Self-Inferencing Reflection Resolution for Java

Yue Li, Tian Tan, Yulei Sui, and Jingling Xue

School of Computer Science and Engineering, UNSW Australia
`{yueli,tiantan,ysui,jingling}@cse.unsw.edu.au`

Abstract. Reflection has always been an obstacle both for sound and for effective under-approximate pointer analysis for Java applications. In pointer analysis tools, reflection is either ignored or handled partially, resulting in missed, important behaviors. In this paper, we present our findings on reflection usage in Java benchmarks and applications. Guided by these findings, we introduce a static reflection analysis, called Elf, by exploiting a *self-inferencing property* inherent in many reflective calls. Given a reflective call, the basic idea behind Elf is to automatically infer its targets (methods or fields) based on the dynamic types of the arguments of its target calls and the downcasts (if any) on their returned values, if its targets cannot be already obtained from the `Class`, `Method` or `Field` objects on which the reflective call is made. We evaluate Elf against Doop's state-of-the-art reflection analysis performed in the same context-sensitive Andersen's pointer analysis using all 11 DaCapo benchmarks and two applications. Elf can make a disciplined tradeoff among soundness, precision and scalability while also discovering usually more reflective targets. Elf is useful for any pointer analysis, particularly under-approximate techniques deployed for such clients as bug detection, program understanding and speculative compiler optimization.

1 Introduction

Pointer analysis is an important enabling technology since it can improve the precision and performance of many program analyses. However, reflection poses a major obstacle to pointer analysis. Despite the large literature on whole-program [1, 6, 7, 11, 15, 21] and demand-driven [10, 13, 14, 17] pointer analysis for Java, almost all the analyses reported are unsound in the presence of reflection since it is either ignored or handled partially. As a result, under-approximate or unsound techniques represent an attractive alternative in cases where sound analysis is not required [18] (e.g., for supporting bug detection, program understanding and speculative compiler optimization). Even so, ignoring reflection often leads to missed, important behaviors [18]. This explains why modern pointer analysis tools for Java [4, 19–21] provide some forms of reflection handling.

As reflection is increasingly used in Java programs, the cost of imprecise reflection handling has increased dramatically. To improve the effectiveness of a pointer analysis tool for Java, automatic techniques for handling reflection by balancing soundness, precision and scalability are needed. Despite its importance, this problem has received little attention. Some solutions include (1) dy-

```
 1  A a = new A();
 2  String cName, mName, fName = ...;
 3  Class clz = Class.forName(cName);
 4  Object obj = clz.newInstance();
 5  B b = (B)obj;
 6  Method mtd = clz.getDeclaredMethod(mName,{A.class});
 7  Object l = mtd.invoke(b, {a});
 8  Field fld = clz.getField(fName);
 9  X r = (X)fld.get(a);
10  fld.set(NULL, a);
```

Fig. 1. An example of reflection usage in Java.

namic analysis [2] for recording reflective (call) targets discovered during input-dependent program runs and passing these annotations to a subsequent pointer analysis, (2) online analysis [5] for discovering reflective targets at run time and performing a pointer analysis to support JIT optimizations, and (3) static analysis [4, 8, 20] for resolving reflective targets together with a pointer analysis.

In this paper, we present a new static reflection analysis, called Elf, which is integrated into Doop, a state-of-the-art Datalog-based pointer analysis tool [4] for analyzing Java programs. Elf draws its inspirations from the two earlier reflection analyses [4, 8] and benefits greatly from the open-source reflection analysis implemented in Doop [4]. Livshits et al. [8] suggested resolving reflective calls by tracking the flow of class/method/field names in the program. In the code from Figure 1, this involves tracking the flow of `cName` into `clz` in line 3, `mName` into `mtd` in line 6, and `fName` into `fld` in line 8, if `cName`, `mName` and `fName` are string constants. If `cName` is, say, read from a configuration file, they suggested narrowing the types of reflectively-created objects, e.g., `obj` in line 4, optimistically by using the downcast `(B)` available in line 5. Later, Doop [4] handles reflection analogously, but context-sensitively, to obtain the full benefit from the mutual increase in precision of both component analyses.

However, Elf goes beyond [4, 8] by taking advantage of a *self-inferencing property* inherent in reflective code to strike a disciplined tradeoff among soundness, precision and scalability. Our key observation (made from a reflection-usage study described in Section 2) is that many reflective calls are *self-inferenceable*. Consider `r = (X)fld.get(a)` in Figure 1. Its target fields accessed can often be approximated based on the dynamic types (i.e., `A`) of argument `a` and the downcast that post-dominates its return values, if `fld` represents a statically unknown field named `fName`. In this case, the reflective call is resolved to all possible field reads `r = a.f`. Here, `f` is a field of type T (where T is `X` or a supertype or subtype of `X`), declared in a class C (where C is `A` or a supertype of `A`). To the best of our knowledge, Elf is the first static reflection analysis that exploits such self-inferencing property to resolve reflective calls.

Due to the intricacies and complexities of the Java reflection API, we will postpone a detailed comparison between Elf and the two state-of-the-art reflection analyses [4, 8] later in Section 3 after we have introduced Elf in full.

In summary, this paper makes the following main contributions:

– We report findings on a reflection-usage study using 14 representative Java benchmarks and applications (Section 2). We expect these findings to be useful in guiding the design and implementation of reflection analysis.
– We introduce a static reflection analysis, Elf, to improve the effectiveness of pointer analysis tools for Java (Section 3). Elf adopts a new *self-inferencing* mechanism for reflection resolution and handles a significant part of the Java reflection API that was previously ignored or handled partially.
– We formulate Elf in Datalog consisting of 207 rules, covering the majority of reflection methods frequently used in Java programs (Section 4).
– We have evaluated Elf against a state-of-the-art reflection analysis in Doop (version r160113) under the same context-sensitive Andersen's pointer analysis framework, using all 11 DaCapo benchmarks and two Java applications, `Eclipse4` and `Javac` (Section 5). Our results show that Elf can make a disciplined tradeoff among soundness, precision and scalability while resolving usually more reflective call targets than Doop.

2 Understanding Reflection Usage

Section 2.1 provides a brief introduction to the Java reflection API. Section 2.2 reports our findings on reflection usage in Java benchmarks and applications.

2.1 Background

The Java reflection API provides metaobjects to allow programs to examine themselves and make changes to their structure and behavior at run time. In Figure 1, 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` provides accessor methods such as `getDeclaredMethod()` in line 6 and `getField` in line 8 to allow the other 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 7) and a `Field` object can be commanded to access the field that it represents (lines 9 and 10).

As far as pointer analysis is concerned, we can divide the pointer-affecting methods in the Java reflection API into three categories: (1) *entry methods*, e.g., `forName()` in line 3, for creating `Class` objects, (2) *member-introspecting methods*, e.g., `getDeclaredMethod()` in line 6 and `getField()` in line 8, for retrieving `Method` (`Constructor`) and `Field` objects from a `Class` object, and (3) *side-effect methods*, e.g., `newInstance()`, `invoke()`, `get()` and `set()` in lines 4, 7, 9 and 10, that affect the pointer information in the program reflectively.

`Class` provides a number of accessor methods for introspecting methods, constructors and fields in a target class. Unlike [4, 8], Elf is the first to handle all such accessor methods in reflection analysis. Let us recall the four on returning `Method` objects. `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

6 in Figure 1). `getMethod(String, Class[])` is similar 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 except that all the public methods (either declared or inherited) in the target `Class` object are returned. Given a `Method` object `mtd`, its target method can be called as shown in line 7 in Figure 1.

2.2 Empirical Study

The Java reflection API is rich and complex in details. We conduct an empirical study to understand reflection usage in practice in order to guide the design and implementation of a sophisticated reflection analysis.

We select 14 representative Java programs, including nine DaCapo benchmarks (2006-10-MR2), three latest versions of popular desktop applications, `javac-1.7.0`, `jEdit-5.1.0` and `Eclipse-4.2.2` (denoted `Eclipse4`), and two latest versions of popular server applications, `Jetty-9.0.5` and `Tomcat-7.0.42`. Note that DaCapo consists of 11 benchmarks, including an older version of `Eclipse` (version 3.1.2). We exclude `bloat` 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 1.5), including the ones in `java.lang.reflect` and `java.lang.Class`, `loadClass()` in `java.lang.ClassLoader`, and `getClass()` in `java.lang.Object`. We have also considered `A.`*class*, which represents the `Class` object of a class `A`.

We use Soot [19] to pinpoint the calls to reflection methods in the bytecode of a program. To understand 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 enable Soot to locate all the other jar files used. We manually inspect the reflection usage in a program in a demand-driven manner, starting from its side-effect methods, assisted by *Open Call Hierarchy* in `Eclipse`, by following their backward slices. For a total of 609 side-effect callsites examined, 510 callsites for calling entry methods and 304 callsites for calling member-introspecting methods are tracked and analyzed.

Below we describe our five findings on reflection usage in our empirical study.

Side-Effect Methods Table 1 lists a total of nine side-effect methods that can possibly modify or use (as their side effects) the pointer information in a program. Figure 2 depicts their percentage frequency distribution in the 14 programs studied. We can see that `invoke()` and `Class::newInstance()` are the two most frequently used (32.7% and 35.3%, respectively, on average), which are

Table 1. Nine side-effect methods and their side effects, assuming that the target class of *clz* and *ctor* is *A* and the target method (field) of *mtd* (*fld*) is *m* (*f*).

