26 compares SOLAR and other reflection analyses [2, 28, 32, 38, 51, 68] denoted by “Others” by the degree of automation achieved. For an analysis, this is measured by the number of annotations required in order to achieve the soundness of the reflective calls identified to be potentially unsoundly resolved.

![Number of annotations required for achieving the soundness of unsoundly resolved reflective calls.](image)

Fig. 26. Number of annotations required for achieving the soundness of unsoundly resolved reflective calls.

SOLAR analyzes 7 out of the 11 programs scalably with full automation. For hsqldb, xalan and checkstyle, SOLAR is unscalable (under 3 hours). With PROBE, a total of 14 reflective calls are flagged as being potentially unsoundly/imprecisely resolved in these three programs. After 7 annotations, 2 in hsqldb, 2 in xalan, and 3 in checkstyle, SOLAR is scalable, as discussed in Section 7.4. However, SOLAR is unscalable (under 3 hours) for jython, an interpreter (from DaCapo) for Python in which the Java libraries and application code are invoked reflectively from the Python code. Neither are Doop and Elf.

“Others” cannot identify which reflective calls may be unsoundly resolved; however, they can improve their soundness by adopting an annotation approach suggested in [38]. Generally, this approach requires users to annotate the string arguments of calls to class-retrieving and member-retrieving methods if they are not string constant. To reduce the annotation effort, e.g., for a `clz = Class.forName(cName)` call with cName being an input string, this approach leverages the intra-procedural post-dominating cast on the result of a `clz.newInstance()` call to infer the types of clz.

To the best of our knowledge, the above approach [38] represents currently the best practice with the least annotation burden to achieve a sound reflection analysis. Therefore, “Others” in Figure 26 represents the number of annotation needed (for soundness) by the existing reflection analyses that employs this approach.

As shown in Figure 26, “Others” will require a total of 338 annotations initially and possibly more in the subsequent iterations (when more code is discovered). As discussed in Section 3.2.1, SOLAR’s annotation approach is also iterative. However, for these programs, SOLAR requires a total of 7 annotations in only one iteration.
Solar outperforms “Others” due to its powerful inference engine for performing reflection resolution and its effective mechanism in accurately identifying unsoundly resolved reflective calls as explained in Section 3.1.

7.4 RQ2: Automatically Identifying “Problematic” Reflective Calls

If Solar is scalable in analyzing a program with no unsoundly resolved calls reported, then Solar is sound for this program under Assumptions 1 – 4 (Theorem 5.5). Thus, as discussed in Section 7.3, Solar is sound (in this sense) for the seven Java programs scalably analyzed (under 3 hours) with full automation in our experiment.

Solar is unscalable for hsqldb, xalan and checkstyle (under 3 hours). Probe is then run to identify their “problematic” reflective calls, reporting 13 potentially unsoundly resolved calls: 1 in hsqldb, 12 in xalan and 0 in checkstyle. By code inspection, we found that they are all indeed unsound, demonstrating the effectiveness of Solar in accurately pinpointing a small number of right parts of the program to improve unsoundness.

In addition, we currently adopt a simple approach to alerting users for potentially imprecisely resolved reflective calls. Probe sorts all the newInstance() call sites according to the number of objects lazily created at the cast operations operating on the result of a newInstance() call (by $[L$-Cast$]$) in non-increasing order. In addition, Probe ranks the remaining reflective call sites (of other reflective-action methods) according to the number of reflective targets resolved, also in non-increasing order.

By focusing on unsoundly and imprecisely resolved reflective calls as opposed to the unknown input strings (see Section 7.3), only lightweight annotations are needed as shown in Figure 26, with 2 hsqldb, 2 xalan and 3 in checkstyle as explained below.

