Understanding and Analyzing Java Reflection

by

Yue Li

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY
IN THE SCHOOL
OF
COMPUTER SCIENCE AND ENGINEERING

THE UNIVERSITY OF
NEW SOUTH WALES

SYDNEY · AUSTRALIA

26 July, 2016

PLEASE TYPE

THE UNIVERSITY OF NEW SOUTH WALES
Thesis/Dissertation Sheet

Surname or Family name: **Li**

First name: **Yue** Other name/s:

Abbreviation for degree as given in the University calendar: **PhD**

School: **Computer Science and Engineering** Faculty: **Engineering**

Title: **Understanding and Analyzing Java Reflection**

Abstract 350 words maximum: (PLEASE TYPE)

Java reflection is increasingly used in a range of software and framework architectures. It allows a software system to examine itself and make changes that affect its execution at run-time, but creates significant challenges to static analysis. This is because the usages of reflection are quite complicated in real-world Java programs, and their dynamic behaviors are mainly specified by string arguments, which are usually unknown statically. As a result, in almost all the static analysis tools, reflection is either ignored or handled partially, resulting in missed, important behaviors, i.e., unsound results. Improving or even achieving soundness in reflection analysis will provide significant benefits to many clients, such as bug detectors, security analyzers and program verifiers.

This thesis first introduces what Java reflection is, and conducts an empirical study on how reflection is used in real-world Java applications. Many useful findings are concluded for guiding the designs of more effective reflection analysis methods and tools. Based on this study, this thesis then presents two new techniques for handling reflection statically: a self-inferencing analysis called ELF, and a soundness-guided analysis called SOLAR.

ELF is able to analyze reflection more effectively than the previous string resolution approach by exploiting a self-inferencing property found in our study. Such property is inherent in almost every reflective call, but not fully exploited by existing methods. ELF could make a disciplined trade-off among soundness, precision and scalability, while also discovering usually more reflective targets than in the previous work.

SOLAR allows its soundness to be reasoned about when some reasonable assumptions are met, and yields significantly improved under-approximations otherwise. In addition, SOLAR is able to accurately identify where reflection is analyzed unsoundly or imprecisely and it provides a mechanism to guide users to iteratively refine the analysis results by lightweight annotations until their specific requirements are satisfied.

For both ELF and SOLAR, this thesis presents their methodologies and formalisms and evaluates them against the state-of-the-art solutions with a set of large Java benchmarks and applications. The experimental results demonstrate their effectiveness as the new state-of-the-art reflection analyses in practice. Both ELF and SOLAR have been made available as open-source tools.

Declaration relating to disposition of project thesis/dissertation

I hereby grant to the University of New South Wales or its agents the right to archive and to make available my thesis or dissertation in whole or in part in the University libraries in all forms of media, now or here after known, subject to the provisions of the Copyright Act 1968. I retain all property rights, such as patent rights. I also retain the right to use in future works (such as articles or books) all or part of this thesis or dissertation.

I also authorise University Microfilms to use the 350 word abstract of my thesis in Dissertation Abstracts International (this is applicable to doctoral t

.

The University recognises that there may be exceptional circumstances requiring restrictions on copying or conditions on use. Requests for restriction for a period of up to 2 years must be made in writing. Requests for a longer period of restriction may be considered in exceptional circumstances and require the approval of the Dean of Graduate Research.

FOR OFFICE USE ONLY Date of completion of requirements for Award:

THIS SHEET IS TO BE GLUED TO THE INSIDE FRONT COVER OF THE THESIS

ORIGINALITY STATEMENT

'I hereby declare that this submission is my own work and to the best of my knowledge it contains no materials previously published or written by another person, or substantial proportions of material which have been accepted for the award of any other degree or diploma at UNSW or any other educational institution, except where due acknowledgement is made in the thesis. Any contribution made to the research by others, with whom I have worked at UNSW or elsewhere, is explicitly acknowledged in the thesis. I also declare that the intellectual content of this thesis is the product of my own work, except to the extent that assistance from others in the project's design and conception or in style, presentation and linguistic expression is acknowledged.'

Signed ……..

Date ……..

Abstract

Java reflection is increasingly used in a range of software and framework architectures. It allows a software system to examine itself and make changes that affect its execution at run-time, but creates significant challenges to static analysis. This is because the usages of reflection are quite complicated in real-world Java programs, and their dynamic behaviors are mainly specified by string arguments, which are usually unknown statically. As a result, in almost all the static analysis tools, reflection is either ignored or handled partially, resulting in missed, important behaviors, i.e., unsound results. Improving or even achieving soundness in reflection analysis will provide significant benefits to many clients, such as bug detectors, security analyzers and program verifiers.

This thesis first introduces what Java reflection is, and conducts an empirical study on how reflection is used in real-world Java applications. Many useful findings are concluded for guiding the designs of more effective reflection analysis methods and tools. Based on this study, this thesis then presents two new techniques for handling reflection statically: a self-inferencing analysis called Elf, and a soundness-guided analysis called Solar.

- Elf is able to analyze reflection more effectively than the previous string resolution approach by exploiting a self-inferencing property found in our study. Such property is inherent in almost every reflective call, but not fully

exploited by existing methods. ELF could make a disciplined trade-off among soundness, precision and scalability, while also discovering usually more reflective targets than in the previous work.

- SOLAR allows its soundness to be reasoned about when some reasonable assumptions are met, and yields significantly improved under-approximations otherwise. In addition, SOLAR is able to accurately identify where reflection is analyzed unsoundly or imprecisely and it provides a mechanism to guide users to iteratively refine the analysis results by lightweight annotations until their specific requirements are satisfied.

For both ELF and SOLAR, this thesis presents their methodologies and formalisms and evaluates them against the state-of-the-art solutions with a set of large Java benchmarks and applications. The experimental results demonstrate their effectiveness as the new state-of-the-art reflection analyses in practice. Both ELF and SOLAR have been made available as open-source tools.