Simplified Method	Calling Scenario	Side Effect
Class::newInstance	o = *clz*.newInstance()	o = new *A*()
Constructor::newInstance	o = *ctor*.newInstance({arg$_1$, ...})	o = new *A*(arg$_1$, ...)
Method::invoke	a = *mtd*.invoke(o, {arg$_1$, ...})	a = o.*m*(arg$_1$, ...)
Field::get	a = *fld*.get(o)	a = o.*f*
Field::set	*fld*.set(o, a)	o.*f* = a
Array::newInstance	o = *Array*.newInstance(*clz*, size)	o = new *A*[size]
Array::get	a = *Array*.get(o, i)	a = o[i]
Array::set	*Array*.set(o, i, a)	o[i] = a
Proxy::newProxyInstance	o = *Proxy*.newProxyInstance(...)	o = new Proxy$*(...)

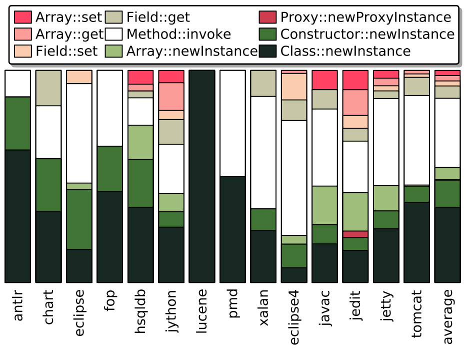


Fig. 2. Side-effect methods.

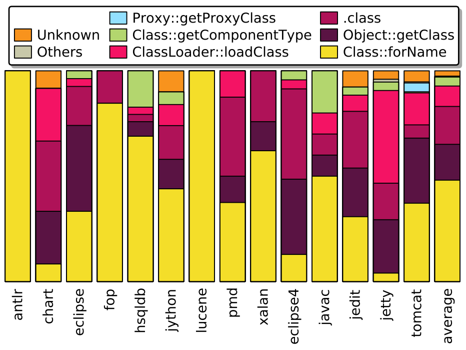
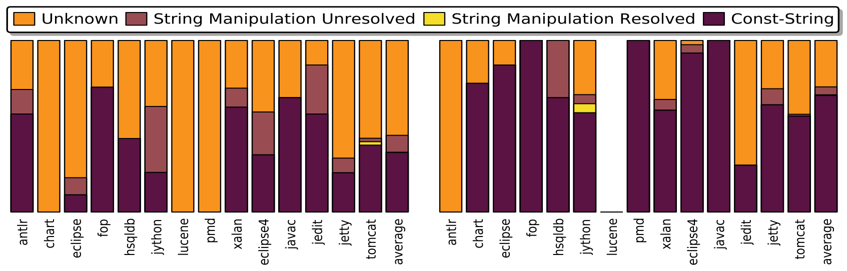


Fig. 3. Entry methods.

handled by prior pointer analysis tools [4, 20, 21]. However, `Array`-related side-effect methods, which are also used in many programs, are previously ignored but handled by ELF. Note that `newProxyInstance()` is used in `jEdit` only.

Entry Reflection Methods Figure 3 shows the percentage frequency distribution of different types of entry methods used. The six as shown are the only ones found in the first 12 programs. In the last two (`Jetty` and `Tomcat`), "*Others*" stands for `defineClass()` in `ClassLoader` and `getParameterTypes()` in `Method` only. "*Unknown*" is included since we failed to find the entry methods for some side-effect calls such as `invoke()` even by using `Eclipse`'s *Open Call Hierarchy* tool. Finally, `getComponentType()` is usually used in the form of `getClass().getComponentType()` for creating a `Class` object argument for `Array.newInstance()`. On average, `Class.forName()` and `.class` are the top two most frequently used entry methods (48.1% and 18.0%, respectively).

String Constants and String Manipulation As shown in Figure 4, string constants are commonly used when calling the two entry methods (34.7% on average) and the four member-introspecting 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 *precisely* handling of many different operations e.g., `subString()` and `append()`. Thus, ELF does



(a) Calls to entry methods (b) Calls to member-introspecting methods

Fig. 4. Classification of the `String` arguments of two entry methods, `forName()` and `loadClass()`, and four member-introspecting methods, `getMethod()`, `getDeclaredMethod()`, `getField()` and `getDeclaredField()`.

not handle string manipulations presently. As suggested in Section 5.3.2, however, incomplete information about class/method/field names can be exploited in our self-inferencing framework, just like the cast and type information.

We also found that many string arguments are *Unknown* (55.3% for calling entry methods and 25.1% for calling member-introspecting methods, on average). These are the strings that may be read from, say, configuration files or command lines. Finally, string constants are found to be more frequently used for calling the four member-introspecting methods than the two entry 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 [4, 20] that ignore string constants flowing into some of these member-introspecting methods may be imprecise (Table 2).

Self-Inferenceable Reflective Calls In real applications, many reflective calls are self-inferenceable, as illustrated in Figures 8 – 10. Therefore, we should try to find their targets by aggressively tracking the flow of constant class/method/field names in the program. However, there are also many input-dependent strings. For many input-dependent reflective calls, such as `factoryField.get(null)` in Figure 8, `field.set(null, value)` in Figure 9 and `method.invoke(target, parameters)` in Figure 10, we can approximate their targets reasonably accurately based on the dynamic types of the arguments of their target calls and the downcasts (if any) on their returned values. ELF will exploit such self-inferencing property inherent in reflective code during its reflection analysis.

Retrieving an Array of `Method/Field/Constructor` Objects `Class` contains a number of accessor methods for returning an array of such metaobjects for the target `Class` object. In the two `Eclipse` programs, there are four `invoke` callsites called on an array of `Method` objects returned from `getMethods` and 15 `fld.get()` and `fld.set()` callsites called on an array of `Field` objects returned by `getDeclaredFields()`. Ignoring such methods as in prior work [4, 8, 21] may lead to many missed methods in the call graph of a program.

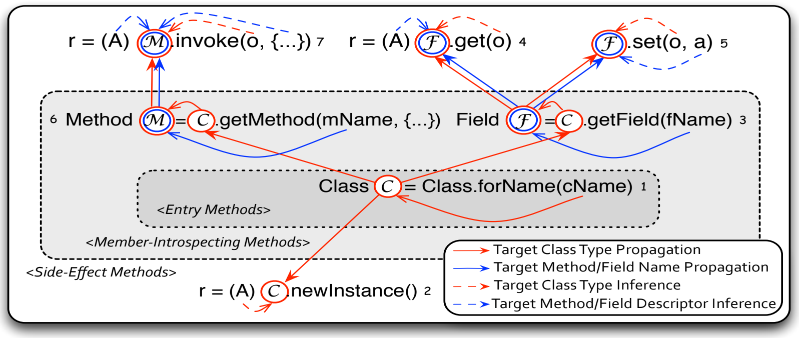

r = (A) $\mathcal{M}$.invoke(o, {...}) 7
r = (A) $\mathcal{F}$.get(o) 4
$\mathcal{F}$.set(o, a) 5
6 Method $\mathcal{M}$ = $\mathcal{C}$.getMethod(mName, {...})
Field $\mathcal{F}$ = $\mathcal{C}$.getField(fName) 3
Class $\mathcal{C}$ = Class.forName(cName) 1
<Entry Methods>
<Member-Introspecting Methods>
<Side-Effect Methods>
r = (A) $\mathcal{C}$.newInstance() 2
Target Class Type Propagation
Target Method/Field Name Propagation
Target Class Type Inference
Target Method/Field Descriptor Inference

Fig. 5. Self-inferencing reflection analysis in ELF.

3 Methodology

We start with a set of assumptions made. We then describe our self-inferencing approach adopted by ELF. Finally, we compare ELF with the two prior reflection analyses [4, 8] by summarizing their similarities and differences.

3.1 Assumptions

We adopt all the assumptions from [8]: (1) *Closed World*: only the classes reachable from the class path at analysis time can be used by the program at run time, (2) *Well-behaved Class Loaders*: the name of the class returned by a call to `forName(cName)` equals `cName`, and (3) *Correct Casts*: the downcasts operating on the result of a call to `newInstance()` are correct. Due to (1), we will not consider the side-effect method `Proxy::newProxyInstance` in Table 1 and the entry method `loadClass` in Figure 3 as both may use custom class loaders. Finally, we broaden *Correct Casts* by also including `fld.get()` and `mtd.invoke()`.

3.2 Self-Inferencing Reflection Resolution

Figure 5 depicts a typical reflection scenario and illustrates how ELF works. In this scenario, a `Class` object $\mathcal{C}$ is first created for the target class named `cName`. Then a `Method` (`Field`) object $\mathcal{M}$ ($\mathcal{F}$) representing the target method (field) named `mName` (`fName`) in the target class of $\mathcal{C}$ is created. Finally, at some reflective callsites, e.g., `invoke()`, `get()` and `set()`, the target method (field) is invoked (accessed) on the target object `o`, with the arguments, `{...}` or `a`. In the case of `newInstance()`, the default constructor "`init()`" called is implicit.

ELF works as part of a pointer analysis, with each being both the producer and consumer of the other. It exploits a self-inferencing property inherent in reflective code, by employing the following two component analyses (Figure 5):

Target Propagation (Marked by Solid Arrows) ELF resolves the targets (methods or fields) of reflective calls, such as `invoke()`, `get()` and `set()`, by propagating the names of the 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. Note that the second argument of `getMethod()` is an array of type `Class[]`. It may not be beneficial to analyze it to disambiguate overloaded methods, because (1) its size may be statically unknown, (2) its components are collapsed by the pointer analysis, and (3) its components may be `Class` objects with unknown class names.