Probing hsqldb. Earlier, Figure 14 shows the output automatically generated for hsqldb by Probe (highlighting which reflective calls are resolved unsoundly or imprecisely), together with the suggested annotation sites (as introduced in Section 4.5). In Figure 14, all the call sites to (invoke) the same method are numbered from 0. The unsound list contains one invoke(), with its relevant code appearing in class org.hsqldb.Function as shown. After Probe has finished, mtd in line 352 points to a Method object $m$ that is initially created in line 179 and later flows into line 182, indicating that the class type of $m$ is unknown since $cn$ in line 169 is unknown. By inspecting the code, we find that $cn$ can only be java.lang.Math or org.hsqldb.Library, read from some hash maps or obtained by string manipulations and is annotated as:

\[
\text{org.hsqldb.Function.<init>/java.lang.Class.forName/0 java.lang.Math} \\
\text{org.hsqldb.Function.<init>/java.lang.Class.forName/0 org.hsqldb.Library}
\]

The annotation contains two parts: the reflective call site that needs to be annotated and the signatures of its reflective targets. As the reflective targets in the above example have two different class types, the signatures are just their class names, java.lang.Math and org.hsqldb.Library. The prefix before the two names serves to indicate that the reflective call site java.lang.Class.forName is the first forName call (indexed by number 0) in the constructor (denoted by <init>) of class org.hsqldb.Function.

In Figure 14, the imprecise list for hsqldb is divided into two sections. In “newInstance (Type Casting)”, there are 10 listed cast operations (7) reached by an $o$ object such that the number of types inferred from $T$ is larger than 10. The top cast java.io.Serializable has 1391 subtypes and is marked to be reached by a newInstance() call site in java.io.ObjectStreamClass. However, this is a false positive for the harness used as the newInstance() call site is unreachable from the
harness during runtime, but it is reachable by the pointer analysis due to its imprecision. Thus, we have annotated its corresponding `Class.forName()` call site in method `resolveClass` of class `java.io.ObjectInputStream` (which is also unreachable from the harness used) to return nothing. With the two annotations, SOLAR terminates in 45 minutes with its unsound list being empty.

Probing xalan. PROBE reports 12 unsoundly resolved `invoke()` call sites. All `Method` objects flowing into these call sites are from two `getMethods()` call sites in class `extensions.MethodResolver`. By inspecting the code, we find that the string arguments for the two `getMethods()` calls and their corresponding class-retrieving methods are all read from a file with its name hard-wired as `xmlspec.xsl`. For this only input file provided by DaCapo, these two calls are never executed and thus annotated to be disregarded. With these two annotations, SOLAR terminates in 28 minutes with its unsound list being empty.

Probing checkstyle. PROBE reports no unsoundly resolved call. To see why SOLAR is unscalable, we examine one `invoke()` call in line 1773 of Figure 27 found automatically by PROBE that stands out as being imprecisely resolved.

![Fig. 27. Probing checkstyle.](image)

There are 962 target methods inferred at this call site. PROBE highlights its corresponding member-retrieving method `clz.getMethods()` (in line 1294) and its class-retrieving methods (with one of these being shown in line 926). Based on this, we find easily by code inspection that the target methods called reflectively at the `invoke()` call are the setters whose names share the prefix “set”. As a result, we annotate the `clz.getMethods()` call to return 158 “setX” methods in all the subclasses of `AutomaticBean`.

In addition, the `Method` objects created at one `getMethods()` call and one `getDeclaredMethods()` call in class `*.beanutils.MappedPropertyDescriptor$1` flow into the `invoke()` call in line 1773 as false positives due to imprecision in the pointer analysis. These `Method` objects have been annotated away.

After the three annotations, SOLAR is scalable, terminating in 38 minutes.
7.5 RQ3: Soundness, Precision and Efficiency

In this section, we first compare the soundness of Doop', Elf and Solar and then compare their analysis precision and efficiency to see if Solar’s precision and efficiency are still acceptable while being significantly more sound than the prior work.

