Effective Soundness-Guided Reflection Analysis

Yue Li, Tian Tan, and Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering, UNSW Australia

Abstract. We introduce SOLAR, the first reflection analysis that allows its soundness to be reasoned about when some assumptions are met and produces significantly improved under-approximations otherwise. In both settings, SOLAR has three novel aspects: (1) lazy heap modeling for reflective allocation sites, (2) collective inference for improving the inferences on related reflective calls, and (3) automatic identification of "problematic" reflective calls that may threaten its soundness, precision and scalability, thereby enabling their improvement via lightweight annotations. We evaluate SOLAR against two state-of-the-art solutions, DOOP and ELF, with the three treated as under-approximate reflection analyses, using 11 large Java benchmarks and applications. SOLAR is significantly more sound while achieving nearly the same precision and running only several-fold more slowly, subject to only 7 annotations in 3 programs.

1 Introduction

Reflection is increasingly used in a range of software and framework architectures, allowing a software system to choose and change implementations of services at run-time, but posing significant challenges to static program analysis. In the case of Java programs, reflection has always been an obstacle for pointer analysis [1–10], a fundamental static analysis on which virtually all others [11–16] are built. All pointer analysis tools for Java [2, 17–19] either ignore reflection or handle it partially since their underlying best-effort reflection analyses [5, 17, 18, 20–22] provide only under-approximated handling of reflection heuristically.

However, such unsoundness can render much of the codebase invisible for analysis. There is a recent community initiative [23] calling for the development of soundy analysis to handle "hard" language features (such as reflection). A *soundy* analysis is one that is as sound as possible without excessively compromising precision and/or scalability. Thus, improving or even achieving soundness in reflection analysis will provide significant benefits to many clients, such as program verifiers, optimizing compilers, bug detectors and security analyzers.

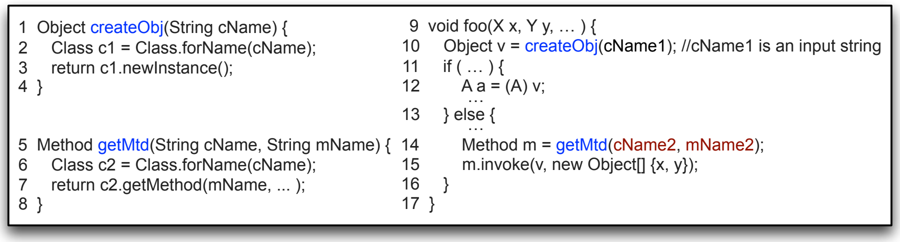
In this paper, we make the following contributions:

– We introduce SOLAR, the first reflection analysis that allows its soundness to be reasoned about when some reasonable assumptions are met and yields significantly improved under-approximations otherwise (Section 2). We have developed SOLAR by adopting three novel aspects in its design: (N1) lazy heap modeling for reflective allocation sites, (N2) collective inference for

related reflective calls, and (N3) automatic identification of "problematic" reflective calls that may threaten its soundness, precision and scalability.
– We formalize SOLAR as part of a pointer analysis for Java (including a small core of its reflection API) and reason about its soundness under a set of assumptions (Section 3). We have produced an open source implementation on top of DOOP [18], which is a modern pointer analysis tool for Java.
– We evaluate SOLAR against two state-of-the-art reflection analyses, DOOP [5] and ELF [21], with 11 large Java benchmarks/applications (Section 4), where all the three are treated as under-approximate analyses (due to, e.g., native code). By instrumenting these programs under their associated inputs (when available), SOLAR is the only one to achieve total recall (for all reflective targets accessed), with 371% (148%) more target methods resolved than DOOP (ELF) in total, which translates into 49700 (40570) more true caller-callee relations statically calculated w.r.t. these inputs alone. SOLAR has done so by maintaining nearly the same precision as and running only several-fold more slowly than ELF and DOOP, subject to only 7 annotations in 3 programs.

2 Methodology

Fig. 1 illustrates an example of reflection usage abstracted in real code. In line 2, a `Class` metaobject `c1` is created by calling `Class.forName(cName)` to represent the class named `cName`, where `cName`, i.e., `cName1` in line 10 is an input string to be read from a command line or a configuration file. In line 3, an object o is reflectively created as an instance of `c1` by calling `c1.newInstance()` and then assigned to `v` with the declared type as `Java.lang.Object` in line 10. Subsequently, o is used in two common scenarios. In the `if` branch, o is downcast to a specific type, `A`, and then used appropriately. The `else` branch is more interesting. In line 14, a `Method` metaobject `m` is created by calling `getMethod()` indirectly in line 7, with its class name, method name and formal parameters specified by `cName2`, `mName2` and "..." (elided) in line 7, respectively. In line 15, this method is called reflectively on the receiver object o (pointed to by `v`) with the actual argument being passed in an array, `new Object[] {x, y}`.



```
1 Object createObj(String cName) {
2    Class c1 = Class.forName(cName);
3    return c1.newInstance();
4 }

5 Method getMtd(String cName, String mName) {
6    Class c2 = Class.forName(cName);
7    return c2.getMethod(mName, ... );
8 }

 9 void foo(X x, Y y, ... ) {
10    Object v = createObj(cName1); //cName1 is an input string
11    if ( ... ) {
12       A a = (A) v;
          ...
13    } else {
          ...
14       Method m = getMtd(cName2, mName2);
15       m.invoke(v, new Object[] {x, y});
16    }
17 }
```

Fig. 1. An example of reflection usage abstracted from `JDK 1.6.0_45`.

A reflection analysis infers, i.e., resolves statically the reflective targets accessed at reflective call sites. As usual, soundnesss demands over-approximation. Reflection introduces many challenges for static analysis. First, a modern reflection API is large and complex. Second, reflection is typically used as a means of supporting dynamic adaptation of object-oriented software. As such, metaobjects are often created reflectively as shown in Fig. 1 from input strings. Thus, reflective object creation via `newInstance()` is hard to model statically. Finally, picking judicious approximations to balance soundness, precision and scalability is non-trivial. A simple-minded sound modeling of a reflective call (e.g., by assuming arbitrary behaviour) would destroy precision. Imprecision, in turn, often destroys scalability because too many spurious results would be computed.

Solar automates reflection analysis for Java by working with a pointer analysis. We first define some assumptions (Section 2.1). We then look at the three limitations of the prior work (Section 2.2). Finally, we introduce Solar to address these limitations by adopting three novel aspects in its design (Section 2.3).

2.1 Assumptions

The first three are made previously on reflection analysis for Java [20, 21]. The last one is introduced to allow reflective allocation sites to be modeled lazily.

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 conceivably download from the net 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`.*

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

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

As discussed in Section 4.2, Assumption 4 is found to hold for most reflective allocation sites in real code (as illustrated in Fig. 1). Here, (1) and (2) represent two kinds of usage points at which the class types of object o will be inferred lazily. This makes it possible to handle reflective allocation sites more accurately than before and to reason about the soundness of Solar for the first time.