Publications

- Tian Tan, **Yue Li** and Jingling Xue. Making k-Object-Sensitive Pointer Analysis More Precise with Still k-Limiting. *23rd International Static Analysis Symposium (SAS'16).*

- **Yue Li**, Tian Tan, Yifei Zhang and Jingling Xue. Program Tailoring: Slicing by Sequential Criteria. *30th European Conference on Object-Oriented Programming (ECOOP'16).*
  (Distinguished Paper Award)

- **Yue Li**, Tian Tan and Jingling Xue. Effective Soundness-Guided Reflection Analysis. *22nd International Static Analysis Symposium (SAS'15).*

- **Yue Li**, Tian Tan, Yulei Sui and Jingling Xue. Self-Inferencing Reflection Resolution for Java. *28th European Conference on Object-Oriented Programming (ECOOP'14).*

- Yulei Sui, **Yue Li** and Jingling Xue. Query-Directed Adaptive Heap Cloning For Optimizing Compilers. *11th Annual IEEE/ACM International Sym- posium on Code Generation and Optimization (CGO'13).*
  (Best Paper Award)

Acknowledgements

First, I owe very deep thanks to my dear supervisor, Jingling Xue, for giving me the freedom, trust, encouragement and resources to pursue my PhD research. He patiently taught me what good research is and how to convey it in both papers and talks. I cannot hope to list all the ways that he has helped me improve as a researcher, but perhaps the most important things I learnt from him were: (1) see difficulties as the good opportunities to improve your work and (2) always be enthusiastic about your research. I believe his influence will continue to inspire me in my future pursuits.

Second, my deep gratitude goes to Tian Tan, who was my junior brother-in-learning, my student and also a very good friend when I was in NPU China. During the last three years, Tian has been a great collaborator of my research work and also a very good roommate. I really enjoy the time that I spent with him on both research and life in Sydney.

Third, many thanks go to the other members of CORG group at UNSW including Yi Lu, Xinwei Xie, Lei Shang, Qing Wan, Yulei Sui, Peng Di, Ding Ye, Yu Su, Sen Ye, Hao Zhou, XiaoKang Fan, Hua Yan, Feng Zhang, Yifei Zhang, JieYuan Zhang and Jie Liu for making my time at UNSW memorable. Particularly, I am indebted to Yi (now at Oracle) for patiently answering my questions about programming languages when I worked with him. I am grateful to Yulei for

introducing me to this group and helping me during my early research stage. I also thank Ding and Peng for providing me so many good suggestions for my various questions regarding both research and life. I will not forget the happy time talking with Hao, especially when the pressure finds us and I will always remember the cheerful moments when hanging out with Yifei, Jie, Jieyuan, Feng and Tian.

Fourth, I thank the Doop team for making Doop (such a good pointer analysis framework for Java) public available, as both my thesis work Elf and Soar are built on top of it. Especially, I'd like to show my gratitude to Prof. Yannis Smaragdakis (the leader of the Doop team) and the LogicBlox Inc. for kindly providing me the Datalog engine to run Doop. In addition, I want to thank Dr. Cristina Cifuentes for kindly inviting me to give a talk about one of my thesis chapters at Oracle Labs Australia and I will remember the comfortable discussions with Cristina, Lian, Yi and the other nice members at Oracle Labs Australia.

Fifth, many thanks go to the two international examiners of my thesis, Prof. Anders Møller at Aarhus University and Prof. Ana Milanova at Rensselaer Polytechnic Institute, for their valuable time and comments.

Finally, I owe my deepest gratitude to my parents and my elder sister Lin, for their constant support and endless love. I dedicate this thesis to them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Java reflection allows a software system to examine itself and make changes that affect its execution at run-time. Such dynamic feature could significantly ease the development and maintenance of Java programs for some specific requirements — such as flexibly integrating the third-party code or configuring the main behaviors of a program according to the environment in a decoupled way. Considering such benefits, reflection is increasingly used in a broad array of Java programs: 15 out of 16 popular Java benchmarks and applications (studied in this thesis) are found to use reflection, and a recent study on Android applications (written by Java) reports that 88% of 500 top free apps on Google Play contain reflection usages [106].

Static analysis is a useful technique for many applications, such as bug and security vulnerability detection, optimization and verification [61, 100, 99, 86, 66, 82, 62, 67, 39, 10, 38, 32, 13, 12, 43, 49, 105, 5, 11]. In recent years, there has been a surge in demand for utilizing static analysis to handle reliability and security problems (mainly due to its good code coverage), resulting in many promising ideas and tools in both academia and industries [87, 76, 57, 91, 104, 94, 89, 9, 33, 65, 63, 64, 3, 44, 50, 25, 22, 23, 35]. When applying static analysis for Java

programs, reflection has always been a major obstacle as it is a dynamic feature, which is hard to analyze statically [52, 53, 17, 74, 46, 47, 48, 4, 51]. However, we still have to deal with reflection in static analysis since it is so important and in high demand (especially for the complex and large software today). Not modeling it can render much of the codebase invisible for analysis [51], resulting in missed important behaviors, i.e., unsound analysis results. As a result, improving or even achieving soundness in reflection analysis will provide significant benefits to many clients, such as bug detectors, security analyzers, optimizing compilers and program verifiers [51, 47, 74, 4, 68, 106].

1.1 Challenges

Reflection is an open hard language feature for the static analysis community [51] and there are many real-world complaints about it in terms of program analysis:

> *"Reflection usage and the size of libraries/frameworks make it very difficult to scale points-to analysis to modern Java programs"* [93];
>
> *"Reflection makes it difficult to analyze statically"* [68];
>
> *"In our experience [24], the largest challenge to analyzing Android apps is their use of reflection ..."* [4].

As a result, in the past years, almost all the papers regarding static analysis for object-oriented languages, such as Java, see reflection as a separate assumption (or do not mention it), and most analysis tools either ignore reflection completely or handle it partially and ineffectively.

There are three main reasons for this knotty problem:

- The Java reflection API is fairly large and its usages in Java programs are

quite complex. What remains unclear is how to analyze which of its reflection methods to satisfy the practical needs of an analysis.

- The dynamic behaviors introduced by the reflective calls are mainly specified by their string arguments, which are usually unknown statically (e.g., read from configuration files or downloaded from the Internet).

- A reflection analysis usually works inter-dependently with a pointer analysis [72, 52, 17, 90, 81], with each being both the producer and consumer of the other. When some reflective calls are not yet resolved, the pointer information currently available can be over- or under-approximate. Care must be taken to ensure that the reflection analysis helps to increase soundness (coverage) while still maintaining sufficient precision for the pointer analysis. Otherwise, both analyses would be unscalable.