7.5.1 Soundness. To compare the soundness of Doop', Elf and Solar, it is the most relevant to compare their recall, measured by the number of true reflective targets discovered at reflective calls to reflective-action methods that are dynamically executed under certain inputs. Unlike Doop' and Elf, Solar can automatically identify “problematic” reflective calls for lightweight annotations. To ensure a fair comparison, the three annotated programs shown in Figure 26 are used by all the three reflection analyses.

Recall. Table 2 compares the recall of Doop', Elf and Solar. Due to the lack of the ground truth about the reflective targets in the benchmarks and applications considered, we use TamiFlex [5], a practical dynamic reflection analysis tool to find the targets accessed at reflective calls in our programs under the inputs described in Section 7.2. We then compare these three analyses in terms of their recall rates with respect to these true reflective targets. Of the three analyses compared, Solar is the only one to achieve total recall, for all reflective targets, including both methods and fields, accessed.

For each DaCapo benchmark, its main code body is run under a reflection-based harness. As a result, static analysis tools, including Doop', Elf and Solar, use its xdeps version driven by a reflection-free harness. However, TamiFlex needs to run each DaCapo benchmark with the reflection-based harness dynamically. For each DaCapo benchmark, two harnesses lead to different versions used for a few classes (e.g., org.apache.xerces.parsers.SAXParser) in eclipse, fop, pmd and xalan. In Table 2, we thus ignore the totally 21 extra reflective targets accessed by TamiFlex.

Although Solar achieves total recall, it does not mean that Solar can replace dynamic reflection analyses like TamiFlex [5]. To simulate the closed-world, as described in Section 7.2, in order to verify the soundness of Solar under the assumptions made in Section 4.1, we ran TamiFlex to find some additional dynamically loaded classes (which are missed by Doop’s fact generator) in the closed-world. In other words, how to construct a sound closed-world is still a big challenge for static analysis, and the current practice is to leverage dynamic analysis to help build it more soundly. However, as shown in Table 2, once a closed-world is given, Solar can achieve total recall (and better soundness than other static reflection analyses). So we have experimentally verified the soundness of Solar (that is theoretically proven in Section 5.6) in terms of recall.

In practice, a reflection analysis must handle newInstance() and invoke() well in order to build the call graph for a program. Let us see how resolving more (true) reflective targets in a program by Solar can affect the call graph of the program.

Call Graph. Figure 28 compares Doop', Elf and Solar in terms of true caller-callee relations discovered. These numbers are statically calculated and obtained by an instrumental tool written on top of Javassist [10]. According to Table 2, Solar recalls a total of 191% (148%) more targets than Doop' (Elf) at the calls to newInstance() and invoke(), translating into 44787 (40583) more true caller-callee relations found for the 10 programs under the inputs described in Section 7.2. These numbers are expected to improve when more inputs are used.

Note that all method targets recalled by Doop' and Elf are recalled by Solar so that we can use the “subtraction” (i.e., Solar − Doop' and Solar − Elf) in Figure 28. More true caller-callee relations identified implies more soundness achieved. As shown in Figure 28, Solar is more sound than the other two analyses in all programs and is significantly more sound than them in eclipse, findbugs and checkstyle.
Table 2. Recall comparison. For each program, TamiFlex denotes the number of targets found by TamiFlex and Recall denotes the number of such (true) targets also discovered by each reflection analysis. There are two types of newInstance(): c.newInstance (in Class) and ct.newInstance (in Constructor).