Target Inference (Marked by Dashed Arrows) By using *Target Propagation* alone, a target method/field name (blue circle) or its target class type (red circle) at a reflective callsite may be missing, i.e., unknown, due to the presence of input-dependent strings (Figure 4). If the target class type (red circle) is missing, ELF will infer it from the dynamic type of the target object `o` (obtained by pointer analysis) at `invoke()`, `get()` or `set()` (when `o != null`) or the downcast (if any), such as `(A)`, that post-dominantly operates on the result of a call to `newInstance()`. If the target method/field name (blue circle) is missing, ELF will infer it from (1) the dynamic types of the arguments of the target call, e.g., `{...}` of `invoke()` and `a` of `set()`, and/or (2) the downcast on the result of the call, such as `(A)` at `invoke()` and `get()`. Just like `getMethod`, the second argument of `invoke()` is also an array, which is also similarly hard to analyze statically. To improve precision, we disambiguate overloaded target methods with a simple intraprocedural analysis only when the array argument can be analyzed exactly element-wise.

To balance soundness, precision and scalability in a disciplined manner, ELF adopts the following inference principle: *a target method or field is resolved at a reflective callsite if both its target class type (red circle) and its target method/field name (blue circle) can be resolved (i.e., statically discovered) during either Target Propagation or Target Inference.* As a result, the number of spurious targets introduced when analyzing a reflective call, `invoke()`, `get()` or `set()`, is minimized due to the existence of two simultaneous constraints (the red and blue circles). How to relax ELF in the presence of just one such a constraint will be investigated in future work. Note that the cast operations on `newInstance()` will still have to be handled heuristically as only one of the two constraints exists. As ELF is unsound, so is the underlying pointer analysis. Therefore, a reflective callsite is said to be *resolved* if at least one of its targets is resolved.

Let us illustrate *Target Inference* by considering `r = (A)` $\mathcal{F}$`.get(o)` in Figure 5. If a target field name is known but its target class type (i.e., red circle) is missing, we infer it by looking at the types of all pointed-to objects o' by `o`. If B is the type of o', then a potential target class of `o` is B or any of its supertypes. If the target class type of $\mathcal{F}$ is B but a potential target field name (i.e., blue circle) is missing, we can deduce it from the downcast `(A)` to resolve the call to `r = 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 of this supertype may initially point to an object of type, say, `A` and then downcast to `A`.

In Figure 5, if `getMethods()` (`getFields()`) is called at Label 6 (Label 3) instead, then an array of `Method` (`Field`) objects will be returned so that *Target*

Table 2. Comparing Elf with the two closely-related reflection analyses [4, 8].

Side-Effect Methods	Member-Introspecting Methods	[8]			Doop [4]		Elf			
		$\color{red}\rightarrow$	$\color{blue}\rightarrow$	$\color{red}\dashrightarrow$	$\color{red}\rightarrow$	$\color{blue}\rightarrow$	$\color{red}\rightarrow$	$\color{blue}\rightarrow$	$\color{red}\dashrightarrow$	$\color{blue}\dashrightarrow$
invoke	getMethod		√				√	√	√	√
	getDeclaredMethod		√		√		√	√	√	√
	getMethods		n/a			n/a	√	n/a	√	√
	getDeclaredMethods		n/a			n/a	√	n/a	√	√
get set	getField		√				√	√	√	√
	getDeclaredField		√		√		√	√	√	√
	getFields		n/a			n/a	√	n/a	√	√
	getDeclaredFields		n/a			n/a	√	n/a	√	√
newInstance		√	n/a	√	√	n/a	√	n/a	√	n/a

Propagation from them is implicitly performed. All the other methods available in `Class` for introspecting methods/fields/constructors are handled similarly.

3.3 Elf vs. Livshits et al.'s Analysis and Doop

Table 2 compares Elf with Livshits et al.'s and Doop's analyses [4, 8] in terms of how four representative side-effect reflective calls are resolved.

Target Propagation Elf resolves a target method/field at a reflective callsite by requiring both its target class type (red circle) and its target name (blue circle) to be known. However, this is not the case in the other two analyses. In the case of Livshits et al.'s analysis, the target class type is always ignored. Therefore, the target methods/fields with a given name in all the classes in the program are conservatively included. Doop suffers the opposite problem by ignoring the target method/field names. As a result, all methods/fields in the target class are included. Finally, of the three analyses, Elf is the only one that can handle all the member-introspecting methods listed.

Target Inference Of the three analyses, Elf is the only one to adopt a self-inferencing principle to find the target classes and methods/fields at a reflective callsite. Livshits et al.'s analysis narrows the type of reflectively-created objects at `newInstance()` in Figure 5, but Doop does not do this. However, Doop is more sophisticated than Livshits et al.'s analysis in distinguishing virtual, static and special calls and considering the modifiers of fields for loads and stores. These are all handled by the Elf reflection analysis.

4 Reflection Resolution

We specify the reflection resolution in Elf as a set of Datalog rules, i.e., monotonic logical inferences (with no negation in a recursion cycle), following the style of [6]. The main advantage is that the specification is close to the actual implementation. Datalog has been the basis of several pointer analysis tools [4, 6, 8, 21]. Our rules are declarative: the order of evaluation of rules or examination of their clauses do not affect the final results. Given a program to be analyzed, these rules are repeatedly applied to infer more facts until a fixed point is reached.

Elf works as part of a flow-insensitive Andersen's pointer analysis context-sensitively. However, all the Datalog rules are given here context-insensitively.

There are 207 Datalog rules. One set of rules handles all the 98 possible scenarios (i.e., combinations) involving the methods listed in Table 2 (illustrated in Figure 5), where $98 = 4$ (four member-introspecting methods) $\times$ 3 (three side-effect methods, `invoke()`, `get()` and `set()`) $\times$ 4 (four possible arrows in Figure 5) $\times$ 2 (two types of side-effect methods each, instance or static) + 2 (`newInstance()` with a statically known or unknown type). This set of rules is further divided into those for performing target propagation (involving $4\times3\times1\times2+1 = 25$ scenarios) and those for performing target inference. The remaining set of rules handles `Constructor` and arrays and performs bookkeeping duties.

Section 4.1 gives a set of domains and input/output relations used. Section 4.2 describes the seven target propagation scenarios corresponding to Labels $1-7$ in Figure 5. Section 4.3 describes four representative target inference scenarios. All the other rules (available as an open-source tool) can be understood analogously. Section 4.4 discusses briefly some properties about our analysis.

T: set of class types	V: set of program variables
M: set of methods	F: set of fields
H: set of heap abstractions	I: set of invocation sites
N: set of natural numbers	S: set of strings
Scall(*invo:I, mtd:M*)	Vcall(*invo:I, base:V, mtd:M*)
ActualArg(*invo:I, i:N, arg:V*)	ActualReturn(*invo:I, var:V*)
HeapType(*heap:H, type:T*)	Assignable(*toType:T, fromType:T*)
ThisVar(*mtd:M, this:V*)	LookUpMtd(*type:T, mName:H, dp:S, mtd:M*)
MtdString(*mtd:M, str:S*)	StringToClass(*strConst:H, type:T*)
MtdDecl(*type:T, mName:H, dp:S, mtd:M*)	FldDecl(*type:T, fName:H, fType:T, fld:F*)
PublicMtd(*type:T, mName:H, mtd:M*)	PublicFld(*type:T, fName:H, fld:F*)
NewInstanceHeap(*type:T, heap:H*)	Type-ClassHeap(*type:T, clzHeap:H*)
Mtd-MtdHeap(*mtd:M, mtdHeap:H*)	Fld-FldHeap(*fld:F, fldHeap:H*)
VarPointsTo(*var:V, heap:H*)	CallGraph(*invo:I, mtd:M*)
FldPointsTo(*base:H, fld:F, heap:H*)	RefCallGraph(*invo:I, mtd:M*)

Fig. 6. Domains and input/output relations.

4.1 Domains and Input/Output Relations

Figure 6 shows the eight domains used, 18 input relations and four output relations. Given a method *mtd* called at an invocation site I, as a static call (Scall) or a virtual call (Vcall), its i-th argument *arg* is identified by ActualArg and its returned value is assigned to *var* as identified by ActualReturn.

HeapType describes the types of heap objects. Assignable is the usual subtyping relation. ThisVar correlates *this* to each method where it is declared. MtdString specifies the signatures (in the form of strings) for all the methods, including also their containing class types and return types. StringToClass records the class type information for all compile-time string names. LookUpMtd matches a method *mtd* named *mName* with descriptor *dp* to its definition in a class, *type*. For simplicity, *mName* is modeled as a heap object in domain H rather than a string in S. We have done the same for method/field names in MtdDecl, FldDecl, PublicMtd and PublicFld.

MtdDecl records all methods and their declaring classes and FldDecl records all fields and their declaring classes. To find the metaobjects returned by `getMethod()` and `getField()`, PublicMtd matches a `public` target method m named *mName* in a class of type *type*, its superclasses or its interfaces searched in that order (as discussed in Section 2.1) and PublicFld does the same for fields except that *type*'s interfaces are searched before *type*'s superclasses.

The last four input relations record four different types of heap objects created. NewInstanceHeap relates the heap objects created at `newInstance()` calls with their class types. Type-ClassHeap, Mtd-MtdHeap and Fld-FldHeap relate all the classes, methods and fields in the (closed-world) program to their metaobjects (i.e., `Class`, `Method` and `Field` objects), respectively.