A (more) sound reflection analysis means that (more) all true reflective targets (available at runtime) are resolved statically. In practice, reflection analysis inevitably makes a trade-off among soundness, precision, scalability, and (sometimes) automation.

1.2 Previous Approaches

Generally, previous reflection analysis mainly uses string analysis, especially when the string arguments to reflective calls are string constants, to resolve reflective targets. Currently, this mainstream approach for handling Java reflection is still adopted by many analysis tools [95, 93, 21, 92, 19]. However, as described above, this method would fail in many situations where string arguments are unknown, resulting in achieving limited soundness and precision. In addition, previous reflection analysis [52, 53, 17, 93, 21], including the very recent one [74], is unaware

of (1) how sound the reflection analysis is, and (2) where it is resolved unsoundly. We argue its importance in practice due to the following reasons.

- If (1) is unknown, users would be unsure (or lose confidence) about the effectiveness of any analysis results. For example, a bug detection analysis reports no bugs; but users cannot trust such results if many reflective calls are resolved unsoundly, as much runtime behaviors introduced by those reflective calls would be missed.

- If (2) cannot be answered effectively, for certain high-quality analysis where nearly soundness is required or some clients (e.g., verification) where soundness is demanded, users would hardly have the chance to improve the analysis by manual efforts such as annotations.

1.3 Contributions

This thesis uncovers the mysterious veil of Java reflection and changes the informed opinion in the program analysis community that "*In terms of static analysis, reflection is a dynamic feature and is nearly impossible to handle effectively*" by making the following contributions.

- We conduct a comprehensive study about reflection usages in real-world Java programs and conclude many useful findings in terms of program analysis.

  - There are about two-hundred reflection API provided by Java and they are briefly explained in documentations. However, some questions such as which reflection API are necessary to handle in program analysis, what the relationship is among them, how they are used in applications, and what we can learn from their usages, etc., still remain unclear.

To answer these questions, we select 16 representative Java programs and manually study 1,423 reflective call sites. Many useful findings are summarized including the insight to design more effective reflection handling approaches and the advice to build more practical reflection analysis tools.

- We introduce a self-inferencing reflection analysis called ELF.
  - ELF is able to analyze reflection more effectively than the previous string resolution approach, and does so by exploiting a self-inferencing property found in our study. Such property is inherent in almost every reflective call, but not fully exploited by existing methods.
  - ELF could make a disciplined trade-off among soundness, precision and scalability, while also discovering usually more reflective targets than the string resolution approach.
- We introduce a soundness-guided reflection analysis called SOLAR.
  - SOLAR allows its soundness to be reasoned about when some reasonable assumptions are met and yields significantly improved under-approximations otherwise.
  - SOLAR is able to accurately identify where reflection is analyzed unsoundly or imprecisely, making users aware of the effectiveness of their analysis results regarding reflection handling (as discussed before).
  - SOLAR provides a mechanism to guide users to iteratively refine the analysis results by lightweight annotations until their specific requirements are satisfied, which enables reflection to be analyzed under control.

- We implement both ELF and SOLAR on top of DOOP (a state-of-the-art pointer analysis tool for Java) and release them as open-source tools at

  http://www.cse.unsw.edu.au/~corg/elf

  http://www.cse.unsw.edu.au/~corg/solar

  respectively. Presently, both of them can also output their reflection analysis results with the format that is supported by SOOT (a popular framework for analyzing and transforming Java and Android applications), which enables SOOT's clients to use their results directly.

1.4 Thesis Organization

The rest of this thesis is organized as follows.

In Chapter 2, we initially introduce the background knowledge of Java reflection in terms of program analysis. Then we conduct an empirical study on how reflection is used in real-world Java programs. The study results are given finally in the form of seven remarks, which are helpful for designing (building) effective reflection analysis (tools).

In Chapters 3 and 4, we respectively introduce ELF and SOLAR following the order of methodology, formalism, implementation and evaluation. At the end of each chapter, we also briefly discuss their limitations. As ELF and SOLAR are not only two reflection analysis methods, but also two practical open-source tools, we adopt two different kinds of formalisms to describe them.

In Chapter 3, ELF is formulated as a set of Datalog rules, which have been the basis of several pointer analysis tools [21, 19, 95] and used in many literatures [52, 17, 75, 73, 37, 77, 74, 36, 70]. The main advantage of the Datalog rules is that

the specification is close to the actual implementation, easing the readers to better understand our tools.

In Chapter 4, SOLAR is formalized as a set of constraint rules for the whole Java language (but with its core reflection API), which allows the readers to understand our methods systematically and precisely.

In Chapters 5 and 6, we discuss the related work and give the conclusions respectively.

Chapter 2

Understanding Java Reflection

Reflection, as a concept for computation systems, dates from Brian Smith's doctoral dissertation [78]. Generally, each computation system is related to a domain: it incorporates internal structures representing the domain, including the data in the domain and the program describing how these data may be manipulated [56]. Usually, a computation system is called a reflective system, if the following two conditions are satisfied. First, the system has its own representation in the domain as a kind of data to be examined and manipulated. Second, the system and its representation are causally connected, i.e., a change to the representation implies a change to the system and vice versa.

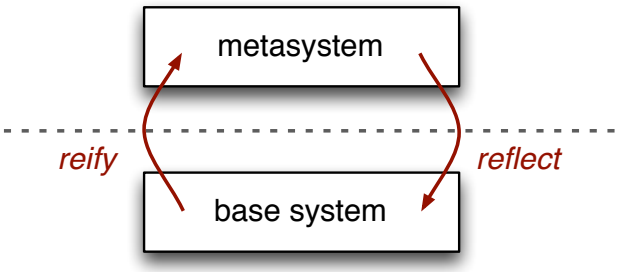


Figure 2.1: The execution model of the reflective system in [28].

Figure 2.1 presents a straightforward model for a reflective system introduced

by Friedman and Wand [28] and explained in Forman and Forman [27]. Briefly, the base system should be **reified** into its representation before a metasystem can operate. Then the metasystem examines and manipulates using the reified representation. If any changes are made by the metasystem, the effects will also be **reflected** in the behavior of the base system.