<table>
<thead>
<tr>
<th></th>
<th>c.newInstance</th>
<th>ct.newInstance</th>
<th>invoke</th>
<th>get/set</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Doop' Elf Solar</td>
<td>Doop' Elf Solar</td>
<td>Doop' Elf Solar</td>
<td>Doop' Elf Solar</td>
</tr>
<tr>
<td>chart</td>
<td>Recall</td>
<td>13 21 22</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>22 22 22</td>
<td>2 2 2 2</td>
<td>2 2 2 2</td>
</tr>
<tr>
<td>eclipse</td>
<td>Recall</td>
<td>9 24 37</td>
<td>1 3 3 3</td>
<td>2 7 7 7</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>37 37 37</td>
<td>3 3 3 3</td>
<td>7 7 7 7</td>
</tr>
<tr>
<td>fop</td>
<td>Recall</td>
<td>9 12 13</td>
<td>0 0 0 0</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>13 13 13</td>
<td>0 0 0 0</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>hsqldb</td>
<td>Recall</td>
<td>5 9 10</td>
<td>1 1 1 1</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>10 10 10</td>
<td>1 1 1 1</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>pmd</td>
<td>Recall</td>
<td>4 8 13</td>
<td>0 0 0 0</td>
<td>0 0 0 7</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>13 13 13</td>
<td>0 0 0 0</td>
<td>7 7 7 7</td>
</tr>
<tr>
<td>xalan</td>
<td>Recall</td>
<td>36 42 43</td>
<td>0 0 0 0</td>
<td>32 5 32</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>43 43 43</td>
<td>0 0 0 0</td>
<td>32 32 32</td>
</tr>
<tr>
<td>avrora</td>
<td>Recall</td>
<td>50 46 53</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>53 53 53</td>
<td>0 0 0 0</td>
<td>0 0 0 0</td>
</tr>
<tr>
<td>checkstyle</td>
<td>Recall</td>
<td>7 9 72</td>
<td>1 1 1 24</td>
<td>1 5 28</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>72 72 72</td>
<td>24 24 24</td>
<td>24 24 24</td>
</tr>
<tr>
<td>findbugs</td>
<td>Recall</td>
<td>6 10 15</td>
<td>8 8 115</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>15 15 15</td>
<td>115 115 115</td>
<td>1 1 1 1</td>
</tr>
<tr>
<td>frees</td>
<td>Recall</td>
<td>6 10 12</td>
<td>2 2 2 2</td>
<td>1 1 55</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>12 12 12</td>
<td>2 2 2 2</td>
<td>1 1 55</td>
</tr>
<tr>
<td>Total</td>
<td>Recall</td>
<td>141 191 290</td>
<td>15 17 147</td>
<td>40 22 133</td>
</tr>
<tr>
<td></td>
<td>TamiFlex</td>
<td>290 290 290</td>
<td>147 147 147</td>
<td>133 133 133</td>
</tr>
</tbody>
</table>

As examples, let us examine eclipse and checkstyle. We start with eclipse. According to Table 2, Solar finds 35 more target methods than Doop’, causing 25177 more true caller-callee relations to be discovered. Due to LHM, Solar finds 13 more target methods at newInstance() calls than Elf, resulting in 20400 additional true caller-callee relations to be introduced.

Let us consider now checkstyle. Due to collective inference, Solar has resolved 23 more target methods at invoke() calls than Elf, resulting in 2437 more true caller-callee relations (i.e., over a third of the total number of such relations, 6706) to be discovered.

7.5.2 Precision. Table 3 compares the analysis precision of Doop’, Elf and Solar with an important client, devirtualization, which is only applicable to virtual calls with one single target each. This client is critical in call graph construction and often used for measuring the precision of a pointer analysis. The higher the percentage of devirtualization is, the more precise its corresponding analysis is. Note that this statement is valid only if all the static analyses have the same degree of soundness. In our case, however, Solar is more sound than the other two analyses (as demonstrated
in Section 7.5.1). Instead of comparing their exact precision, Table 3 serves to show that despite achieving better recall (Table 2), which results in more true caller-callee relations to be discovered (Figure 28), Solar still maintains a relatively good precision (similar precision as Doop$^r$ and Elf for this client).