2.2 Past Work: Best-Effort Reflection Resolution

All the existing solutions [5, 17, 18, 20–22] adopt a best-effort approach to reflection analysis, and consequently, suffer from the following three limitations:

L1. Eager Heap Modeling An abstract object o created at a call to, e.g., `c.newInstance()` is modeled eagerly if its type `c` can be inferred from a string constant or intraprocedural post-dominant cast, and ignored otherwise. Specifically, if `c` represents a known class name, e.g., “`A`”, then o’s type is “`A`”. Otherwise, an intraprocedurally post-dominating cast operation `(T)` operating on the result of the `newInstance()` call will allow `c` to be over-approximated as `T` or any of its subtypes. This eager approach often fails in real code shown in Fig. 1, where `cName1` is an input string and the cast is not post-dominating. Thus, its `newInstance()` call is ignored. Recently, in DOOP (r5459247-beta) [18], the objects created in line 10 (or line 3) are assumed to be of type `A` by taking advantage of the non-post-dominating cast `(A)` in line 12 to analyze more code. However, the objects with other types created along both the `if` and `else` branches are ignored. In prior work, such under-approximate handling of `newInstance()` is a significant source of unsoundness, as a large part of the program called on the thus ignored objects has been rendered invisible for analysis.

L2. Isolated Inferences Many reflective calls (e.g., those in Fig. 1) are related but analysed mostly in isolation, resulting in under-approximated behaviours. In [21], we presented a *self-inferencing* reflection analysis, called ELF, that can infer more targets at a reflective call site than before [5, 17, 18, 20, 22], by exploiting more information available (e.g., from its arguments and return type). However, due to eager heap modeling, ELF will still ignore the `invoke()` call in line 15 as `v` points to objects of unknown types as discussed above.

L3. Design-Time Soundness, Precision and Scalability When analysing a program heuristically, a best-effort approach does not know which reflective calls may potentially affect its soundness, precision and scalability. As a result, a developer is out of luck with a program if such best-effort analysis is either unscalable or scalable but with undesired soundness or precision or both.

2.3 SOLAR: Soundness-Guided Reflection Resolution

Fig. 2 illustrates the SOLAR design, with its three novel aspects marked by N1 – N3, where Ni is introduced to overcome the afore-mentioned limitation Li.

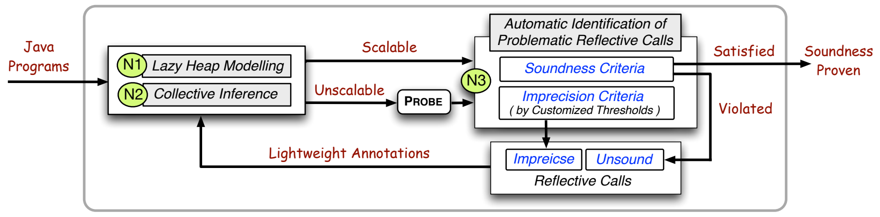


Fig. 2. SOLAR: a soundness-guided analysis with three novel aspects, N1 – N3.

N1. Lazy Heap Modeling (LHM) SOLAR handles reflective object creation lazily by delaying the creation of objects at their usage points where their types may be inferred, achieving significantly improved soundness and precision.

Let us describe the basic idea behind using the example in Fig. 1. As `cName` at `c1 = Class.forName(cName)` in line 2 is unknown, SOLAR will create a `Class` metaobject $\texttt{c1}^u$ that represents this unknown class and assign it to `c1`. As `c1` points to $\texttt{c1}^u$ at the allocation site `v = c1.newInstance()` in line 3, SOLAR will create an abstract object $\texttt{o}_3^u$ of an unknown type for the site to mark it as being unresolved yet. Subsequently, $\texttt{o}_3^u$ will flow into two usage points: Case (I) a type cast operation in line 12 and Case (II) a reflective method call site in line 15.

In Case (I), where $\texttt{o}_3^u$ is downcast to `A`, its type u is inferred to be `A` or any of its subtypes. Let $t_1, \ldots, t_n$ be all the inferred types. Then $\texttt{o}_3^u$ is split into n

distinct objects $\mathtt{o}_3^{t_1}$,..., $\mathtt{o}_3^{t_n}$ to be assigned to `a` in line 12. In Case (II), SOLAR will infer u by performing a collective inference as described below, based on the information available in line 15. Let $t'_1, \ldots, t'_m$ be all the inferred types. Then $\mathtt{o}_3^u$ is split into m distinct objects $\mathtt{o}_3^{t'_1}$,..., $\mathtt{o}_3^{t'_m}$ to be assigned to `v` in line 15.

According to Assumption 4 that states a key observation validated later, a reflectively created object like $\mathtt{o}_3^u$ is typically used in either Case (I) or Case (II) along every program path. The only but rare exception is that $\mathtt{o}_3^u$ is created but never used later. Then the corresponding constructor must be annotated to be analyzed statically unless ignoring it will not affect the points-to information.

N2. Collective Inference SOLAR builds on the prior work [5, 17, 18, 20, 22, 21] by relying on collective inference emphasized for the first time in reflection analysis. Let us return to the `invoke()` call, which cannot be analyzed previously. As `v` points to $\mathtt{o}_3^u$, SOLAR can infer u based on the information available at the call site. This happens when Case (1) `cName2` is known or Case (2) `cName2` is unknown but `mName2` is known. In SOLAR, inference is performed "collectively", whereby inferences on related reflective calls (lines 3, 6 and 15 for Case (1) and lines 3, 7 and 15 for Case (2)) can mutually reinforce each other. We will examine the second case, i.e., the more complex of the two, in Section 3.4.3. This paper is the first to do so by exploiting the connection between `newInstance()` (via LHM) and reflective calls for manipulating methods and fields.

N3. Automatic Identification of "Problematic" Reflective Calls Due to this capability, SOLAR is the first that can reason about its soundness. When such reasoning is not possible due to, e.g., native code, SOLAR reduces to an effective under-approximate analysis due to its soundness-guided design, allowing a disciplined tradeoff to be made among soundness, precision and scalability.

If SOLAR is scalable for a program, SOLAR can automatically identify "problematic" reflective calls (as opposed to reporting input strings as in [20]) that may threaten its soundness and precision to enable both to be improved with lightweight annotations. If SOLAR is unscalable for a program, a simplified version of SOLAR, denoted PROBE in Fig. 2, is called for next. With some "problematic" reflective calls to be annotated, SOLAR will re-analyze the program, scalably after one or more iterations of this "probing" process. We envisage providing a range of PROBE variants with different tradeoffs among soundness, precision and scalability, so that the scalability of PROBE is always guaranteed.