Providing full reflective ability as shown above is hardly acceptable in practice, as it would introduce implementation complexity and performance problems [18]. As a result, in modern programming languages, only partial reflective ability is supported. Java reflection supports introspection (the ability to examine the structure and the state of a system), but not intercession (the ability to modify the semantics of the underlying programming language from within the language itself) and self-modification (the ability of a system to change its own structure) [15]. In addition, Java reflection does not have a reify operation in Figure 2.1 to turn the basic (running) system (including stack frames, etc.) into a representation (data structure) that is passed to a metasystem. Instead, a kind of metarepresentation, called metaobject, exists when the system begins running and persists throughout the execution of the system [27].

In the following chapter, we first show how such metaobject is used in Java to fulfill various reflective tasks in Section 2.1. In Section 2.2, we give the setups of our study on how reflection is used in real-world Java programs, including what applications we select and how their reflective calls are collected and analyzed. Finally, the study results are summarized and discussed in Section 2.3.

2.1 Background

There are mainly four kinds of metaobjects (`Class`, `Method`, `Field` and `Constructor`

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

Figure 2.2: An example of reflection usage in Java.

objects) and everything in Java reflection begins from the `Class` object. Figure 2.2 shows the typical usages of the first three kinds of metaobjects. 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 5 and `getField()` in line 7 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 6) and a `Field` object can be commanded to access the field that it represents (lines 8 and 9).

In terms of static analysis, we care more about how reflection affects pointer analysis (as described in Section 1.1) as the later is the key to precisely over-approximate the data flow information on which a static analysis usually relies [75, 37, 73, 77, 83, 80, 42, 6, 40, 103, 8, 71, 54, 101, 88, 84, 85, 98, 45, 29, 102, 30, 2, 96, 55, 58, 36, 60, 59, 26]. For example, call graphs are mostly built by pointer analysis in modern analysis tools [21, 93, 92, 19, 1]. Thus as far as pointer analysis is concerned, in reflection analysis, we usually consider the pointer-affecting methods

in the Java reflection API [52, 21]. We can divide such reflective methods 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 5 and `getField()` in line 7, 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, 6, 8 and 9, 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. 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 5 in Figure 2.2). `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 6 in Figure 2.2.

2.2 Study Setups

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 16 representative Java programs, including all eleven DaCapo benchmarks (2006-10-MR2) [7], 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 the DaCapo benchmarks includes an older version of `Eclipse` (version 3.1.2). We finally 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 [92] to pinpoint the calls to reflection methods in the bytecode of a program. To understand the common reflection usage, we consider only the reflective calls found in the application classes and their dependent libraries but exclude the standard Java libraries. To increase the code coverage for the five applications considered, we include the `jar` files whose names contain the names of these applications (e.g., `*jetty*.jar` for `Jetty`) and make them available under the *process-dir* option supported by Soot. For `Eclipse4`, we use `org.eclipse.core.runtime.adaptor.EclipseStarter` to let Soot locate all the other jar files used.

We manually inspect the reflection usage in a program in a demand-driven manner, starting from its side-effect methods, assisted by *Open Call Hierarchy* in `Eclipse`, by following their backward slices. For a total of 609 side-effect call sites examined, 510 call sites for calling entry methods and 304 call sites for calling member-introspecting methods are tracked and analyzed. As a result, 1,423 reflective call sites and the code around them are examined in our study.

2.3 Study Results

Below we describe our seven main findings on reflection usages in our empirical study and summarize them as remarks separately, which are expected to be helpful to guide the designs of more effective reflection analysis approaches and tools.

2.3.1 Side-Effect Methods

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$*(...)

Table 2.1: 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).

Table 2.1 lists a total of nine side-effect methods that can possibly modify or use (as their side effects) the pointer information in a program. We take `newInstance()`, `invoke()` and `newProxyInstance()` as examples to explain. The side effect of `newInstance()` is allocating an object with the type represented by its metaobject *clz* or *ctor* (say A) and initializing the constructor of class A. The side effect of `invoke()` is usually a virtual call (when the first argument of `invoke()`, say o, is not NULL) and the call's receiver object is pointed to by o. The side effect of `newProxyInstance()` is difficult to analyze statically since the proxy class

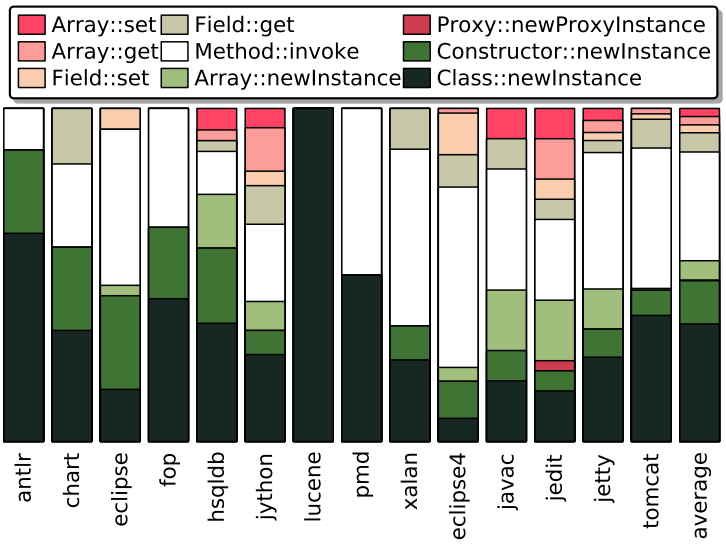


Figure 2.3: Side-effect methods.

`Proxy$*` is generated dynamically according to its arguments. One possible solution is to analyze `Proxy.newProxyInstance()` according to its semantics. A call to this method returns a `Proxy` object, which has an associated invocation handler object that implements the `InvocationHandler` interface. A method invocation on a `Proxy` object through one of its `Proxy` interfaces will be dispatched to the `invoke()` method of the object's invocation handler.

Figure 2.3 depicts the percentage frequency distribution of these nine side-effect methods in the 14 programs studied. We can see that `newInstance()` and `invoke()` are the ones that are most frequently used (46.3% and 32.7%, respectively, on average). Both of them are handled by prior pointer analysis tools [21, 93, 95, 92]. However, `Field`- and `Array`-related side-effect methods, which are also used in many programs, are ignored by most of them (only handled by the latest version of Doop (r5459247-beta), our Elf and Solar). Note that `newProxyInstance()` is used in `jEdit` only.