In Solar, its lazy heap modeling (LHM) relies on cast types (and their subtypes) to infer reflective targets. Will this inference strategy introduce too much imprecision into the analysis when a cast type has many subtypes? To address this concern, we conduct an experiment about the percentage distribution for the number of types inferred at cast-related LHM points. As shown in Figure 29, the number of inferred types in a program is 1 ($\leq 10$) at 30.8% (85.4%) of its cast-related LHM points. Some types (e.g., java.io.Serializable) are quite wide, giving rise to more than 50 inferred types each, but rare, appearing at an average of about 1.9% cast-related LHM points in a program.

7.5.3 Efficiency. Table 4 compares the analysis times of Doop$^r$, Elf and Solar. Despite producing significantly better soundness than Doop$^r$ and Elf, Solar is only several-fold slower (with the average calculated as a geometric mean). When analyzing hsqldb, xalan and checkstyle, Solar requires some lightweight annotations (Section 7.3). Their analysis times are the ones consumed by Solar on analyzing the annotated programs. Note that these annotated programs are also used by Doop and Elf (to ensure a fair comparison).
Fig. 29. Percentage distribution for the number of types inferred at cast-related LHM points. For each program, the total number of the cast-related LHM points is shown in the middle of its own bar.

Table 4. Comparing Doopr, Elf, Solar and Solar$^s$ in terms of analysis times (secs).

<table>
<thead>
<tr>
<th></th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
<th>pmd</th>
<th>xalan</th>
<th>avrora</th>
<th>checkstyle</th>
<th>findbugs</th>
<th>freecs</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doopr</td>
<td>1864</td>
<td>340</td>
<td>2631</td>
<td>1173</td>
<td>602</td>
<td>1092</td>
<td>2934</td>
<td>531</td>
<td>2753</td>
<td>371</td>
<td>1074</td>
</tr>
<tr>
<td>Elf</td>
<td>3434</td>
<td>5496</td>
<td>2821</td>
<td>1765</td>
<td>1363</td>
<td>1432</td>
<td>932</td>
<td>1463</td>
<td>2281</td>
<td>1259</td>
<td>1930</td>
</tr>
<tr>
<td>Solar</td>
<td>4322</td>
<td>9710</td>
<td>4089</td>
<td>2471</td>
<td>2084</td>
<td>1634</td>
<td>3371</td>
<td>2160</td>
<td>7350</td>
<td>2589</td>
<td>3390</td>
</tr>
<tr>
<td>Solar$^s$</td>
<td>2005</td>
<td>542</td>
<td>1603</td>
<td>1863</td>
<td>1238</td>
<td>1117</td>
<td>790</td>
<td>1307</td>
<td>1868</td>
<td>909</td>
<td>1229</td>
</tr>
</tbody>
</table>

Table 5. Comparing Doopr, Elf, Solar and Solar$^s$ in terms of the number of statically resolved call graph edges.

<table>
<thead>
<tr>
<th></th>
<th>chart</th>
<th>eclipse</th>
<th>fop</th>
<th>hsqldb</th>
<th>pmd</th>
<th>xalan</th>
<th>avrora</th>
<th>checkstyle</th>
<th>findbugs</th>
<th>freecs</th>
<th>Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>Doopr</td>
<td>109,020</td>
<td>51,161</td>
<td>107,805</td>
<td>68,849</td>
<td>63,479</td>
<td>56,275</td>
<td>106,741</td>
<td>66,872</td>
<td>74,151</td>
<td>51,322</td>
<td>72,509</td>
</tr>
<tr>
<td>Elf</td>
<td>126,791</td>
<td>97,443</td>
<td>75,389</td>
<td>72,855</td>
<td>76,589</td>
<td>82,747</td>
<td>74,469</td>
<td>90,888</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solar</td>
<td>127,912</td>
<td>187,343</td>
<td>102,497</td>
<td>72,789</td>
<td>118,573</td>
<td>91,194</td>
<td>122,820</td>
<td>92,377</td>
<td>104,654</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Solar$^s$</td>
<td>106,830</td>
<td>49,971</td>
<td>64,442</td>
<td>70,195</td>
<td>63,290</td>
<td>51,203</td>
<td>61,250</td>
<td>66,384</td>
<td>74,747</td>
<td>52,181</td>
<td>64,473</td>
</tr>
</tbody>
</table>