Consider Fig. 1 again. If both `cName2` and `mName2` are unknown (given that the type of o_3^u is unknown), then SOLAR will flag the `invoke()` call in line 15 as being potentially unsoundly resolved, detected automatically by verifying Condition (3) in Section 3.5. In addition, SOLAR will also automatically highlight reflective calls that may be potentially imprecisely resolved. Their lightweight annotations will allow SOLAR to yield improved soundness and precision.

Discussion Under Assumptions 1 – 4, we can establish the soundness of SOLAR by verifying a soundness criterion (given in Section 3.5). Otherwise, our soundness-guided approach has made SOLAR demonstrably more effective than existing under-approximate reflection analyses [5, 17, 20, 21] as validated later.

3 Formalism

We formalise Solar, illustrated in Fig. 2, for RefJava, which is Java restricted to a small core of its reflection API. Solar is flow-insensitive but context-sensitive. However, our formalisation is context-insensitive.

3.1 The RefJava Language

RefJava consists of all Java programs (under Assumptions 1 – 4) except that the Java reflection API is restricted to the four methods in Fig. 1: `Class.forName()`, `newInstance()`, `getMethod()` and `invoke()`. Our formalism is designed to allow its straightforward generalization to the entire API. For example, reflective field accesses via `getField()`, `get()` and `set()` can be handled similarly. As is standard, a Java program is represented only by five kinds of statements in the SSA form, as shown in Fig. 5. For simplicity, we assume that all the methods of a class accessed reflectively are its instance members, i.e., $\texttt{v} \neq \texttt{null}$ in `invoke(v,...)` in Fig. 1. We will discuss how to handle static members in Section 3.9.

3.2 Road Map

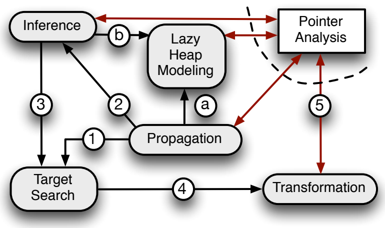


Fig. 3. Solar's inference system.

As depicted in Fig. 3, Solar's inference system, which consists of five components, works together with a pointer analysis. The arrow $\longleftrightarrow$ between a component and the pointer analysis indicates that each is both a producer and consumer of the other.

Let us see how Solar resolves the `invoke()` call in Fig. 1. If `cName2` and `mName2` are string constants, *Propagation* will create a `Method` metaobject (pointed to by `m`) carrying its known class and method information and pass it to *Target Search* (①). If `cName2` or `mName2` is not a constant, a `Method` metaobject marked as such is created and passed to *Inference* (②), which will infer the missing information and pass a freshly generated `Method` metaobject enriched with the missing information to *Target Search* (③). Then *Target Search* maps a `Method` metaobject to its reflective target *mtd* in its declaring class (④). Finally, *Transformation* turns the reflective call in line 15 into a regular call `v.`*mtd*`(...)` and pass it to the pointer analysis (⑤). *Lazy Heap Modeling* handles `newInstance()` in Fig. 1 to resolve the dynamic type of `v` based on the information discovered by *Propagation* (ⓐ) or *Inference* (ⓑ).

3.3 Notations

We will use the notations in Fig. 4. 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 declared or inherited. $\mathbb{CO}$ and $\mathbb{MO}$ represent the set of `Class` and `Method` metaobjects, respectively. In particular, $\texttt{c}^t$ denotes a `Class` metaobject of a known class t and $\texttt{c}^u$ a `Class` metaobject of an unknown class u. As illustrated earlier with Fig. 1, we write o_i^t

to represent an abstract object created at an allocation site i if it is an instance of a known class t and o_i^u of (an unknown class type) otherwise. For a `Method` metaobject, we write $\mathtt{m}_s^t$ if it is a member in a known class t and $\mathtt{m}_s^u$ otherwise, with its signature being s. In particular, we write $\mathtt{m}_u^-$ as a shorthand for $\mathtt{m}_s^-$ when s is unknown (with the return type $s.t_r$ being ignored), i.e., when $s.n_m = s.p = u$.

class type	$t \in \mathbb{T}$	`Class` metaobject	$\mathtt{c}^t, \mathtt{c}^u \in \mathbb{CO}$
method name	$n_m \in \mathbb{N}$	`Method` metaobject*	$\mathtt{m}_s^t, \mathtt{m}_u^t, \mathtt{m}_s^u, \mathtt{m}_u^u \in \mathbb{MO} = \widehat{\mathbb{T}} \times \mathbb{S}_m$
parameter types	$p \in \mathbb{P} = \mathbb{T}^0 \cup \mathbb{T}^1 \cup \mathbb{T}^2 \ldots$	method signature*	$s \in \mathbb{S}_m = \widehat{\mathbb{T}} \times \widehat{\mathbb{N}} \times \widehat{\mathbb{P}}$
method	$m \in \mathbb{M} = \mathbb{T} \times \mathbb{T} \times \mathbb{N} \times \mathbb{P}$	return type*	$s.t_r \in \widehat{\mathbb{T}}$
local variable	$\mathrm{c}, \mathrm{m} \in \mathbb{V}$	method name*	$s.n_m \in \widehat{\mathbb{N}}$
abstract heap object	$o_1^t, o_2^t, \ldots, o_1^u, o_2^u, \cdots \in \mathbb{H}$	parameter types*	$s.p \in \widehat{\mathbb{P}}$

Fig. 4. Notations ($\widehat{X} = X \cup \{u\}$, where u is an unknown class type or an unknown method signature). A superscript '*' marks a domain that contains u.

3.4 The Solar's Inference System

We present the inference rules used by all the components in Fig. 3, starting with the pointer analysis and moving to the five components of Solar.

3.4.1 Pointer Analysis Fig. 5 gives a standard formulation of a flow-insensitive Andersen's pointer analysis for RefJava. *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], the field accesses are handled.

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

$$\frac{\mathrm{x} = \mathrm{y}.m(\mathrm{arg}_1, \ldots,\ \mathrm{arg}_n) \quad o_i^- \in pt(\mathrm{y}) \quad m' = \mathit{dispatch}(o_i^-, m)}{\{o_i^-\} \subseteq pt(m'_{this}) \quad pt(m'_{ret}) \subseteq pt(\mathrm{x}) \quad \forall\ 1 \leqslant k \leqslant n : pt(\mathrm{arg}_k) \subseteq pt(m'_{pk})}\text{[A-Call]}$$

Fig. 5. Rules for *Pointer Analysis.*

In [A-Call] (for non-reflective calls), the function $\mathit{dispatch}(o_i^-, m)$ is used to resolve the virtual dispatch of method m on the receiver object o_i^- to be m' (when m is invokable on o_i^-). Following [24], we assume that m' has a formal parameter m'_{this} for the receiver object and $m'_{p1}, \ldots, m'_{pn}$ for the remaining parameters, and a pseudo-variable m'_{ret} is used to hold the return value of m'.