When working with a pointer analysis, Elf both uses and modifies the four output relations recording the results of the pointer analysis. VarPointsTo and FldPointsTo maintain the points-to relations and CallGraph encodes the call graph of the program. As in [4], RefCallGraph is used to record the potential callees resolved from a call to `invoke()`. The second argument of `invoke()` is an array containing the arguments of its target calls; special handling is needed to assign these arguments to the corresponding parameters of its target methods.

4.2 Target Propagation

We give seven target propagation scenarios corresponding to Labels 1 – 7 in Figure 5 when both a target method/field name and its target class type are known. These rules (used later in Section 4.3) are standard except for `getField()` and `getMethod()`. These two methods are ignored by Doop [4] but handled conservatively in [8], as shown in Table 2, with the target class of a target method/field ignored, causing the targets in all the classes in the program to be included.

The syntax of a rule is easy to understand: "$\leftarrow$" separates the inferred fact (i.e., the *head* of the rule) from the preciously established facts (i.e., the *body* of the rule). In Scenario P1, the rule for ForName says that among all static invocation sites, record the calls to `forName()` in the ForName relation. The rule for ResolvedClassType records the fact that all such invocation sites with constant names are resolved. Note that *const* is a heap object representing "string constant". Meanwhile, the points-to and call-graph relations are updated. For each resolved class, its static initialiser "<clinit>()", at the callsite is discovered in case the class has never been referenced in the program.

In Scenario P2, a `newInstance()` call is analyzed for each statically known class type pointed by *clz*. For such a type, a call to its default constructor "<init> ()" is noted. In Scenario P3 for handling a `getField()` call, both the statically known field and all the known target classes pointed by *clz*, i.e., *fldName* (a heap object representing "string constant") and *type* are considered. Similarly, a `getMethod()` call is handled in Scenario P6. Note that its second argument is ignored as discussed in Section 3.2. In Scenarios P4 and P5, calls to `get()` and `set()` are analyzed, respectively. Finally, in Scenario P7, an `invoke()`

Scenario P1: *Class clz = `Class.forName`("string constant");*

ForName(*invo*) ←
  Scall(*invo, mtd*), MtdString(*mtd*,
  "java.lang.Class: java.lang.Class forName(java.lang.String)").
ResolvedClassType(*invo, type*) ←
  ForName(*invo*), ActualArg(*invo*, 1, *arg*),
  VarPointsTo(*arg, const*), StringToClass(*const, type*).
CallGraph(*invo, clinit*), **VarPointsTo**(*clz, clzHeap*) ←
  ResolvedClassType(*invo, type*), Type-ClassHeap(*type, clzHeap*),
  MtdString(*clinit, type.toString()+".<clinit>()"*), ActualReturn(*invo, clz*).

Scenario P2: *Object obj = clz.`newInstance`();*

NewInstance(*invo, clz*) ←
  Vcall(*invo, clz, mtd*), MtdString(*mtd*, *"java.lang.Class: java.lang.Object newInstance()"*).
CallGraph(*invo, init*), **HeapType**(*heap, type*),
VarPointsTo(*this, heap*), **VarPointsTo**(*obj, heap*) ←
  NewInstance(*invo, clz*), VarPointsTo(*clz, clzHeap*), Type-ClassHeap(*type, clzHeap*),
  NewInstanceHeap(*type, heap*), MtdString(*init, type.toString()+".<init>()"*),
  ThisVar(*init, this*), ActualReturn(*invo, obj*).

Scenario P3: *Field f = clz.`getField`("string constant");*

GetField(*invo, clz*) ←
  Vcall(*invo, clz, mtd*), MtdString(*mtd*,
  "java.lang.Class: java.lang.reflect.Field getField(java.lang.String)").
ResolvedField(*invo, fld*) ←
  GetField(*invo, clz*), VarPointsTo(*clz, clzHeap*),
  Type-ClassHeap(*type, clzHeap*), ActualArg(*invo*, 1, *arg*),
  VarPointsTo(*arg, fldName*), PublicFld(*type, fldName, fld*).
VarPointsTo(*f, fldHeap*) ←
  ResolvedField(*invo, fld*), Fld-FldHeap(*fld, fldHeap*), ActualReturn(*invo, f*).

Scenario P4: *Object to = f.`get`(obj);*

Get(*invo, f*) ←
  Vcall(*invo, f, mtd*), MtdString(*mtd*,
  "java.lang.reflect.Field: java.lang.Object get(java.lang.Object)").
VarPointsTo(*to, valHeap*) ←
  Get(*invo, f*), VarPointsTo(*f, fldHeap*), Fld-FldHeap(*fld, fldHeap*),
  ActualArg(*invo, 1, obj*), VarPointsTo(*obj, baseHeap*),
  FldPointsTo(*baseHeap, fld, valHeap*), ActualReturn(*invo, to*).

call is handled, identically as in Doop [4] but differently from [8], which approximates its target methods by disregarding the target object *obj*, on which the target methods are called.

4.3 Target Inference

When a target method/field name or a target class type is unknown, Elf will infer the missing information, symbolized by red and blue circles along the dashed arrows in Figure 5. Below we give the Datalog rules for four representative scenarios (out of a total of 73 scenarios mentioned earlier for target inference).

Scenario I1: *Class clz1 = `Class.forName`(?); A a = (A) clz2.`newInstance`().*
The post-dominating cast *(A)* is used to infer the target class types of the objects reflectively created and pointed to by *a*, where *clz2* points to a `Class` object of an unknown type that is initially pointed to by *clz1*.

Scenario P5: *f.*`set`*(obj, val);*

SET(*invo, f*) ←
  VCALL(*invo, f, mtd*), MTDSTRING(*mtd,*
  "java.lang.reflect.Field: void set(java.lang.Object, java.lang.Object)").
FLDPOINTSTO(*baseHeap, fld, valHeap*) ←
  SET(*invo, f*), VARPOINTSTO(*f, fldHeap*), FLD-FLDHEAP(*fld, fldHeap*),
  ACTUALARG(*invo, 1, obj*), VARPOINTSTO(*obj, baseHeap*),
  ACTUALARG(*invo, 2, val*), VARPOINTSTO(*val, valHeap*).

Scenario P6: *Method m = clz.*`getMethod`*("string const", {...});*

GETMETHOD(*invo, clz*) ←
  VCALL(*invo, clz, mtd*), MTDSTRING(*mtd,*
  "java.lang.Class: java.lang.reflect.Method getMethod(java.lang.String, java.lang.Class[])").
RESOLVEDMETHOD(*invo, mtd*) ←
  GETMETHOD(*invo, clz*), VARPOINTSTO(*clz, clzHeap*),
  TYPE-CLASSHEAP(*type, clzHeap*), ACTUALARG(*invo*, 1, *arg*),
  VARPOINTSTO(*arg, mtdName*), PUBLICMTD(*type, mtdName, mtd*).
VARPOINTSTO(*m, mtdHeap*) ←
  RESOLVEDMETHOD(*invo, mtd*), MTD-MTDHEAP(*mtd, mtdHeap*), ACTUALRETURN(*invo, m*).

Scenario P7: *Object to = m.*`invoke`*(obj, {...});*

INVOKE(*invo, m*) ←
  VCALL(*invo, m, mtd*), MTDSIGSTRING(*mtd, "java.lang.reflect.Method:*
  java.lang.Object invoke(java.lang.Object, java.lang.Object[])").
REFCALLGRAPH(*invo, virtualMtd*), **VARPOINTSTO**(*this, heap*) ←
  INVOKE(*invo, m*), VARPOINTSTO(*m, mtdHeap*), MTD-MTDHEAP(*mtd, mtdHeap*),
  ACTUALARG(*invo, 1, obj*), VARPOINTSTO(*obj, heap*), HEAPTYPE(*heap, type*),
  MTDDECL(_ *, mtdName, mtdDescriptor, mtd*), THISVAR(*virtualMtd, this*),
  LOOKUPMETHOD(*type, mtdName, mtdDescriptor, virtualMtd*).

Scenario I2: *Field[] fs1=clz.*`getDeclaredFields`*();f2=fs2[i]; a=(A)f1.*`get`*(obj)*. The post-dominating type *(A)* is used to infer the target fields reflectively accessed at `get`*()* on the `Field` objects that are initially stored into *fs1* and later pointed to by *f1*. Note that *clz* is known in this case.

Scenario I3: *Field[] fs1=clz.*`getDeclaredFields`*(); f2 = fs2[i]; f1.*`set`*(obj, val)*. The dynamic types of *val* are used to infer the target fields modified.

Scenario I4: *Method m1=clz.*`getMethod`*(?, params); a=m2.*`invoke`*(obj, args)*. The dynamic types of *args* will be used to infer the target methods called on the `Method` objects that are pointed to by *m2* but initially created at a call to *m1=clz.*`getMethod`*()*, where *clz* is known.

Figure 7 gives a few new relations used for handling these four scenarios. The first three are used to identify metaobjects with non-constant names (called *placeholder objects*). CLASSPH identifies all the invocation sites, e.g., `Class.forName(?)`, where `Class` objects with unknown class names are created. MEMBERPH identifies the invocation sites, e.g., calls to *clz.*`getMethod(?, ...)` (*clz.*`getField(?)`), where `Method` (`Field`) objects are created to represent unknown method (field) names '?' in a known class *clz* of type *type*. If *clz* is also unknown, a different relation (not used here) is called for. Furthermore, MEMBERPHARRAY identifies which placeholder objects represent arrays. For example, a call to *clz.*`getDeclaredFields()` returns an array of `Field` objects.