Solar spends more time than the other two reflection analyses. For a benchmark, the extra time is mainly spent on analyzing the extra code discovered by Solar’s more powerful inference capability. According to Table 5, Solar discovers significantly more call graph edges than Doopr and Elf, making it significantly more sound (as evaluated in Section 7.5.1).

To demonstrate that Solar has actually spent the extra time on analyzing the more code discovered, we introduce Solar$^s$, a configuration of Solar with its inference capability turned off, implying that Solar$^s$ resolves a reflective call only if its string arguments are statically known. The results for Solar$^s$ are included in Tables 4 and 5. Without taking advantage of Solar’s inference capability, Solar$^s$ is faster than Solar (Table 4), but at the expense of resolving much fewer call graph edges (Table 5).

In general, Doopr is more powerful than Solar$^s$ since Doopr resolves more call graph edges than Solar$^s$ in most of the benchmarks considered. However, for some benchmarks, such as chart
and xalan, where Doop$^r$ resolves more call graph edges, Doop$^r$ also turns out to be faster. This is because Solar$^r$ (just like Solar) is built on an older version of Doop than Doop$^r$. In Doop$^r$, the underlying pointer analysis is optimized for better performance, by e.g., merging some library objects of the same type per method.

In summary, the experimental results described in Section 7.5 demonstrate that Solar’s soundness-guided design is effective in balancing soundness, precision and scalability, in practice: Solar is able to achieve significantly better soundness than the state-of-the-art while being still reasonably precise and efficient.

8 RELATED WORK

8.1 Static Reflection Analysis

Livshits et al. [38] introduce the first static reflection analysis for Java. By interleaving with a pointer analysis, this reflection analysis discovers constant string values by performing regular string inference and infers the types of reflectively created objects at clz.newInstance() calls, where clz is unknown, by exploiting their intra-procedurally post-dominating cast operations, if any. Many modern pointer analysis frameworks such as Doop [8], Wala [64] and Chord [43], adopt a similar approach to analyze Java reflection, and many subsequent reflection analyses [32, 33, 51, 67, 68], including Solar, are also inspired by the same work.

Elf [32] represents the first reflection analysis that takes advantage of the self-inferencing property (Definition 2.4) to handle Java reflection. Elf can be considered as performing the collective inference in Solar’s inference engine except that Elf’s inference rules are more strict (for better analysis precision). In Elf, a reflective target will not be resolved unless both a red circle and a blue circle in Figure 15 (i.e., a class name and part of a member signature) are available. As a result, Elf is usually more precise and efficient, but (much) less sound than Solar.

In Doop$^r$ [51] (the one compared with Solar in Section 7), the authors propose to leverage partial string information to resolve reflective targets. As explained in Section 2.3.3, many string arguments are generated through complex string manipulations, causing their values to be hard to resolve statically. However, in some cases, its substring analysis makes it possible to leverage some partially available string values to help infer reflective targets. Combining this with some sophisticated string analyses [11] is expected to generate more effective results. Recently, Grech et al. [22] introduce an efficient reflection string analysis via graph coloring.