3.4.2 Propagation Fig. 6 gives the rules for `forName()` and `getMethod()` calls. Depending on whether their arguments are string constants or not, different kinds of `Class` and `Method` metaobjects are created. $\mathbb{SC}$ is a set of string constants and *toClass* returns a `Class` metaobject c^t, where t is the class specified by the string value returned by $val(o_i)$ (with $val : \mathbb{H} \rightarrow$ `java.lang.String`).

By design, $\mathtt{c}^t$ and $\mathtt{m}_s^t$ will flow to *Target Search* but all the others, i.e., $\mathtt{c}^u$, $\mathtt{m}_-^u$ and $\mathtt{m}_u^-$ will flow to *Inference*, where the missing information is inferred. During *Propagation*, only the name of a method signature s (i.e., $s.n_m$) can be discovered but its other parts are unknown: $s.t_r = s.p = u$.

$$\frac{\mathit{Class}\ \ \mathrm{c} = \mathit{Class.forName}(\mathrm{cName}) \quad o_i^{\texttt{String}} \in pt(\mathrm{cName})}{pt(\mathrm{c}) \supseteq \begin{cases} \{\mathtt{c}^t\} & \text{if } o_i^{\texttt{String}} \in \mathbb{SC} \\ \{\mathtt{c}^u\} & \text{otherwise} \end{cases} \qquad \mathtt{c}^t = toClass(val(o_i^{\texttt{String}}))}\text{[P-ForName]}$$

$$\frac{\mathit{Method}\ \ \mathrm{m} = \mathrm{c}'.\mathit{getMethod}(\mathrm{mName}, \ldots) \quad o_i^{\texttt{String}} \in pt(\mathrm{mName}) \quad \mathtt{c}^{-} \in pt(\mathrm{c}')}{pt(\mathrm{m}) \supseteq \begin{cases} \{\mathtt{m}_s^t\} & \text{if } \mathtt{c}^{-} = \mathtt{c}^t \wedge o_i^{\texttt{String}} \in \mathbb{SC} \\ \{\mathtt{m}_u^t\} & \text{if } \mathtt{c}^{-} = \mathtt{c}^t \wedge o_i^{\texttt{String}} \notin \mathbb{SC} \\ \{\mathtt{m}_s^u\} & \text{if } \mathtt{c}^{-} = \mathtt{c}^u \wedge o_i^{\texttt{String}} \in \mathbb{SC} \\ \{\mathtt{m}_u^u\} & \text{if } \mathtt{c}^{-} = \mathtt{c}^u \wedge o_i^{\texttt{String}} \notin \mathbb{SC} \end{cases} \qquad \begin{matrix} s.t_r = u \\ s.n_m = val(o_i^{\texttt{String}}) \\ s.p = u \end{matrix}}\text{[P-GetMtd]}$$

Fig. 6. Rules for *Propagation.*

3.4.3 Inference

Fig. 7 gives three rules to infer the reflective target methods for `x = (A) m.invoke(y,args)`, where `A` indicates a post-dominating cast on its result. If `A = Object`, then no such cast exists. In [I-InvTp], we use the types of the objects pointed to by `y` to infer the class types of the target methods called. Note that $\mathtt{m}_{-}^{t}$ represents a freshly generated `Method` metaobject. In [I-InvSig], we use the information available at a call site (excluding `y`) to infer the descriptor in the signature of a target method. In [I-InvS2T], we use the signature of a method to infer the class types of the method.

$$\frac{\mathrm{m}.\mathit{invoke}(\mathrm{y}, \mathrm{args}) \quad \mathtt{m}_{-}^{u} \in pt(\mathrm{m})}{pt(\mathrm{m}) \supseteq \{\, \mathtt{m}_{-}^{t} \mid o_i^t \in pt(\mathrm{y})\}}\text{[I-InvTp]} \qquad \frac{\mathrm{x} = (\mathsf{A})\ \mathrm{m}.\mathit{invoke}(\mathrm{y}, \mathrm{args}) \quad \mathtt{m}_u^{-} \in pt(\mathrm{m})}{pt(\mathrm{m}) \supseteq \{\, \mathtt{m}_s^{-} \mid s.p \in Ptp(\mathrm{args}), s.t_r \ll: \mathsf{A}, s.n_m = u\}}\text{[I-InvSig]}$$

$$\frac{\mathrm{x} = (\mathsf{A})\ \mathrm{m}.\mathit{invoke}(\mathrm{y}, \mathrm{args}) \quad \mathtt{m}_s^u \in pt(\mathrm{m}) \quad o_i^u \in pt(\mathrm{y}) \quad s.t_r \ll: \mathsf{A} \quad s.n_m \neq u \quad s.p \in Ptp(\mathrm{args})}{pt(\mathrm{m}) \supseteq \{\mathtt{m}_s^t \mid t \in \mathcal{M}(s.t_r, s.n_m, s.p)\}}\text{[I-InvS2T]}$$

Fig. 7. Rules for *Inference.*

As is standard, $t <: t'$ holds when t is t' or a subtype of t'. In [I-InvSig] and [I-InvS2T], $\ll:$ is used to take advantage of the post-dominating cast `(A)` during inference when `A` is not `Object`. By definition, $u \ll:$ `Object` holds. If t' is not `Object`, then $t \ll: t'$ holds if and only if $t <: t'$ or $t' <: t$ holds. The information on `args` is also exploited, where `args` is an array of type `Object[]`, only when it can be analyzed exactly element-wise by an intraprocedural analysis. In this case, suppose that `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, `args`$[i]$. Let $P_i = \{t' \mid t \in A_i, t <: t'\}$. Then $Ptp(\texttt{args}) = P_0 \times \cdots \times P_{n-1}$. Otherwise, $Ptp(\texttt{args}) = \varnothing$, implying that `args` is ignored as it cannot be exploited effectively during inference.

To maintain precision in [I-InvS2T], we use a method signature to infer its classes when both its name and descriptor are known. In this rule, the function $\mathcal{M}(s_{t_r}, s.n_m, s.p)$ returns the set of class types where the method with the specified signature s is declared if $s.n_m \neq u$ and $s.p \neq u$, and $\varnothing$ otherwise. The return type of the matching method is ignored if $s.t_r = u$.

Let us illustrate some of our rules by considering our example in Fig. 1.