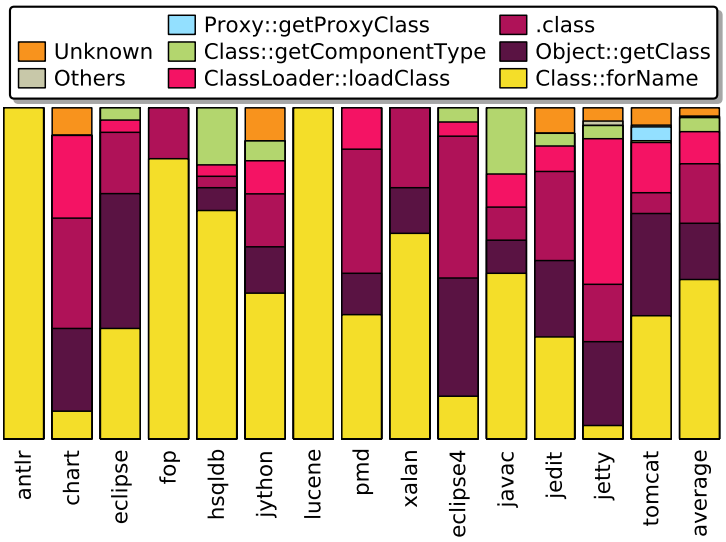


Figure 2.4: Entry methods.

Remark 1. *Reflection analysis should at least handle* `newInstance()` *and* `invoke()` *as they are the most frequently used side-effect methods (79% on average) which may significantly affect a program's behavior; otherwise, much of the codebase may be invisible for analysis. Effective reflection analysis should also consider* `Field`*- and* `Array`*-related side-effect methods, as they are also commonly used.*

2.3.2 Entry Methods

Figure 2.4 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()`, `.class`, `getClass()` and `loadClass()` are the top four most frequently used entry methods (48.1%, 18.0%, 17.0% and 9.7% respectively). Particularly, the class loading strategy could be configured in `forName()` (the interface which supports two arguments) and `loadClass()`. In practice, `forName()` usually loads classes by the system class loader and `loadClass()` is usually overwritten in customer class loaders, especially in some framework applications such as `Tomcat` and `Jetty`.

Remark 2. *Reflection analysis should handle `Class.forName()`, `getClass()`, `.class` and `loadClass()` which are the main four entry methods to retrieve `Class` objects. `getComponentType()` should also be modeled if `Array`-related side-effect methods are analyzed, as they are usually used together.*

2.3.3 String Constants and String Manipulations

Of the six types of entry methods illustrated in Figure 2.4, `Class.forName()` and `loadClass()` each have a `String` parameter. In addition, `Class` contains `getDeclaredMethod(String,...)` and `getMethod(String,...)` for returning a `Method` object named by the first parameter and `getDeclaredField(String)` and `getField(String)` for returning a `Field` object named by its single parameter.

As shown in Figure 2.5, 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()`

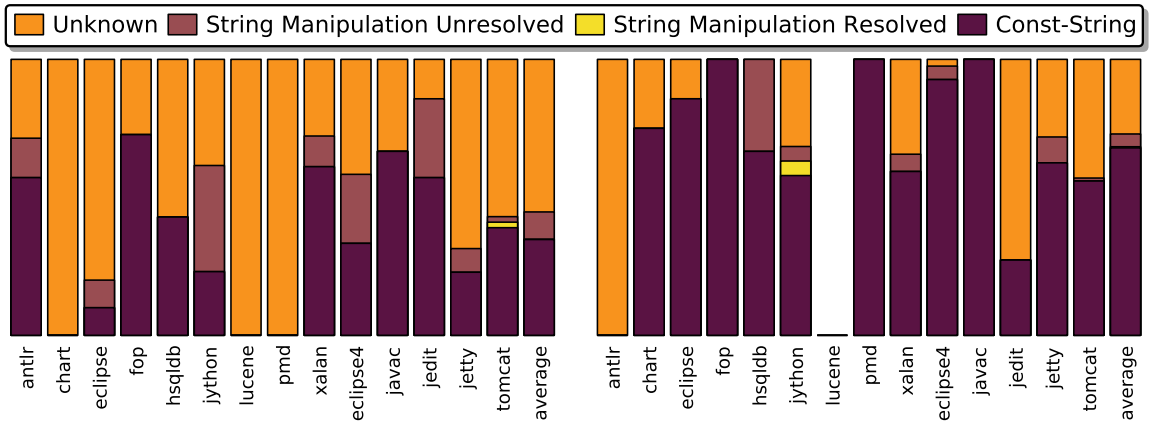
and `append()`. In fact, the cases we found in the study exhibit much more complicated scenarios, which are beyond the ability of modeling partially the `java.lang.String`-related API. Thus, both ELF and SOLAR do not handle string manipulations presently. However, incomplete information about class/method/-field names (i.e., partial string information) can be exploited to infer the reflective targets [74].

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, e.g., DOOP (r160113), etc., that ignore string constants flowing into some member-introspecting methods may be imprecise.

Remark 3. *Resolving reflective targets by string constants does not always work: on average, only 49% reflective call sites (where string arguments are used to specify reflective targets) use string constants. Fully resolving non-constant string arguments by string manipulation, although mentioned in some literatures [52, 53, 14], is very hard to achieve in practice.*

2.3.4 Retrieving an Array of Member 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()` call sites called on an array of `Method` objects returned from `getMethods()`



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

Figure 2.5: Classification of the `String` arguments of two entry methods, `forName()` and `loadClass()`, and four member-introspecting methods, `getMethod()`, `getDeclaredMethod()`, `getField()` and `getDeclaredField()`.

and 15 `fld.get()` and `fld.set()` call sites called on an array of `Field` objects returned by `getDeclaredFields()`. Ignoring such methods as in prior work [52, 95, 93, 92] may lead to many missed methods in the call graph of a program.