To improve soundness, Doop$^r$ attempts to infer the class types at Class.forName() (rather than newInstance()) call sites by leveraging both (1) the intra- and inter-procedural cast operations that are not necessarily post-dominating at their corresponding newInstance() call sites and (2) the member names at their corresponding member-retrieving call sites (i.e., the blue circles formed by Target Propagation in Figure 15). In some cases, these two strategies may make the analysis imprecise. For example, the second strategy may suffer from a precision loss when different unrelated classes contain identically-named members [32]. In addition, in both strategies (1) and (2), the class types (at a Class.forName() call site), which are back-propagated (inferred) from some member-retrieving call sites (cast sites), may further pollute the analysis precision at the other member-retrieving and reflective-action calls. To reduce such imprecision, an approach called inventing objects is proposed in Doop$^r$. It creates objects at the newInstance() call sites (according to the types at the related cast sites), rather than the corresponding Class.forName() call sites. This method is similar to Case (II) in Solar’s lazy heap modeling (Figure 16).

Barros et al. [2] analyze Java reflection and intents in Android apps and its soundness is from the perspective of a given client. Consider a client for detecting sensitive data leaks. Any invoke() call that cannot be resolved soundly is assumed to return conservatively some sensitive data.
While being sound for the client, the reflection analysis itself is still unsound. In terms of reflection resolution, Barros et al. [2] exploit only a subset of self-inferencing hints used in SOLAR. For instance, regarding how the arguments in an `invoke()` call are leveraged in reflection resolution, Barros et al. [2] use only their arity but ignore their types while SOLAR takes both into account. There are precision implications. Consider the code fragment for Eclipse in Figure 10. If the types of the elements of its one-dimensional array, parameters, are ignored, many spurious target methods in line 175 would be inferred, as single-argument method is very common.

Despite recent advances on reflection analysis [32, 33, 51], a sophisticated reflection analysis does not co-exist well with a sophisticated pointer analysis, since the latter is usually unscalable for large programs [32, 33, 38, 51]. If a scalable but imprecise pointer analysis is used instead, the reflection analysis may introduce many spurious call graph edges, making its underlying client applications to be too imprecise to be practically useful. This problem can be alleviated by using a recently proposed program slicing approach, called program tailoring [34].

Briefly, program tailoring accepts a sequential criterion (e.g., a sequence of reflective call sites: `forName()`→`getMethod()`→`invoke()`), and generates a soundly tailored program that contains the statements only relevant to the given sequential criterion. In other words, the tailored program comprises the statements in all possible execution paths passing through the sequence(s) in the given order. As a result, a more precise (but less scalable) pointer analysis may be scalable when the tailored program (with smaller size) is analyzed, resulting a more precise result resolved at the given reflective call site (e.g., the above `invoke()` call site). These reflective call sites can be the problematic ones generated by SOLAR as demonstrated in [34].

DROIDRA [28] analyzes reflection in Android apps by propagating string constants. This is similar to the target propagation in SOLAR’s collective inference except that DROIDRA uses the solver COAL [45] to resolve the string values in a context- and flow-sensitive manner. Currently, SOLAR’s target propagation is context-sensitive only, achieved by the pointer analysis used.

RIPPLE [67, 68] resolves reflection in Android apps by tackling their incomplete information environments (e.g., undetermined intents or unmodeled services). Technically, even if some data flows are missing (i.e., `null`), it is still able to resolve reflective targets by performing type inference, which is similar to SOLAR’s collective inference except that some sophisticated inference rules are not supported.

Unlike the prior work [2, 22, 28, 32, 35, 38, 51, 67], as highlighted in Figure 1, SOLAR is capable of reasoning about its soundness and accurately identifying its unsoundness.

8.2 Dynamic Reflection Analysis

Hirzel et al. [24] propose an online pointer analysis for handling some dynamic features in Java at run time. To tackle reflection, their analysis instruments a program so that constraints are generated dynamically when the injected code is triggered at run time. Thus, the points-to information is incrementally updated when new constraints are gradually introduced by reflection. This technique for reflection handling can be used in JIT optimizations but may not be suitable for whole-program static analysis.

Bodden et al. [5] leverage the runtime information gathered at reflective calls. Their tool, TAMI-FLEX, records usage information of reflective calls in the program at run-time, interprets the logging information, and finally, transforms these reflective calls into regular Java method calls to facilitate static analysis. In addition, TAMI-FLEX inserts runtime checks to warn the user in cases that the program encounters reflective calls that diverge from the recorded information of previous runs.