Example 1. Note that `cName1` is an input string. Suppose that `cName2` is also an input string but `mName2` is a string constant. By applying [P-ForName], [P-GetMtd] and [L-UkwTp] (in Fig. 9) to the calls to `forName()` in lines 2 and 6, `getMethod()` and `newInstance()`, respectively, we obtain $\texttt{c1}^u \in pt(\texttt{c1})$, $\texttt{c2}^u \in pt(\texttt{c2})$, $\mathtt{m}_s^u \in pt(\mathtt{m})$ and $o_i^u \in pt(\mathtt{v})$, where s is a signature with a known method name in `mName2`. Given `args = new Object[] {x,y}`, $Ptp(args)$ is built as described earlier. SOLAR can infer the classes t where this method is declared by [I-INVS2T]. Finally, SOLAR will add all inferred `Method` objects $\mathtt{m}_s^t$ to $pt(\mathtt{m})$ at the call site. □

3.4.4 Target Search For a `Method` object $\mathtt{m}_s^t$ in a known class t (with s being possibly u), we define $MTD : \mathbb{MO} \to \mathcal{P}(\mathbb{M})$ to find all target methods matched:

$$MTD(\mathtt{m}_s^t) = \bigcup_{t <: t'} mtdLookUp(t', s.t_r, s.n_m, s.p) \tag{1}$$

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 and (2) any u that appears in s is treated as a wild card.

3.4.5 Transformation Fig. 8 gives the rules used for transforming a reflective call into a regular statement, which will be analyzed by the pointer analysis.

$$\frac{\begin{array}{c}\mathrm{x} = \mathrm{m}.\mathit{invoke}(\mathrm{y},\ \mathrm{args}) \quad \mathtt{m}_{\_}^t \in pt(\mathrm{m}) \quad m' \in MTD(\mathtt{m}_{\_}^t) \quad o_i^{\_} \in pt(\mathrm{args}) \\ o_j^{t'} \in pt(o_i^{\_}.arr) \quad t'' \text{ is declaring type of } m'_{pk} \quad k \in [1, n] \quad t' <: t''\end{array}}{\{o_j^{t'}\} \subseteq pt(\mathrm{arg}_k) \quad \mathrm{x} = \mathrm{y}.m'(\mathrm{arg}_1, ..., \mathrm{arg}_n)} \text{[T-INV]}$$

Fig. 8. Rules for *Transformation*.

Let us examine [T-INV] in more detail. The second argument `args` points to a 1-D array of type `Object[]`, with its elements collapsed to a single field arr during the pointer analysis, unless `args` can be analyzed exactly intraprocedurally in our current implementation. Let $\mathtt{arg}_1, \ldots, \mathtt{arg}_n$ be the n freshly created arguments to be passed to each potential target method m' found by *Target Search*. Let $m'_{p1}, \ldots, m'_{pn}$ be the n parameters (excluding *this*) of m', such that the declaring type of m'_{pk} is t''. We include $o_j^{t'}$ to $pt(\mathtt{arg}_k)$ only when $t' <: t''$ holds in order to filter out the objects that cannot be assigned to m'_{pk}. Finally, the regular call obtained can be analyzed by [A-CALL] in Fig. 5.

3.4.6 Lazy Heap Modeling Fig. 9 gives the rules for resolving `newInstance()` lazily. In [L-KWTP], for each `Class` object $\mathtt{c}^t$ pointed to by $\mathtt{c}'$, an object, o_i^t, is created as an instance of this known type at allocation site i straightaway. In [L-UKWTP], as illustrated with Fig. 1, o_i^u is created to enable LHM if $\mathtt{c}'$ points to a $\mathtt{c}^u$ instead. Then its lazy object creation happens at a type cast by applying [L-CAST] (with o_i^u blocked from flowing from `x` to `a`) and an `invoke()` call site by applying [L-INV]. Note that `A` is assumed not to be `Object` in [L-CAST].

$$\frac{i : \mathrm{v} = \mathrm{c}'.\mathit{newInstance}() \quad \mathtt{c}^t \in pt(\mathrm{c}')}{\{o_i^t\} \subseteq pt(\mathrm{v})}\text{[L-KWTP]} \qquad \frac{i : \mathrm{v} = \mathrm{c}'.\mathit{newInstance}() \quad \mathtt{c}^u \in pt(\mathrm{c}')}{\{o_i^u\} \subseteq pt(\mathrm{v})}\text{[L-UKWTP]}$$

$$\frac{\mathsf{A}\ a = (\mathsf{A})\,\mathrm{x} \quad o_i^u \in pt(\mathrm{x}) \quad t <: \mathsf{A}}{\{o_i^t\} \subseteq pt(a)}\text{[L-CAST]} \qquad \frac{\mathrm{x} = \mathrm{m}.\mathit{invoke}(\mathrm{y}, ...) \quad o_i^u \in pt(\mathrm{y}) \quad \mathtt{m}_{\_}^t \in pt(\mathrm{m}) \quad t' \ll: t}{\{o_i^{t'}\} \subseteq pt(\mathrm{y})}\text{[L-INV]}$$

Fig. 9. Rules for *Lazy Heap Modeling*.

3.5 Soundness Criterion

REFJAVA consists of the four methods from the Java reflection API as shown in Fig. 1. SOLAR is sound if their calls are resolved soundly under Assumptions 1 – 4. By construction, calls to `Class.forName()` and `getMethod()` are always soundly resolved (with the metaobjects created being modelled appropriately). Due to Assumption 4, there is no need to consider `newInstance()` calls since they are soundly resolved if all `invoke()` calls are. For convenience, we define:

$$AllKwn(v) = \nexists\, o_i^u \in pt(v) \tag{2}$$

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

Consider Fig. 7. For the `Method` metaobjects $\mathtt{m}_s^t$ 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]. For the `Method` objects $\mathtt{m}_s^u$ with unknown class types u, the targets accessed are inferred by [I-InvTp] and [I-InvS2T]. Let us consider a call to `(A) m.invoke(y, args)`. Solar attempts to infer the missing classes of its `Method` metaobjects in two ways, by applying [I-InvTp] and [I-InvS2T]. Such a call is soundly resolved if the following condition holds:

$$SC(\mathtt{m.invoke(y,args)}) = AllKwn(\mathtt{y}) \vee \forall\, \mathtt{m}_s^u \in pt(\mathtt{m}) : s.n_m \neq u \wedge Ptp(\mathtt{args}) \neq \varnothing \tag{3}$$

If the first disjunct holds, applying [I-InvTp] to `invoke()` can over-approximate its target methods from the types of all objects pointed to by `y`. Thus, every `Method` metaobject $\mathtt{m}_-^u \in pt(m)$ is refined into a new one $\mathtt{m}_-^t$ for every $o_i^t \in pt(\mathtt{y})$.

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 `args`. Thus, every `Method` metaobject $\mathtt{m}_s^u \in pt(m)$ is refined into a new one $\mathtt{m}_s^t$, where $s.t_r \ll: A$ and $s.p \in Ptp(\mathtt{args}) \neq \varnothing$. 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 = Object`. In this case, $u \ll:$ `Object` holds (only for u).

Theorem 1. Solar *is sound for* RefJava *if $SC(c)$ holds at every reflective call c of the form "`(A) m.invoke(y, args)`" under Assumptions 1 – 4.*

3.6 Identifying *Unsoundly* Resolved Reflective Calls

Solar flags a call c to `invoke()` as resolved unsoundly if $SC(c)$ is false. This can be conservative as some points-to information at c can be over-approximate. However, our evaluation shows that Solar can analyze 7 out of the 10 large programs considered scalably with full automation, implying that its inference system is powerful and precise. In addition, all 13 unsound calls reported by Solar in the remaining three programs are truly unsound, as discussed in Section 4.4, validating Solar's effectiveness in identifying unsoundness.