ClassPh(*invo:I, heap:H*)	MemberPh(*invo:I, type:T, heap:H*)
MemberPhArray(*invo:I, array:H*)	NewInstanceCast(*invo:I, castType:T*)
GetCast(*invo:I, castType:T*)	HierarchyType(*castType:T, type:T*)
ArrayPointsTo(*arr:H, heap:H*)	

Fig. 7. Input and output relations for handling target inference.

We leverage the type cast information in target inference. The NewInstanceCast and GetCast relations correlate each downcast with their post-dominated invocation sites `newInstance()` and `get()`, respectively. HierarchyType(*type*, *castType*) records all the types such that either Assignable(*castType*, *type*) or Assignable(*type*, *castType*) holds. Finally, the output relation ArrayPointsTo records the heap objects stored in an array heap object *arr*.

Below we describe the target inference rules for the four scenarios above. Note that once a missing target name or a target class or both are inferred, some target propagation rules that could not be applied earlier may be fired.

Scenario I1: Class clz1 = `Class.forName`(?); A a = (A) clz2.`newInstance`(). If the string argument *strHeap* marked by '?' in `Class.forName(?)` is not constant (i.e., if StringToClass does not hold), then *clz1* points to a placeholder object *phHeap*, indicating a `Class` object of an unknown type. Such pointer information is computed together with the pointer analysis used. If *clz2* points to a placeholder object, then *a* can be inferred to have a type *type* that is assignable to the post-dominating cast *castType*, i.e., *A*. As *type* may not be initialized elsewhere, a call to its "<clinit>()" is conservatively assumed. After this, the second rule in Scenario P2 can be applied to the *clz2.`newInstance()`* call.

Scenario I1: *Class clz1 = `Class.forName(?)`; A a = (A) clz2.`newInstance()`;*

VarPointsTo(*clz1, phHeap*) ←
  ForName(*invo*), ActualArg(*invo, 1, arg*), VarPointsTo(*arg, strHeap*),
  ¬StringToClass(*strHeap,* _), ClassPh(*invo, phHeap*), ActualReturn(*invo, clz1*).
CallGraph(*invo, clinit*), **VarPointsTo**(*clz2, clzHeap*) ←
  NewInstance(*invo, clz2*), VarPointsTo(*clz2, phHeap*), ClassPh(_ *, phHeap*),
  NewInstanceCast(*invo, castType*), Assignable(*castType, type*),
  Type-ClassHeap(*type, clzHeap*), MtdString(*clinit, type.toString()+".<clinit>()"*).

Unlike [8], Elf does not use the cast *(A)* to further constrain the `Class` objects that are created for *clz1* and later passed to *clz2*, because the cast operation may not necessarily post-dominate the corresponding `forName()` call.

Scenario I2: Field[] fs1=clz.`getDeclaredFields`(); f2=fs2[i]; a=(A) f1.`get`(obj). Let us first consider a real case in Figure 8. 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`. By using the cast information, 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. Otherwise, all the static fields in `URLConnection` must be assumed. The reason why both the supertypes and subtypes must be considered was explained in Section 3.2. These type relations are captured by HierarchyType.

```
Application:Eclipse(v4.2.2):
Class:org.eclipse.osgi.framework.internal.core.Framework
public static Field getField(Class clazz, ...) {
  Field[] fields = clazz.getDeclaredFields(); ...
  for(int i=0; i<fields.length; i++) { ...
    return fields[i]; }}
private static void forceContentHandlerFactory(...) {
  Field factoryField = getField(URLConnection.class, ...);
  java.net.ContentHandlerFactory factory =
       (java.net.ContentHandlerFactory) factoryField.get(null);...}
```

Fig. 8. Target field inference based on the type cast at `get()`.

The same code pattern in Figure 8 also appears in five other places in `Eclipse4`. The prior analyses [4, 8, 20] cannot resolve the call `get()` above since `getDeclaredFields()` is ignored. Elf has succeeded in deducing that only two out of a total of 13 static fields in `URLConnection` are accessed at the callsite.

Scenario I2: *Field[] fs1 = clz.**getDeclaredFields**(); f2 = fs2[i]; a = (A) f1.**get**(obj);*

GetDeclaredFields(*invo, clz*) ←
  VCall(*invo, clz, mtd*), MtdString(*mtd,*
  "java.lang.Class: java.lang.reflect.Field[] getDeclaredFields()").
ArrayPointsTo(*phArray, phHeap*), **VarPointsTo**(*fs1, phArray*) ←
  GetDeclaredFields(*invo, clz*), VarPointsTo(*clz, clzHeap*), Type-ClassType(*type, clzHeap*)
  MemberPhArray(*invo, phArray*), MemberPh(*invo, type, phHeap*), ActualReturn(*invo, fs1*).
VarPointsTo(*f1, fldHeap*) ←
  Get(*invo, f1*), VarPointsTo(*f1, phHeap*),
  MemberPh(*getDecInvo, type, phHeap*), GetDeclaredFields(*getDecInvo,* _),
  GetCast(*invo, castType*), HierarchyType(*castType, fldType*),
  FldDecl(*type,* _*, fldType, fld*), Fld-FldHeap(*fld, fldHeap*).

It is now easy to understand Scenario I2. The second rule processes each call to `getDeclaredFields()`. For each class *clz* of a known type, *type*, *fs1* is made to point to *phArray* (a placeholder representing an array), which points to *phHeap* (a placeholder representing implicitly all the fields obtained in the call to `getDeclaredFields()`). When *f2 = fs2[i]* is analyzed by the pointer analysis engine, *f1* will point to whatever *fs1* contains if the values of *fs1* flow into *fs2* and the values of *f2* flow into *f1*. The last rule leverages the type cast information to resolve *f1* at a `get()` call to its potential target `Field` objects, *fldHeap*. As a result, the second rule in Scenario P4 has now been enabled.

Scenario I3: Field[] fs1=clz.`getDeclaredFields`(); f2=fs2[i]; f1.`set`(obj, val). This is similar to Scenario I2, except that the dynamic types of *val* (e.g., the dynamic type of `value` in line 290 in Figure 9 is `java.lang.String`) are used to infer the target fields modified. Thus, the second rule in Scenario P5 is enabled.

```
Application:Eclipse(v4.2.2):
Class:org.eclipse.osgi.util.NLS
300 static void load(final String bundleName, Class<?> clazz) {
302   final Field[] fieldArray = clazz.getDeclaredFields();
336   computeMissingMessages(..., fieldArray, ...);...}
267 private static void computeMissingMessages(..., Field[] fieldArray, ...) {
272   for (int i = 0; i < numFields; i++) {
273     Field field = fieldArray[i];
284     String value = "NLS missing message: " + ...;
290     field.set(null, value);...}}
```

Fig. 9. Target field inference based on the dynamic type of `value` in `set()`.

Note that the `set()` call that appears in line 290 in Figure 9 cannot be handled by the prior analyses [4, 8, 20] since `getDeclaredFields()` is ignored. This code pattern appears one more time in line 432 in the same class, i.e., `org.eclipse.osgi.util.NLS`. These two `set()` calls are used to initialize all non-final static fields in four classes (by writing a total of 276 fields each time). Based on target inference, Elf has found all the target fields accessed precisely.

Scenario I3: *Field[] fs1 = clz.*`getDeclaredFields`*(); f2 = fs2[i]; f1.*`set`*(obj, val);*

VarPointsTo(*f1, fldHeap*) ←
  Set(*invo, f1*), VarPointsTo(*f1, phHeap*), MemberPh(*getDecInvo, clzType, phHeap*),
  GetDeclaredFields(*getDecInvo, _*), ActualArg(*invo, 2, val*), VarPointsTo(*val, valHeap*),
  HeapType(*valHeap, type*), Assignable(*fldType, type*),
  FldDecl(*clzType, _, fldType, fld*), Fld-FldHeap(*fld, fldHeap*).

Scenario I4: Method m1=clz.`getMethod`(?, params); a=m2.`invoke`(obj, args). Let us consider a real case from `Eclipse4` in Figure 10. In line 174, the `Class` objects on which `getMethod()` is invoked can be deduced from the types of the objects pointed to by `target` but `cmd` is read from input. Thus, in line 174, `method` is unknown even though its target class is known. Note that `parameters` is explicitly initialized to `{this}` in line 155. As the type `FrameworkCommandInterpreter` has not subtypes, we conclude that the corresponding parameter of each potential target method must have this type or one of its supertypes.

```
Application:Eclipse(v4.2.2):
Class:org.eclipse.osgi.framework.internal.core.FrameworkCommandInterpreter
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); ...}}
```

Fig. 10. Target inference based on the dynamic types of `parameters` in `invoke()`.

As explained in Section 3.2, we have relied on an intraprocedural analysis to perform the inference when *args* can be analyzed exactly element-wise as is the case in Figure 10. The MatchArgs(*args, mtd*) relation over $V \times M$ maintains target methods *mtd* found from *args* this way.

Scenario I4: *Method m1 = clz.*`getMethod`*(?, params); a = m2.*`invoke`*(obj, args);*

VarPointsTo(*m1, phHeap*) ←
  GetMethod(*getInvo, clz*), ActualArg(*getInvo, 1, arg*), VarPointsTo(*arg, strHeap*),
  ¬MtdDecl(*_, strHeap, _, _*), VarPointsTo(*clz, clzHeap*), Type-ClassHeap(*type, clzHeap*),
  MemberPh(*getInvo, type, phHeap*), ActualReturn(*getInvo, m1*).
VarPointsTo(*m2, mtdHeap*) ←
  Invoke(*invo, m2*), VarPointsTo(*m2, phHeap*), MemberPh(*getInvo, type, phHeap*),
  GetMethod(*getInvo, _*), PublicMtd(*type, _, mtd*), ActualArg(*invo, 2, args*),
  MatchArgs(*args, mtd*), Mtd-MtdHeap(*mtd, mtdHeap*).