HARVESTER [47] is designed to automatically extract runtime values from Android applications. It uses a variation of traditional program slicing and dynamic execution to extract values from
obfuscated malware samples that obfuscate method calls using reflection or hide sensitive values in native code.

8.3 Others

Braux and Noyé [7] provide offline partial evaluation support for reflection in order to perform aggressive compiler optimizations for Java programs. It transforms a program by compiling away the reflection code into regular operations on objects according to their concrete types that are constrained manually. The inference engine of SOLAR can be viewed as a tool for inferring such constraints automatically.

To increase code coverage, some static analysis tools [8, 65] allow users to provide ad hoc manual specifications about reflection usage in a program. However, due to the diversity and complexity of applications, it is not yet clear how to do so in a systematic manner. For framework-based web applications, Sridharan et al. [56] introduce a framework that exploits domain knowledge to automatically generate a specification of framework-related behaviours (e.g., reflection usage) by processing both application code and configuration files. SOLAR may also utilize domain knowledge to analyze some configuration files, but only for those reflective call sites that cannot be resolved effectively.

Liu et al. [35] introduce a hybrid reflection analysis by combing static and dynamic analyses. In static analysis, a reflection-oriented slicing approach is applied to extract a small number of slices for a reflective call. Then in dynamic analysis, these slices are executed with automatically generated test cases to find out the reflective targets. Although the approach is neither sound nor complete, it can increase significantly the code coverage of dynamic reflection analysis while keeping false reflective targets low in practice.

Li et al. [29] propose an object-oriented dynamic symbolic execution framework for testing object-oriented libraries in Java. They support polymorphism by introducing constraints for method invocation targets and field manipulations via symbolic types. They have also generalized the notion of symbolic types to symbolic methods and symbolic fields to handle Java reflection symbolically.

Finally, the dynamic analyses [5, 24, 47] work in the presence of both dynamic class loading and reflection. Nguyen and Xue [44, 66] introduce an inter-procedural side-effect analysis for open-world Java programs by allowing dynamic class loading but disallowing reflection. Like other static reflection analyses [2, 22, 28, 32, 38, 51, 67], SOLAR can presently analyze closed-world Java programs only.

9 CONCLUSIONS

Reflection analysis is a long-standing hard open problem. In the past years, almost all the research papers consider Java reflection as a separate assumption and most static analysis tools either handle it partially or totally ignore it. In the program analysis community, people hold to the common opinion that “reflection is a dynamic feature, so how could it be handled effectively in static analysis?” This paper aims to change such informed opinion by showing that effective static analysis for handling Java reflection is feasible. We present a comprehensive understanding of Java reflection by illustrating its concept, interface design and real-world usage. Many useful findings are given to guide the design of more effective reflection analysis approaches and tools.

We introduce SOLAR, a soundness-guided reflection analysis, which is able to achieve significantly better soundness than previous work and can also accurately and automatically identify which reflective calls are resolved unsoundly or imprecisely. Such capabilities are helpful, in practice, as users can be aware of how sound an analysis is. In addition, for some clients where high-quality analysis results are needed (e.g., bug detectors with good precision or verification tools with good soundness) for a program, SOLAR provides an opportunity to help users obtain the analysis results.
as desired, by guiding them to add some lightweight annotations, if needed, to the parts of the program, where unsoundly or imprecisely resolved reflective calls are identified.

ACKNOWLEDGMENTS

The authors thank the reviewers for their constructive comments. This work was supported by the Australia Research Council (ARC) Grant No. DP170103956, and European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (Grant No. 647544).

REFERENCES


[51] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More Sound Static Handling of Java Reflection. In Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings (Lecture Notes in Computer Science), Xinyu Feng and Sungwoo Park (Eds.),