3.7 Identifying *Imprecisely* Resolved Reflective Calls

Presently, Solar performs this task depicted in Fig. 2, by simply ranking the reflective call sites according to the number of reflective targets inferred. This simple metric often gives a good indication about the sources of imprecision.

3.8 Probe

For evaluation purposes, we instantiate Probe, as shown in Fig. 2, from Solar as follows. We refrain from performing Solar's LHM (by retaining [L-UkwTp] but ignoring [L-Cast] and [L-Inv]) and abandon some of Solar's sophisticated inference rules (by disabling [I-InvS2T]). In *Target Search*, Probe will restrict itself to only `Method` metaobjects $\mathtt{m}_s^t$, where the signature s is at least partially known.

3.9 Static Class Members

To handle static class members, our rules can be modified. In Fig. 7, `y = null`. [I-InvTp] is not needed (by assuming $pt(\mathtt{null}) = \varnothing$). In (3), the first disjunct is removed. [I-InvS2T] is modified with $o_i^u \in pt(\mathtt{y})$ replaced by $\mathtt{y} = \mathtt{null}$. The rules in Fig. 8 are modified to deal with static members. In Fig. 9, [L-Inv] is no longer relevant. 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] and [L-Inv].

4 Evaluation

We have implemented Solar on top of Doop [18], a modern pointer analysis tool for Java. We compare Solar with two state-of-the-art under-approximate reflection analyses, Elf [21] and the reflection analysis provided in Doop (also referred to as Doop). In some programs, Assumptions 1 – 4 may not hold. Thus, Solar is also treated as being under-approximate. Due to its soundness-guided design, however, Solar can yield significantly better under-approximations than Doop and Elf. Like Doop and Elf, Solar is also implemented in the Datalog language. As far as we know, Solar is more comprehensive in handling the Java reflection API than the prior reflection analyses [2, 5, 17, 18, 20, 21].

In particular, our evaluation addresses the following research questions (RQs):

– **RQ1.** How well does Solar achieve full automation without using Probe?
– **RQ2.** How does Solar identify automatically "problematic" reflective calls affecting its soundness, precision and scalability, thereby facilitating their improvement by means of some lightweight annotations?
– **RQ3.** How significantly does Solar improve recall compared to Doop [18] and Elf [21], while maintaining nearly the same precision?
– **RQ4.** How does Solar scale in analysing large reflection-rich applications?

4.1 Experimental Setup

The three reflection analyses are compared by running each together with the same Doop pointer analysis framework (using its stable version r160113) [18]. For the Doop framework, we did not use its beta release (r5459247). The beta release handles a larger part of the Java reflection API but discovers fewer reflective targets in our recall experiment, since it ignores reflective targets whose class types are in the libraries (for efficiency reasons). All the three reflection analyses operate on the SSA form of a program emitted by Soot [19], context-sensitively under selective-2-type-sensitive+heap provided by Doop.

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`.

4.2 Assumptions

When analysing real code under-approximately, 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 [25]. 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 [22]. For the DaCapo benchmarks, `avrora` and `checkstyle`, their associated inputs are used. For `findbugs`, one Java program is developed as its input. For `freecs`, a server requiring user interactions, we only initialize it as the input in order to ensure repeatability. Assumptions 2 and 3 are taken for granted.

As for Assumption 4, we validate it for all reflective allocation sites where o_i^u 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 intraprocedural analysis. We have inspected the remaining 25% interprocedurally and found only two violating sites (in `eclipse` and `checkstyle`), where o_i^u is never used. In the other sites inspected, o_i^u flows through only local variables with all the call-chain lengths being at most 2.

4.3 RQ1: Full Automation

Fig. 10 compares Solar and existing reflection analyses [5, 17, 18, 20–22] denoted by "Others" by the degree of automation achieved. For an analysis, this is measured by the number of annotations required in order to improve the soundness of the reflective calls identified to be potentially unsoundly resolved. 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, 13 reflective calls are flagged as being potentially unsoundly resolved. After 7 annotations, 2 in `hsqldb`, 2 in `xalan` and 3 in `checkstyle`, Solar is scalable, as discussed in Section 4.4. However, Solar, like Doop and Elf, is unscalable (under 3 hours) for `jython`, an interpreter for Python in which the Java libraries and application code are invoked reflectively from the Python code.

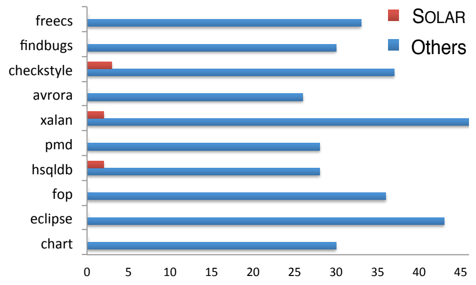


Fig. 10. The number of annotations required for improving the soundness of unsoundly resolved reflective calls.

"Others" cannot identify which reflective calls may be unsoundly resolved. However, they may improve soundness by requiring users to annotate the string arguments of calls to, e.g., `Class.forName()` and `getMethod()`, as suggested in [20]. As shown in Fig. 10, "Others" will require 338 annotations initially and possibly more in the subsequent iterations (when more code is discovered). As discussed in Section 2.3, Solar's annotation approach is also iterative. However, for these programs, Solar requires only 7 annotations in one iteration.

Solar outperforms "Others" due to its powerful inference system for performing reflection resolution and effective mechanism in identifying unsoundness.

4.4 RQ2: Automatically Identifying "Problematic" Reflective Calls

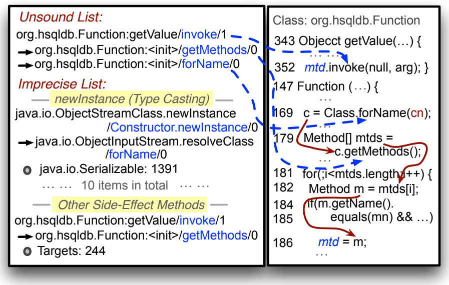
Solar is unscalable for `hsqldb`, `xalan` and `checkstyle` (under 3 hours). Probe is then run to identify their "problematic" reflective calls, reporting 13 potentially unsound calls: 1 `hsqldb`, 12 in `xalan` and 0 in `checkstyle`. Their handling is all unsound by code inspection, highlighting the effectiveness of Solar in pinpointing a small number of right parts of the program to improve unsoundness.

In addition, we presently 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 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 input strings), only lightweight annotations are needed as shown in Fig. 10, with 2 in `hsqldb`, 2 in `xalan` and 3 in `checkstyle`, as explained below.