Let us now look at the rules given in Scenario I4 where *clz* points to statically known class, *type*, but the target methods at `invoke()` are unknown, just like the the case illustrated in Figure 10. In the first rule applied to `getMethod()`, MthDecl(*_, strHeap, _, _*) does not hold, since *strHeap* is not a constant. As a result, *m1* points to a placeholder `Method` object (indicating that its method name is unknown). In the second rule, if $m2$ at the `invoke()` callsite points to a placeholder object, PublicMtd will be used to find all the target methods from the class *type* based on the ones inferred from *args* and stored in MatchArgs.

Once the `Method` objects at an `invoke` callsite are resolved, the second rule in Scenario P7 can be applied to resolve the target methods.

Note that the `invoke()` call in Figure 10 cannot be resolved by the prior analyses [4, 8] since `getMethod()` is either ignored [4] or cannot be handled due to unknown method name [8]. Based on target inference, Elf has found 50 target methods at this callsite, out of which 48 are real targets by manual inspection.

4.4 Properties

Like the prior reflection analyses [4, 8, 20], Elf is unsound. Firstly, Elf ignores the part of the Java reflection API related to dynamic class loading. Second, Elf infers a target at a reflective callsite if and only if both its target name and its target class are known to strike a good tradeoff between soundness and precision. However, Elf's rules can soundly analyze a reflective callsite if all its targets are known (by its target propagation) or inferred (by its target inference). These properties follow directly from the Datalog rules formulated in this section.

5 Evaluation

The goal of this research is to produce an open-source reflection analysis to improve the effectiveness of modern pointer analysis tools for Java applications. We evaluate Elf against a state-of-the-art reflection analysis implemented in Doop [4]. Being unsound, both analyses make different tradeoffs among soundness, precision and scalability. Our evaluation has validated the following hypotheses about our self-inferencing approach in handling reflective code.

Soundness and Precision Tradeoffs Elf can usually resolve more reflective call targets than Doop while avoiding many of its spurious targets.

Target Propagation vs. Target Inference Elf can resolve more reflective call targets when target propagation fails, by inferring the missing target information with target inference. This can be particularly effective for some reflection idioms used in practice (as highlighted in Figures 8 – 10).

Effectiveness When used as part of an under-approximate pointer analysis, Elf is effective measured in terms of a few popular metrics used.

Scalability Compared to Doop, Elf achieves the above results at small analysis time increases for a set of Java programs evaluated.

5.1 Implementation

We have implemented Elf with context sensitivity in Doop (r160113) [4], a modern pointer analysis tool for Java. On top of Doop's 64 Datalog rules for reflection handling, we have added 207 rules. Elf is comprehensive in handling the Java reflection API, by tackling significantly more methods than prior work

[4, 8, 9, 20]. Specifically, Elf handles the first eight side-effect methods listed in Table 1, all *member-introspecting* methods in the reflection API, and four out of the six *entry* methods, `forName()`, `getClass()`, `getComponentType()` and `.class`, shown in Figure 3. For the three side-effect methods on `Array`, `Array::newInstance` is handled similarly as `Class::newInstance`. We have ignored `Proxy::newProxyInstance(...)` in Table 1 and `loadClass()` and `getProxyClass()` in Figure 3 due to the closed-world assumption (Section 3.1).

We have modified the fact generator in Doop by using an intraprocedural post-dominance algorithm in Soot [19] to generate the post-dominance facts, e.g., NewInstanceCast and GetCast in Figure 7 (and InvokeCast not given).

5.2 Experimental Setup

Our setting uses the LogicBlox Datalog engine (v3.9.0), on a Xeon E5-2650 2GHz machine with 64GB of RAM. We use all the 11 DaCapo benchmarks (v.2006-10-MR2) and two real-world applications from our reflection-usage study, `Eclipse-4.2.2` and `javac-1.7.0`. We have excluded `Tomcat`, `Jetty` and `jEdit`, since neither Doop nor Elf handles the custom class loaders used in the first two applications and neither can terminate in three hours for the last one. We have used recent (large) standard libraries: `JDK 1.7.0_25` for `Eclipse v4.2.2` and `javac v1.7.0` and `JDK 1.6.0_45` for the remaining programs. For the *fop* benchmark from DaCapo, we added `org.w3c.dom` and `org.w3c.css` to enable it to be analyzed. Since `java.util.CurrencyData` is only used reflectively, we have made it available in the class path of the fact generator to make it analyzable.

We compare Elf with Doop's reflection analysis, when both are performed in the Doop's pointer analysis framework. Both analyses for a program are performed in the SSA form of the program generated by Soot, under 1-callsite context sensitivity implemented in Doop. An array is treated as a whole.

5.3 Results and Analysis

For each program analyzed, the results presented are obtained from all the analyzed code, in both the application itself and the libraries used.

5.3.1 Soundness and Precision Tradeoffs Elf and Doop are unsound in different ways. So either reflection analysis, when working with the same pointer analysis, may resolve some *true* targets that are missed by the other, in general. Elf handles a significant part of the Java reflection API that is ignored by Doop (Table 2). To eliminate the impact of this aspect of Elf on its analysis results, we have designed a configuration of Elf, called Elf^d, that is restricted to the part of the reflection API handled by Doop. These include three entry methods, `forName()`, `getClass()` and `.class`, two member-introspecting methods, `getDeclaredMethod()` and `getDeclaredField()`, as well as four side-effect methods, `invoke()`, `set()`, `get()` and `newInstance()` without using the cast inference. Elf^d behaves identically as Doop except for the following three differences. First, Elf^d applies target propagation since this is more precise than

DOOP's analysis in cases when both target method/field names and their target class names are known. Second, ELF^d uses target inference wherever target propagation fails. Finally, ELF^d handles m=*clz*.`getDeclaredMethod(mName, ...)` (m=*clz*.`getDeclaredField(fName)`) identically as DOOP for each known `Class` object C pointed to by *clz* only when `mName` (`fName`) points to a target name that cannot be resolved by either target propagation or target inference. In this case, m is resolved to be the set of all declared targets in the target class C.

There are two caveats. First, a call to `getDeclaredMethod("str-const")` or `getDeclaredField("str-const")` is ignored if `str-const` is absent in the closed-world. Second, in its current release (r160113), DOOP resolves `mtd.invoke(o,args)` to calls to potential target methods unsoundly by using B from the dynamic types $B[]$ of the array objects *obj* pointed by `args` to help filter out many objects passed from `args` to the corresponding parameters in the target methods.[1] We have modified two rules, LOADHEAPARRAYINDEX in `reflective.logic` and VARPOINTSTO in `context-sensitive.logic`, to make this handling sound by using the dynamic types of the objects pointed to by *obj* instead. Both ELF^d and DOOP handle all such interprocedural assignments exactly this way.

Table 3 compares ELF^d and DOOP in terms of their soundness and precision tradeoffs made when resolving `invoke()`, `get()` and `set()` calls. Both analyses happen to resolve the same number of reflective callsites. For a program, ELF^d usually discovers the same target methods/fields while avoiding many spurious ones introduced by DOOP. We have carried out a recall experiment for all the 11 DaCapo benchmarks by using Tamiflex [2] under its three inputs (small, default and large). We have excluded `Eclipse4` and `Javac` since the former cannot be analyzed by Tamiflex and the latter has no standard inputs. We found that the set of true targets resolved by ELF^d is always the same as the set of true targets resolved by DOOP for all the benchmarks except `jython` (analyzed below).

In `jython`, there is a call `m=clz.getDeclaredMethod("typeSetup", ...)` in method `PyType::addFromClass()`, where `clz` points to a spurious `Class` object representing the class `__builtin__` during the analysis. ELF^d ignores `__builtin__` since `typeSetup` is not one of its members. However, DOOP resolves `m` to be any of the declared methods in the class, including `classDictInit()`, opportunistically. As a result, a spurious call edge to `__builtin__::classDictInit()` is added from an `invoke()` site in `PyType::fillFromClass()`. However, this target method turns out to be called from the (only) `invoke` site contained in `PyJavaClass ::initialize()` on a `Method` object created at the (only) `getMethod` call, which is also contained in `initialize()`. By analyzing this target method, DOOP eventually resolves five true target methods named `typeSetup` at `m=clz.getDeclaredMethod ("typeSetup", ...)` and seven true target fields at `clz.getDeclareField ("exposed_" + name).get(null)` in `PyType::exposed_decl_get_object()`. These 12 targets are missed by ELF^d.

In ELF^d, the primary contributor for ELF's precision improvement (over DOOP) is its target propagation component. It is significantly more beneficial

[1] DOOP has recently fixed this unsound handling in its latest beta version (r5459247), which also includes analyzing some reflective calls not handled in Table 2.