Remark 4. *In member-introspecting methods, get(Declared)Methods/Fields /Constructors() (which return an array of member metaobjects) are usually ignored by most reflection analysis tools; however, they play an important role in certain applications for both method invocations and field accesses.*

2.3.5 Static or Instance Members

In most reflection analysis literatures [52, 53, 74, 46], reflective targets are assumed as instance members by default. Thus the side-effect methods such as `invoke()`, `get()` and `set()`, are usually considered as a virtual call, an instance field access and an instance field modification, respectively (see Table 2.1 for details). However,

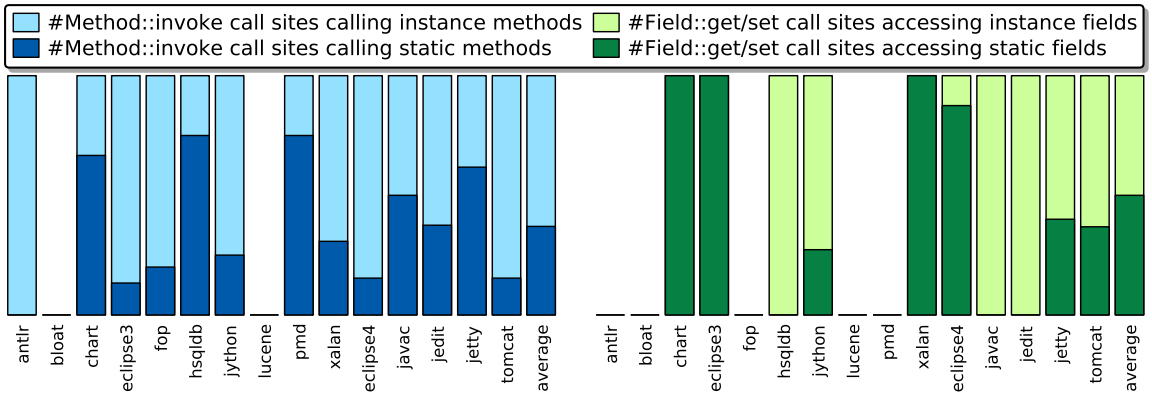


Figure 2.6: The percentage frequency distribution of side-effect call sites calling/accessing instance or static members.

if the reflective targets are static members, the approach for handling these side-effect methods would be different; in addition, some reflection analysis strategies may have to be changed too. As shown in Figure 2.6, on average, 37% of the `invoke()` call sites are found to call static methods and 50% of the `get()`/`set()` call sites are found to access/modify static fields.

Remark 5. *Static methods/fields are called/accessed as frequently as instance ones in Java reflection. As static members are handled differently from instance members, we expect future reflection analysis approaches (or tools) to consider both of them in order to meet the client needs satisfactorily.*

2.3.6 `newInstance()` Resolution by Casts

In Figure 2.2, when `cName` is not string constant, the (dynamic) type of `obj` created by `newInstance()` in line 4 is unknown. For this case, Livshits et al. [52] propose to infer the type of `obj` by leveraging the cast operations which intra-procedurally post-dominate the `newInstance()` call site. If the cast type is `A`, the type of `obj`

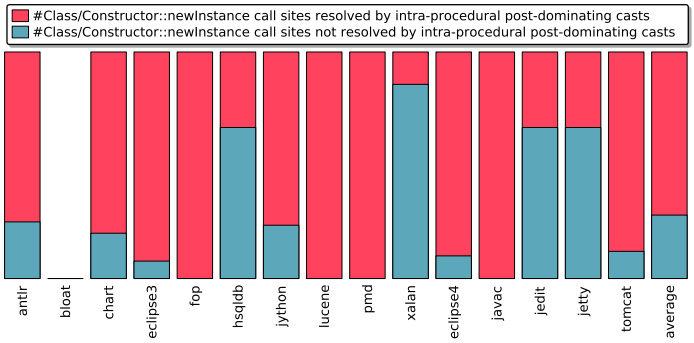


Figure 2.7: `newInstance()` resolution by intra-procedural post-dominating casts.

must be `A` or one of its subtypes assuming that the cast operation does not throw any exceptions. This approach has been implemented in many pointer analysis tools to resolve reflection [21, 93, 95]. However, as shown in Figure 2.7, it does not always work: on average, 28% of the`newInstance()` call sites have no such intra-procedural post-dominating casts, rendering this approach ineffective. Since `newInstance()` is the most widely used reflection method, their unresolved call sites could cause static analysis to miss much true codebase as shown in Section 4.6.5.1. Hence, we need a better solution to handle `newInstance()`.

Remark 6. *The method for resolving `newInstance()`, by leveraging the intra-procedural post-dominating cast operations, fails to work for 28% of the `newInstance()` call sites used. As `newInstance()` is important for reflection analysis (Remark 1), a more effective approach for handling it is required.*

2.3.7 Self-Inferencing Property

As shown in Figure 2.2, the names of the reflective targets are specified by the *string arguments* (e.g., `cName`, `mName` and `fName`) at the entry and member-introspecting reflective calls; thus string resolution has been the representative method for static

reflection analysis in the last decade. However, if the string values are unknown statically (e.g., read from external files, command lines or composed by complicated string operations), the related reflective calls would be ignored, rendering the corresponding codebase or operations invisible to analysis. In this situation, the last effort made is to handle `newInstance()` if and only if there are some cast operations which intra-procedurally post-dominate its call site (Section 2.3.6).

However, in our study, we find that there are rich hints in reflective code at their usage sites which could be leveraged to make reflection analysis more effective, even if the string values are partially or fully unknown. In the following, we first take three real examples to show what these hints are, and then give their informal definition, called self-inferencing property. Finally, we explain why self-inferencing property is a pervasive fact for Java reflection and discuss its potential to help reflection analysis.

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); ...}
    }
```

Figure 2.8: Partial self-inferencing property of reflective method calls: the size and the dynamic types of `parameters` in `invoke()`.

Example 1: reflective method calls (Figure 2.8).

The method name (the first argument of `getMethod()` in line 174) is statically unknown as it is read from command line `cmd`. However, the target method (represented by `method`) can be deduced from the second argu-

ment (`parameters`) of the corresponding side-effect call `invoke()` in line 175. `parameters` is an array of objects and has only one element (line 155). As the type of the object pointed to by `this` is `FrameworkCommandInterpreter` which has no subtypes, we can thus infer that the signature of the target method must have only one argument and its declared type must be `FrameworkCommandInterpreter` or one of its supertypes.

Application:Eclipse (v4.2.2):

Class:org.eclipse.osgi.framework.internal.core.Framework

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

Figure 2.9: Partial self-inferencing property of reflective field accesses: the cast type and the `null` argument at `get()`.

Example 2: reflective field accesses (Figure 2.9).

`factoryField` (line 1683) 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 and the `null` argument, we know that the call to `get()` may only access the static fields of `URLConnection` with the type `java.net.ContentHandlerFactory`, its supertypes or its subtypes. Otherwise, all the fields in `URLConnection` must be assumed.