4.4.1 `hsqldb`

Fig. 11 shows the unsound and imprecise lists automatically generated by Probe, together with the suggested annotation points (found by tracing value flow). All the call sites to the same method are numbered from 0.



```
Unsound List:
org.hsqldb.Function:getValue/invoke/1
 → org.hsqldb.Function:<init>/getMethods/0
 → org.hsqldb.Function:<init>/forName/0
Imprecise List:
   — newInstance (Type Casting) —
java.io.ObjectStreamClass.newInstance
                 /Constructor.newInstance/0
 → java.io.ObjectInputStream.resolveClass
                 /forName/0
  ● java.io.Serializable: 1391
      … …  10 items in total  … …
   — Other Side-Effect Methods —
org.hsqldb.Function:getValue/invoke/1
 → org.hsqldb.Function:<init>/getMethods/0
  ● Targets: 244
```

```
Class: org.hsqldb.Function
343 Objecct getValue(…) {
          … …
352  mtd.invoke(null, arg); }
147 Function (…) {
          …
169  c = Class.forName(cn);
          …
179  Method[] mtds =
             c.getMethods();
          …
181  for(;i<mtds.length;i++) {
182   Method m = mtds[i];
184   if(m.getName().
185         equals(mn) && …)
186     mtd = m;
          …
```

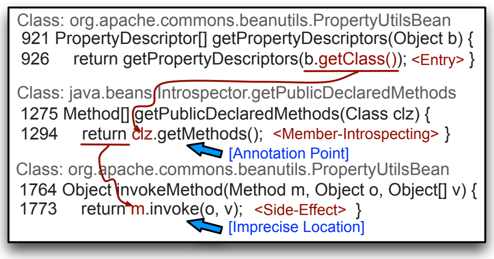
Fig. 11. Probing `hsqldb`.

The unsound list contains one `invoke()`, with its relevant code contained in class `org.hsqldb.Function` as shown. After Probe has finished, `mtd` in line 352 points to a `Method` metaobject $\mathtt{m}^u_u$ that is initially created in line 179 and later flows into line 182, indicating that the class type of $\mathtt{m}^u_u$ 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. So it has been annotated this way afterwards. The imprecise list for `hsqldb` is divided into two sections. In "newInstance (Type Casting)", there are 10 listed cast operations (T) reached by an o^u_i 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 due to imprecision in pointer analysis. Thus, we have annotated its corresponding `forName()` call site in method `resolveClass` of class `java.io.ObjectInputStream` to return nothing. With the two annotations, Solar terminates in 45 minutes with its unsound list being empty.

4.4.2 `xalan`

Probe reports 12 unsoundly resolved `invoke()` calls. All `Method` objects flowing into these call sites are created at 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 entry methods are all read from a file with its name hard-wired as `xmlspec.xsl` in this benchmark. For this particular 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.

4.4.3 `checkstyle` Probe reports no unsoundly resolved call. To see why Solar is unscalable, we examine one `invoke()` call in line 1773 of Fig. 12 found automatically by Probe that stands out as being possibly imprecisely resolved.



```
Class: org.apache.commons.beanutils.PropertyUtilsBean
921 PropertyDescriptor[] getPropertyDescriptors(Object b) {
926     return getPropertyDescriptors(b.getClass()); <Entry> }

Class: java.beans.Introspector.getPublicDeclaredMethods
1275 Method[] getPublicDeclaredMethods(Class clz) {
1294     return clz.getMethods();  <Member-Introspecting> }

Class: org.apache.commons.beanutils.PropertyUtilsBean
1764 Object invokeMethod(Method m, Object o, Object[] v) {
1773     return m.invoke(o, v);  <Side-Effect>  }
```

Fig. 12. Probing `checkstyle`.

There are 962 target methods inferred at this call site. Probe highlights its corresponding member-introspecting method `clz.getMethods()` (in line 1294) and its entry 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, the `clz.getMethods()` call is annotated 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.

Given the same annotations, existing reflection analyses [5, 17, 20, 21] still cannot handle the `invoke()` call in line 1773 soundly, because its argument `o` points to the objects that are initially created at a `newInstance()` call and then flow into a non-post-dominating cast operation (like the one in line 12 Fig. 1). However, Solar has handled this `invoke()` call soundly by using LHM, highlighting once again the importance of collective inference in reflection analysis.

4.5 RQ3: Recall and Precision

To compare the effectiveness of Doop, Elf and Solar as under-approximate reflection analyses, it is the most relevant to compare their *recall*, measured by the number of true reflective targets discovered at reflective call sites that are dynamically executed under certain inputs. In addition, we also compare their (static) analysis precision with two clients, but the results must be looked at with one caveat. Existing reflection analyses can happen to be "precise" due to their highly under-approximated handling of reflection. Therefore, our precision results are presented to show that Solar exhibits nearly the same precision as prior work despite its significantly improved recall achieved for real code.

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 Fig. 10 are used by all the three analyses.

4.5.1 Recall We use TamiFlex [22] to find the targets accessed at reflective calls in our programs under the inputs described in Section 4.2. Solar is the only one to achieve total recall for all reflective targets accessed.

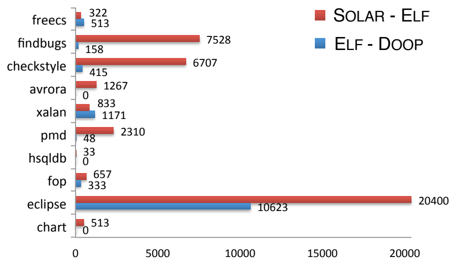


Fig. 13. More true caller-callee relations found in recall by Solar than Elf (Solar−Elf) and by Elf than Doop (Elf−Doop).

Here, we demonstrate one significant benefit of achieving higher recall, in practice. Fig. 13 compares Doop, Elf and Solar in terms of true caller-callee relations statically calculated and obtained by an instrumental tool written in terms of Javassist [26]. Solar recalls a total of 371% (148%) more targets than Doop (Elf) at the calls to `newInstance()` and `invoke()`, translating into 49700 (40570) more true caller-callee relations found for the 10 programs. These numbers are expected to improve when more inputs are used. Note that all targets recalled by Doop are recalled by Elf and all targets recalled by Elf are recalled by Solar. These results demonstrate the effectiveness of our LHM and collective inference.