Table 3. Comparing Elf^d and Doop on reflection resolution. According to this particular configuration of Elf, C denotes the same number of resolved side-effect callsites in both analyses and T denotes the number of target methods/fields resolved by either.

			antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	eclipse4	javac
invoke	C		2	2	5	2	5	-	3	2	2	2	2	6	0
	T	Doop	77	77	1523	77	1730	-	897	77	77	77	77	78	0
		Elf^d	3	3	11	3	11	-	15	3	3	3	3	8	0
set	C		0	0	0	0	0	-	0	0	0	0	0	2	0
	T	Doop	0	0	0	0	0	-	0	0	0	0	0	31	0
		Elf^d	0	0	0	0	0	-	0	0	0	0	0	2	0
get	C		9	9	9	9	9	-	10	9	9	9	9	2	2
	T	Doop	194	194	194	194	194	-	1292	194	194	194	194	132	3401
		Elf^d	28	28	28	28	28	-	1094	28	28	28	28	21	23

to track both constant class names and constant method/field names simultaneously rather than either alone, as suggested earlier in Figure 4.

5.3.2 Target Propagation vs. Target Inference To evaluate their individual contributions to the soundness and precision tradeoff made, we have included a version of Elf, named Elf^p, in which only target propagation is used. Table 4 is an analogue of Table 3 except that Elf and Elf^p are compared. By examining their results for a side-effect method across the 13 programs, we find that both component analyses have their respective roles to play. For most programs, Elf has added zero or a moderate number of additional targets on top of Elf^p. This has two implications. First, target propagation can be quite effective for some programs if they exhibit many constant class/method/field names (Figure 4). Second, target inference does not introduce many spurious targets since Elf resolves a reflective target only when both its name and its target class are known (symbolized by the simultaneous presence of two circles in Figure 5).

From the same recall experiment described earlier, Elf is found to resolve no fewer true targets across the 11 DaCapo benchmarks except `jython` than Doop. In `jython`, Elf has resolved all the true target methods resolved by Doop by analyzing all member-introspecting methods. In the case of this afore-mentioned call to `clz.getDeclareField("exposed_" + name).get(null)`, Elf fails to discover any target fields due to the absence of cast information. In contrast, Doop has resolved 1098 target fields declared in all `Class` objects pointed to by `clz`, with only 22 sharing `exposed_` as the prefix in their names. In our recall experiment, 21 of these 22 targets are accessed. Elf can be easily generalized to infer the target fields accessed (the blue circle shown in Figure 5) at this `get()` callsite in a disciplined manner. By also exploiting the partially known information about target names (such as the common prefix `exposed_`), Elf will only need to resolve the 22 target names starting with `exposed_` at this callsite.

Target inference can often succeed where target propagation fails, by resolving more reflective targets at some programs. Let us consider `Eclipse4`. The situation for `Eclipse` in DaCapo is similar. In `Eclipse4`, there are two `set()` callsites with their usage pattern illustrated in Figure 9. Elf^p discovers one target from each callsite. However, Elf has discovered 553 more, one from one of the two callsites and 552 true targets at the two callsites as discussed in Sec-

Table 4. Comparing Elf and Elf^p, where C and T are as defined in Table 3.

			antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	eclipse4	javac
invoke	Elf^p	C	2	2	9	5	9	6	7	2	2	2	15	15	4
		T	3	3	30	20	30	53	58	3	3	3	31	91	25
	Elf	C	2	2	10	8	10	8	7	2	2	2	16	26	4
		T	3	3	37	94	37	228	58	3	3	3	36	227	25
set	Elf^p	C	0	0	0	0	0	0	0	0	0	0	0	2	0
		T	0	0	0	0	0	0	0	0	0	0	0	2	0
	Elf	C	0	0	0	2	0	0	0	0	0	0	0	4	0
		T	0	0	0	580	0	0	0	0	0	0	0	555	0
get	Elf^p	C	9	9	9	9	9	11	9	9	9	9	9	2	2
		T	28	28	28	28	28	32	28	28	28	28	28	21	23
	Elf	C	9	9	9	9	9	11	11	9	9	9	9	8	2
		T	28	28	28	28	28	41	34	28	28	28	28	35	23

tion 4.3. As for `get()`, Elf has found 14 more targets than Elf^p, with 12 true targets found from the six code fragments (with their usage pattern given in Figure 8), contributing two each, as explained in Section 4.3. Finally, there are two `invoke()` callsites similar to the one illustrated in Figure 10. Elf has discovered a total of $2 \times 48 = 96$ true target methods invoked at the two callsites. How to resolve one such `invoke()` call is also discussed in Section 4.3.

When analyzing Java programs, a reflection analysis works together with a pointer analysis. Each benefits from precision improvements from the other. If the pointer analysis used from Doop is 2-callsite-sensitive+heap, then $C = 5$ and $T = 22$ for Elf^p and $C = 8$ and $T = 83$ for Elf for `hsqldb` in Table 4.

5.3.3 Effectiveness Table 5 shows the effectiveness of Elf when it is used in an under-approximate pointer analysis, which is usually regarded as being sound in the literature. In addition to Doop, Doop^b is its baseline version with reflection ignored except that only calls to `newInstance()` are analyzed (precisely). As in [6], the same five precision metrics are used, including two clients, poly v-calls and may-fail casts (smaller is better). Elf distinguishes different constant class/method/field names. As mentioned in an afore-mentioned caveat, Doop has been modified to behave identically. However, Doop^b distinguishes only different constant class names as it ignores the first `String` parameter in calls to `getDeclaredMethod()` or `getDeclaredField()`. As a result, Doop^b represents all other string constants (the ones which do not represent class names) with a single string object. To ensure a fair comparison (and follow [6, 15]), we have post-processed the analysis results from both Doop and Elf using the same string abstraction as in Doop^b. As Doop does not exploit the type cast for `newInstance()`, Elf does not do it either. In addition, Elf's capability for handling reflective code on `Array` is turned off as Doop ignores it.

As all the three analyses are unsound, the results in Table 5 must be interpreted with caution. Having compared Elf^d and Doop earlier, we expect these results to provide a rough indication about the effectiveness of Elf (relative to Doop) in reflection resolution. Despite the fact that Elf usually resolves more true targets as explained earlier (Tables 3 and 4), Elf exhibits smaller numbers in eight programs in terms of all the five metrics and slightly larger ones in the

Table 5. Comparing Elf and Doop in terms of five pointer analysis precision metrics (*smaller is better*): the average size of points-to sets, the number of edges in the computed call-graph (including regular and reflective call graph edges), the number of virtual calls whose targets cannot be disambiguated, the number of casts that cannot be statically shown safe, and the total points-to set size. The benchmarks for which Elf produced larger numbers than Doop are highlighted in **bold**.

		average objects per var	call graph edges ∼ reachable methods	poly v-calls / reachable v-calls	may-fail casts / reachable casts	size of var points-to (M)
antlr	Doop^b	29.26	61107∼8.9K	2000/33K	1040/1.8K	16.1
	Doop	29.43	61701∼9.1K	2002/33K	1060/1.8K	16.3
	Elf	29.02	61521∼9.0K	2001/33K	1051/1.8K	16.1
bloat	Doop^b	42.36	70661∼10.1K	2144/31K	1998/2.8K	32.7
	Doop	42.29	71202∼10.3K	2146/31K	2016/2.8K	32.9
	Elf	42.01	71075∼10.3K	2145/31K	2009/2.8K	32.7
chart	Doop^b	43.06	82148∼15.7K	2820/39K	2414/3.7K	47
	Doop	43.55	85878∼16.3K	2928/40K	2534/3.9K	48.8
	Elf	42.99	83872∼16.1K	2845/40K	2454/3.8K	48.1
eclipse	Doop^b	21.11	53738∼9.4K	1520/23K	1149/2.0K	12.3
	Doop	21.31	54357∼9.6K	1521/23K	1169/2.0K	12.5
	Elf	**21.41**	**55885∼9.9K**	**1582/25K**	**1297/2.2K**	**12.8**
fop	Doop^b	36.72	77052∼15.4K	2751/34K	2082/3.3K	39.7
	Doop	37.3	80958∼16.1K	2871/35K	2177/3.5K	41.5
	Elf	36.7	78758∼15.8K	2775/35K	2119/3.4K	40.7
hsqldb	Doop^b	23.79	73950∼13.2K	1888/36K	1765/2.8K	17.8
	Doop	—	—	—	—	—
	Elf	34.4	78290∼13.7K	1939/37K	1825/2.9K	27.5
jython	Doop^b	28.31	57127∼9.8K	1652/24K	1305/2.2K	17.4
	Doop	107.29	96200∼13.4K	2534/29K	2252/3.2K	89
	Elf	**112.09**	93503∼12.9K	2478/28K	**2291/3.2K**	88.5
luindex	Doop^b	16.65	42130∼7.9K	1189/18K	829/1.5K	7.7
	Doop	16.92	42724∼8.1K	1191/18K	849/1.5K	7.8
	Elf	16.52	42544∼8.0K	1190/18K	840/1.5K	7.7
lusearch	Doop^b	17.57	45399∼8.5K	1368/19K	930/1.6K	8.6
	Doop	17.82	45992∼8.7K	1370/20K	950/1.7K	8.7
	Elf	17.43	45812∼8.7K	1369/19K	941/1.6K	8.6
pmd	Doop^b	18.9	49230∼9.3K	1258/21K	1265/2.0K	11.2
	Doop	19.12	49825∼9.5K	1260/21K	1285/2.0K	11.4
	Elf	18.76	49644∼9.5K	1259/21K	1276/2.0K	11.2
xalan	Doop^b	25.84	58356∼10.6K	1977/26K	1202/2.1K	15.5
	Doop	25.95	58896∼10.8K	1979/26K	1220/2.1K	15.7
	Elf	**27.25**	**60260∼10.9K**	**2085/26K**	**1263/2.1K**	**16.7**
eclipse4	Doop^b	30.48	57141∼10.1K	1634/25K	1223/2.2K	20.3
	Doop	30.4	58060∼10.4K	1671/25K	1335/2.3K	20.4
	Elf	**33.01**	**61129∼10.8K**	**1733/27K**	**1410/2.4K**	**23.1**
javac	Doop^b	48.99	84084∼13.1K	4102/35K	2925/4.0K	43.6
	Doop	54.62	84425∼13.3K	4103/36K	2930/4.0K	45
	Elf	**55.56**	**84747∼13.4K**	**4105/36K**	**2934/4.0K**	**47.9**

remaining five programs (highlighted in bold font). Thus, these results suggest that Elf appears to strike a good tradeoff between soundness and precision.