```
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 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);...}
    }
```

Figure 2.10: Partial self-inferencing property of reflective field modifications: the dynamic type of `value` and the `null` argument at `set()`.

Example 3: reflective field modifications (Figure 2.10).

Like the case in Figure 2.9, the field object in line 290 is read from an array of field objects created in line 302. This code pattern appears one more time in line 432 in the same class, i.e., `org.eclipse.osgi.util.NLS`. According to the two arguments at `set()` (line 290), we can deduce that the target field (to be modified in line 290) is static (by `null`) and its declared type must be `java.lang.String` or one of its supertypes (by the type of `value`).

Informally, for each side-effect call site where reflection is used, all the information of its arguments (e.g., their types, size, etc.) and the possible downcasts on its returned values, together with the possible string information at the corresponding entry and member-introspecting call sites, are called self-inferencing property.

We argue that self-inferencing property is a pervasive fact about Java reflection, rather than only existing in some special reflective code. This is due to the object-oriented programming mechanism and the reflection API design of Java.

For example, since the declared type of the object (reflectively returned or created by `get()`, `invoke()` and `newInstance()`) is `java.lang.Object`, the object must be first cast to a specific type before its further usages as a regular object (except that its dynamic type is `java.lang.Object` or it just uses the inherited methods declared in `java.lang.Object`); otherwise, the compilation would fail. As another example, the signature of the target method reflectively called at `invoke()`, must be consistent with the size and the types of the second argument of `invoke()`; otherwise, exceptions would be thrown at runtime. These inherent constraints form a rich source to resolve reflection in a disciplined way.

The self-inferencing property not only helps resolve reflection more effectively when the values of string arguments are partially known, but also offers the chances to analyze reflection even if the string values are fully unknown. For example, in many Android apps, class and method names for reflective calls are encrypted for benign or malicious obfuscation, which "*makes it impossible for any static analysis to recover the reflective call*" [68]. However, this argument is not correct in our setting, because besides the string values, some other self-inferencing hints could be available to help reflection resolution. For instance, for this `(A)invoke(o, {...})` call, the class type of the target method could be inferred by the dynamic type of `o` (by pointer analysis); in addition, its descriptor and declared return type could be deduced from `{...}` and `A` respectively, as illustrated above.

Remark 7. *Self-inferencing property is an inherent and pervasive property in the reflective code of Java programs; however, such property has not been fully exploited to analyze reflection before. We will show how this property can be leveraged in different ways (for analyzing different kinds of reflective methods as shown in Chapters 3 and 4), to make reflection analysis more effective.*

Chapter 3

Self-Inferencing Reflection Analysis

3.1 Overview

String resolution has been the mainstream approach for static reflection analysis in the past years [92, 93, 95, 52, 21]. Livshits et al. [52, 53] suggested resolving reflective calls by tracking the flow of class, method, and field names in the program. In the code from Figure 2.2, this involves tracking the flow of `cName` into `clz` in line 3, `mName` into `mtd` in line 5, and `fName` into `fld` in line 7, if `cName`, `mName` and `fName` are string constants. In addition, some string values achieved by simple string operations (e.g., `StringBuffer.append()`) may also be resolved statically [20, 52]. These string value flows are usually tracked by pointer analysis [52, 17, 41] and at the same time, as reflective calls could also affect pointer analysis as explained in Chapter 2, reflection analysis usually works as part of a pointer analysis [72, 81], with each being both the producer and consumer of the other.

In this chapter, we present a new static reflection analysis, called ELF, which is implemented on top of DOOP [21], a state-of-the-art Datalog-based pointer analysis tool for analyzing Java programs. Generally, ELF goes beyond the string resolu-

tion method by taking advantage of the self-inferencing property introduced in Section 2.3.7. As a result, many reflective targets could still be resolved effectively even if the related string values (of the names of the target classes, methods and fields) are partially or fully unknown, which enables ELF to make a disciplined trade-off among soundness, precision and scalability.

Specifically, this chapter makes the following contributions:

- We introduce a new static reflection analysis, ELF, and describe how it exploits and leverages the self-inferencing property inherent in the reflective code to resolve reflection more effectively.
- We formulate ELF in Datalog consisting of 207 rules, covering the majority of reflection methods frequently used in Java programs.
- We implement ELF in DOOP and make it public available at http://www.cse.unsw.edu.au/~corg/elf. Presently, ELF (version 0.3) can also output its reflection analysis results with the format that is supported by SOOT [92], which enables SOOT's clients to use its results directly.
- We evaluate ELF against the reflection analysis in DOOP (r160113) under the same context-sensitive Andersen's pointer analysis framework, using all 11 DaCapo benchmarks and two Java applications, `Eclipse4` and `Javac`. Our results show that ELF can make a disciplined trade-off among soundness, precision and scalability while resolving usually more reflective call targets than DOOP.

The rest of this chapter is organized as follows. Section 3.2 describes the common assumptions for static reflection analysis. Then we present the methodology of ELF in Section 3.3 and formulate ELF in Datalog rules in Section 3.4. The implementation of ELF, including what Java reflection API is handled, is introduced in

Section 3.5. Then we evaluate ELF against the reflection analysis implementation in DOOP (r160113) in Section 3.6. Finally, we briefly discuss the advantages and limitations of ELF in Section 3.7.

3.2 Assumptions

The first assumption is commonly made on static analysis [51, 72, 81, 61] and the last two are made previously on reflection analysis for Java [52, 53].

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`*.*

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

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

This is a valid practical assumption [52, 53] and has been adopted by many pointer analysis tools [95, 93, 21] as described in Section 2.3.6.

3.3 Methodology

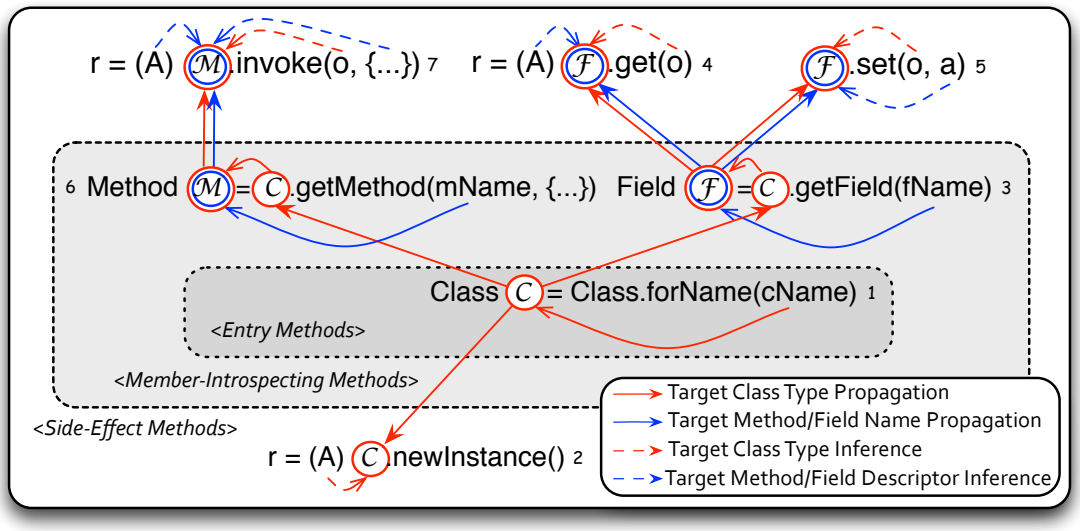