Table 1. Precision comparison. There are two clients: DevirCall denotes the percentage of the virtual calls whose targets can be disambiguated and SafeCast denotes the percentage of the casts that can be statically shown to be safe.

		chart	eclipse	fop	hsqldb	pmd	xalan	avrora	checkstyle	findbugs	freecs	**Average**
Devir	Doop	–	94.94	93.04	–	92.65	93.49	94.79	93.16	92.32	95.46	93.72
Call	Elf	93.53	88.07	92.34	94.80	92.87	92.70	94.50	93.19	92.53	94.94	92.93
(%)	Solar	93.51	87.69	92.26	94.51	92.39	92.65	92.43	93.39	92.37	95.26	92.63
Safe	Doop	–	59.34	53.68	–	45.40	57.97	56.12	50.19	45.78	59.71	53.24
Cast	Elf	49.80	40.71	55.40	53.65	48.24	59.24	57.27	51.79	48.54	59.14	52.07
(%)	Solar	49.53	38.04	54.21	53.11	44.53	59.11	52.56	49.40	43.60	57.96	49.79

4.5.2 Precision Tables 1 compares the precision of Doop, Elf and Solar with two popular clients. Note that Doop is unscalable for `chart` and `hsqldb` (under 3 hours) in our setting. Despite achieving better recall (Fig. 13), Solar maintains nearly the same precision as Doop and Elf, which tend to be more under-approximate than Solar. This suggests that Solar's soundness-guided design is effective in balancing soundness, precision and scalability.

4.6 RQ4: Efficiency

Table 2 compares the analysis times of Doop, Elf and Solar. Despite producing significantly better under-approximations than Doop and Elf, Solar is only several-fold slower. When analysing `hsqldb`, `xalan` and `checkstyle`, Solar requires some lightweight annotations. Their analysis times are the ones consumed by Solar on analysing the annotated programs. Note that these annotated programs are also used by Doop and Elf (as discussed earlier).

5 Related Work

In addition to the prior work already discussed in Section 2.2, we highlight below a few open-source static reflection analysis tools available. bddbddb [2] represents a partial implementation of the reflection analysis introduced in [20].

Table 2. Efficiency comparison (secs).

	chart	eclipse	fop	hsqldb	pmd	xalan	avrora	checkstyle	findbugs	freecs	**Average**
Doop	–	321	779	–	226	254	188	256	718	422	–
Elf	3434	5496	2821	1765	1363	1432	932	1463	2281	1259	1930
Solar	4543	10743	4303	2695	2156	1701	3551	2256	8489	2880	3638

Doop [5, 18] is a pointer analysis framework for Java programs written in Datalog. Its reflection handling was similar to the reflection analysis in [20] except that it is done context-sensitively. Doop can now accept the analysis results of TamiFlex [22] on a program while analyzing the placeholder library generated by Averroes [27], which presently models only `newInstance()` and `invoke()`.

Elf [21] represents a recent reflection analysis, implemented in Doop, for Java, by leveraging a so-called self-inferencing property inherent in a program. However, Elf opts to trade soundness for precision by inferring a target at a reflective call if and only if both its signature and declaring class can be inferred. Building on this, Solar advocates collective inference to improve and even achieve soundness under Assumptions 1 – 4, facilitated by lazy heap modeling for reflective object creation. Solar benefits greatly from the open-source code of Elf and Doop. However, to the best of our knowledge, Solar is the most comprehensive analysis in handling the Java reflection API.

Wala [17] provides static analysis capabilities for Java and other languages like JavaScript. Its reflection handling is similar to [20] (by resolving values of string arguments of reflective calls) but does not handle `Field`-related methods.

6 Conclusion

Achieving soundness in reflection analysis can improve the effectiveness of many clients such as program verifiers, compilers, bug detectors and security analyzers. However, reflection is very challenging to analyze effectively, particularly for reflection-heavy applications. In this paper, we make one significant step forward by introducing a new reflection analysis that can reason about its soundness when certain assumptions are met and produce significantly improved under-approximations than prior art otherwise. We hope that our framework (`www.cse.unsw.edu.au/~corg/solar`) will be useful in future research.

7 Acknowledgements

The authors wish to thank the POPL 2015 and SAS 2015 reviewers for their comments, the Doop team for making Doop available, and LogicBlox Inc. for providing us its Datalog engine. This work is supported by ARC grants, DP130101970 and DP150102109.

References

1. O. Lhoták and L. Hendren, “Scaling Java points-to analysis using Spark,” CC ’03.
2. J. Whaley and M. S. Lam, “Cloning-based context-sensitive pointer alias analysis using binary decision diagrams,” PLDI ’04.

3. A. Milanova, A. Rountev, and B. G. Ryder, “Parameterized object sensitivity for points-to analysis for Java,” *ACM Trans. Softw. Eng. Methodol.*, 2005.
4. M. Sridharan and R. Bodík, “Refinement-based context-sensitive points-to analysis for Java,” PLDI ’06.
5. M. Bravenboer and Y. Smaragdakis, “Strictly declarative specification of sophisticated points-to analyses,” OOPSLA ’09.
6. L. Shang, X. Xie, and J. Xue, “On-demand dynamic summary-based points-to analysis,” CGO ’12.
7. Y. Lu, L. Shang, X. Xie, and J. Xue, “An incremental points-to analysis with CFL-reachability,” in *CC ’13*.
8. Y. Smaragdakis, M. Bravenboer, and O. Lhoták, “Pick your contexts well: understanding object-sensitivity,” POPL ’11.
9. G. Kastrinis and Y. Smaragdakis, “Hybrid context-sensitivity for points-to analysis,” PLDI ’13.
10. Y. Smaragdakis, G. Kastrinis, and G. Balatsouras, “Introspective analysis: Context-sensitivity, across the board,” PLDI ’14.
11. M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind, “Fast online pointer analysis,” *ACM Trans. Program. Lang. Syst*, 2007.
12. M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg, “F4F: Taint analysis of framework-based web applications,” OOPSLA ’11.
13. X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang, “On abstraction refinement for program analyses in datalog,” PLDI ’14.
14. S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, “Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps,” PLDI ’14.
15. P. H. Nguyen and J. Xue, “Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading,” in *ACSC ’05*.
16. J. Xue and P. H. Nguyen, “Completeness analysis for incomplete object-oriented programs,” in *CC ’05*.
17. WALA, “T.J. Watson libraries for analysis.” `http://wala.sf.net`.
18. DOOP. `http://doop.program-analysis.org`.
19. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, “Soot - a Java bytecode optimization framework,” CASCON ’99.
20. B. Livshits, J. Whaley, and M. S. Lam, “Reflection analysis for Java,” APLAS ’05.
21. Y. Li, T. Tan, Y. Sui, and J. Xue, “Self-inferencing reflection resolution for Java,” ECOOP’ 14.
22. E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini, “Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders,” ICSE ’11.
23. B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhotk, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Mller, and D. Vardoulakis, “In defense of soundiness: A manifesto,” *Communications of the ACM*, 2015.
24. M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, ch. Alias Analysis for Object-Oriented Programs. 2013.
25. J. Sawin and A. Rountev, “Improving static resolution of dynamic class loading in java using dynamically gathered environment information,” *Automated Software Engg.*, 2009.
26. Javassist, “A Java bytecode manipulation framework.” `http://www.javassist.org`.
27. K. Ali and O. Lhoták, “Averroes: Whole-program analysis without the whole program,” ECOOP’13.