For `jython`, both Doop and Elf have significantly increased the code coverage of the underlying pointer analysis used. For the `invoke()` site in `PyType::fillFromClass()`, both Doop and Elf have resolved 17 methods named `typeSetup` residing in 17 classes, with five being resolved differently as explained earlier. When each of these methods is executed (during our recall experiment), 1 to 47 inner classes are exercised. So this benchmark demonstrates once again the importance of reflection analysis, in practice.

Table 6. Comparing ELF and DOOP in term of analysis times (secs).

	antlr	bloat	chart	eclipse	fop	hsqldb	jython	luindex	lusearch	pmd	xalan	eclipse4	javac
DOOP	171	299	503	151	442	-	730	103	112	167	215	262	563
ELF	211	309	538	193	804	475	3561	115	122	550	733	445	755

5.3.4 Scalability Table 6 compares ELF with DOOP in terms of analysis time consumed. In the case of `hsqldb`, DOOP cannot run to completion in three hours. In prior work [6, 15], `jython` and `hsqldb` are often analyzed with reflection disabled and `hsqldb` has its entry point set manually in a special harness. Note that if only target method/field names are tracked as described in [8, 9], the resulting version of ELF cannot terminate in three hours for these two benchmarks. As ELF handles more reflection methods than DOOP, by performing target propagation as well as more elaborate and more time-consuming target inference, ELF exhibits a slowdown of 1.9X on average with `hsqldb` disregarded.

6 Related Work

Static Analysis In Section 3.3, we have compared ELF in great detail with the two most-closely related static analyses [4, 8]. Briefly, Livshits et al. [8] introduced the first static reflection analysis for Java, which has influenced the design and implementation of several pointer analysis tools [4, 20, 21]. They suggested tracking the flow of string constants and leveraging the cast information to narrow the types of objects created at `newInstance()`, and implemented their analysis in bddbddb [21], a tool for specifying and querying program analyses. However, ELF is the first to leverage the cast information to resolve targets at other reflective calls, such as `invoke()`, `get()` and `set()`.

DOOP [4] includes a few pointer analyses for Java programs using the Datalog language. Its reflection handling can be seen as analogous to adding a sophisticated analysis similar as [8] but in conjunction with a context-sensitive pointer analysis. In addition, DOOP considers more Java features (such as distinguishing instance from static field operations) when handling reflection.

Wala [20] is a tool from IBM Research designed for static analysis. Its reflection handling is similar to DOOP's (i.e., by considering only class types to resolve reflective calls), but without handling `Field`-related methods.

In summary, existing solutions focus on target propagation by tracking the flow of string constants representing either method/field names [8, 21] or class names [4, 20] in a program. ELF takes a disciplined approach to balance soundness, precision and scalability by exploiting a self-inferencing property inherent in reflective code. As illustrated in Figure 5, ELF resolves a reflective target when both its target class (red circle) and its target method/field name (blue circle) are known, by performing target propagation (through tracking string constants) and target inference (through type inference). In future work, we will improve ELF to infer missing target method/field names based on some partial information obtained from string manipulation operations and to handle the situations when either a target method/field name or a target class type is missing.

Dynamic Analysis Hirzel et al. [5] proposed an online pointer analysis for handling various dynamic features of Java at run time. To tackle reflection, their analysis instruments a program so that constraints are generated dynamically when the injected code is triggered during program execution. Thus, pointer information is incrementally updated when new constraints are gradually introduced by reflection. This technique on reflection handling can be used in JIT optimizations but may not be suitable for whole-program pointer analysis.

To facilitate (static) pointer analysis, Bodden et al. [2] suggested leveraging the runtime information gathered for reflective calls. Their tool, TamiFlex, 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. In addition, TamiFlex inserts runtime checks to warn the user in cases that the program encounters reflective calls that diverge from the recorded information of previous runs. Elf is complementary to TamiFlex by resolving reflective calls statically rather than dynamically.

Soot [19] is a static analysis and optimization framework for Java. For reflective callsites found in the standard libraries, the Soot developers have discovered a list of their possible targets manually. Soot has now a special built-in support for TamiFlex [2], allowing some reflective call targets to be found dynamically.

Others Braux and Noyé [3] provided offline partial evaluation support for reflection in order to perform aggressive compiler optimizations for Java applications. It transforms a program by compiling away the reflection code into regular operations on objects according to their concrete types that are constrained manually. Elf can be viewed as a tool for inferring such constraints automatically.

To increase code coverage, some static analysis tools [4, 21] allow the user 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. [16] introduced 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. Elf may also utilize domain knowledge to analyze some particular configuration files, but only for those reflective call sites that cannot be resolved effectively.

Finally, the dynamic analyses [2, 5] work in the presence of both dynamic class loading and reflection. Nguyen, Potter and Xue [12, 22, 23] introduced an interprocedural side-effect analysis for open-world Java programs (by allowing dynamic class loading but disallowing reflection). Like other static reflection analyses [4, 8, 20, 21], Elf can presently analyze closed-world Java programs only.

7 Conclusion

Reflection analysis is difficult but increasingly important both for sound and for under-approximate pointer analysis for Java applications, especially framework-based applications. This paper advances the state-of-the art in reflection analysis for Java, by (1) presenting some useful findings on reflection usage in Java benchmarks and applications, (2) introducing a self-inferencing resolution approach, (3) contributing an open-source implementation consisting of 207 Datalog rules, and (4) demonstrating the effectiveness of our new reflection analysis.

Acknowledgements. The authors wish to thank the anonymous reviewers for their valuable comments, the DOOP team for making DOOP available, and LogicBlox Inc. for providing us its Datalog engine. This work is supported by an ARC grant, DP130101970.

References

1. M. Berndl, O. Lhoták, F. Qian, L. J. Hendren, and N. Umanee. Points-to analysis using BDDs. PLDI '03, pages 103–114.
2. 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, pages 241–250.
3. M. Braux and J. Noyé. Towards partially evaluating reflection in Java. PEPM '00, pages 2–11.
4. M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA '09, pages 243–262.
5. M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.
6. G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI '13, pages 423–434.
7. O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. CC '03, pages 153 – 169.
8. B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. APLAS '05, pages 139–160.
9. B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. Technical report, Stanford University, 2005.
10. Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. In *CC '13*, pages 61–81.
11. A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1), 2005.
12. P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. In *ACSC '05*, pages 9–18.
13. L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation. In *ASE '12*, pages 270–273.
14. L. Shang, X. Xie, and J. Xue. On-demand dynamic summary-based points-to analysis. CGO '12, pages 264–274.
15. Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. POPL '11, pages 17–30.
16. M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. OOPSLA '11, pages 1053–1068.
17. M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. PLDI '06, pages 387–400.
18. M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, chapter Alias Analysis for Object-Oriented Programs.
19. R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. CASCON '99.
20. WALA. *T.J. Watson Libraries for Analysis, http://wala.sf.net.*
21. J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI '04, pages 131–144.
22. J. Xue and P. H. Nguyen. Completeness analysis for incomplete object-oriented programs. In *CC '05*, pages 271–286.
23. J. Xue, P. H. Nguyen, and J. Potter. Interprocedural side-effect analysis for incomplete object-oriented software modules. *Journal of Systems and Software*, 80(1):92–105, 2007.

A Artifact Description

Authors of the artifact. Design: Yue Li, Tian Tan and Jingling Xue. Developers: Tian Tan and Yue Li.

Summary. The artifact includes all the four analyses evaluated in the paper, namely DOOP, ELF and two variations of ELF, ELF^d and ELF^p.

Content. The artifact package includes:

- an `index.html` file containing the detailed instructions for using the artifact and for reproducing the experimental results in the paper;
- the four analysis tools, DOOP, ELF, ELF^d and ELF^p;
- a modified version of the fact generator provided by DOOP;
- a *Python* script `exec.py` (and some auxiliary scripts) for driving all the provided analyses and formatting the output results;
- all the necessary JREs, applications and benchmarks analyzed.

ELF and its two variations, ELF^d and ELF^p, are all built on top of DOOP (version r160113). ELF presently consists of 207 rules (with about 1800 LOC). To simplify repeatability of our experiments, we have provided these analysis configurations directly instead of DOOP patches.

Getting the artifact. The artifact endorsed by the Artifact Evaluation Committee is available free of charge as supplementary material of this paper on SpringerLink. The latest version of our code is available at `http://www.cse.unsw.edu.au/~jingling/elf`.

Tested platforms. The artifact works on 64-bit Linux (Ubuntu 13.10 LTS in our case) machine with at least 8 GB of RAM.

License. MIT license (http://opensource.org/license/MIT)

MD5 sum of the artifact. 024b6fccc7c7bb2edc7dac443f457761

Size of the artifact. 358M