Figure 3.1: Self-inferencing reflection analysis in ELF.

Figure 3.1 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 call sites, 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 3.1):

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 call site may be missing, i.e., unknown, due to the presence of input-dependent strings (Figure 2.5). 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 intra-procedural analysis only when the array argument can be analyzed exactly element-wise.

Resolution Principle. 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 call site 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).

Our Insight. We believe that the "simultaneous constraints" resolution strategy could achieve considerable soundness while ensuring the good precision of reflection analysis. There are two reasons. First, as shown in Section 2.3.3, about 50% string constants are available for target propagation. Second, as explained in Section 2.3.7, the self-inferencing property (the resources which enable the arrows in Figure 3.1) pervasively exists in Java reflective code, which makes the probability of two simultaneous circles very high as the missing circle may probably be inferable.

How to relax ELF in the presence of just one constraint (circle) for more sound analysis will be investigated in Chapter 4. Note that in ELF, the cast operations on `newInstance()` will still have to be handled heuristically, as only one constraint exists (the blue circle is not applicable for this case), and in the target inference system of ELF, this strategy (i.e., resolving `newInstance()` by the cast operations which intra-procedurally post-dominate it) is the only one that comes from the previous work [52]. In Chapter 4, we will show how to handle `newInstance()` in a lazy way to enable a more effective reflection analysis.

Examples. Let us illustrate *Target Inference* by considering `r = (A)` $\mathcal{F}$`.get(o)` in Figure 3.1. 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 3.1, if `getMethods()` (`getFields()`) is called at Label 6 (Label 3) instead, then an array of `Method` (`Field`) objects will be returned so that *Target Propagation* from them is implicitly performed. All the other methods available in `Class` for introspecting methods/fields/constructors are handled similarly. Note that as ELF is unsound, so is the underlying pointer analysis. Therefore, a reflective call site is said to be *resolved* if at least one of its targets is resolved.

3.4 Formulation with Datalog

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 [37, 17]. The main advantage is that the specification is close to the actual implementation. Datalog has been the basis of several pointer analysis tools [21, 95, 19]. 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 related methods illustrated in Figure 3.1, where 98 = 4 (four member-introspecting methods) × 3 (three side-effect methods, `invoke()`, `get()` and `set()`) × 4 (four possible arrows in Figure 3.1) × 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 $4 \times 3 \times 1 \times 2 + 1 = 25$ scenarios) and those for performing target inference. The remaining set of rules handles `Constructor` and arrays and performs bookkeeping duties.

Section 3.4.1 gives a set of domains and input/output relations used. Section 3.4.2 describes the seven target propagation scenarios corresponding to Labels 1 – 7 in Figure 3.1. Section 3.4.3 describes four representative target inference scenarios. All the other rules (available in our tool) can be understood analogously.

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*)

Figure 3.2: Domains and input/output relations.

3.4.1 Domains and Input/Output Relations

Figure 3.2 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 [21], 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.

3.4.2 Target Propagation

We give seven target propagation scenarios corresponding to Labels 1 – 7 in Figure 3.1 when both a target method/field name and its target class type are known. The syntax of a rule is easy to understand: "←" separates the inferred fact (the *head* of the rule) from the previously established facts (the *body* of the rule).

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*).

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". Mean-

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*).

while, the points-to and call-graph relations are updated. For each resolved class, its static initialiser "<clinit>()", at the call site 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 3.3. In Scenarios P4 and P5, calls to `get()` and `set()` are analyzed, respectively. Finally, in Scenario P7, an `invoke()` call is handled, identically as in Doop [21] but differently from [52, 53], which approximates its target methods by disregarding the target object *obj*, on which the target methods are called.

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*).

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*).

3.4.3 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 3.1. 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 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.

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*)	

Figure 3.3: Input and output relations for handling target inference.

Figure 3.3 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.

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.

Unlike [52], 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 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>()"*).

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

Let us first consider the real case in Figure 2.9. 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 3.3. These type relations are captured by HierarchyType.

The same code pattern in Figure 2.9 also appears in five other places in `Eclipse4`. Elf has succeeded in deducing that only two out of a total of 13 static fields in `URLConnection` are accessed at the call site.

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,

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*).

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 2.10 is `java.lang.String`) are used to infer the target fields modified. So the second rule in Scenario P5 is enabled.

The code pattern (in Figure 2.10) 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 I4: ***Method m1=clz.getMethod(?, params); a=m2.invoke(obj, args).*** Let us consider the real case from `Eclipse4` in Figure 2.8. In line 174, the `Class` objects on which `getMethod()` is invoked can be deduced from the types of

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);*

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*).

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.

As explained in Section 3.3, we have relied on an intra-procedural analysis to perform the inference when *args* can be analyzed exactly element-wise as is the case in Figure 2.8. The MatchArgs(*args, mtd*) relation over $V \times M$ maintains target methods *mtd* found from *args* this way.

Let us now look at the rules given in Scenario I4 where *clz* points to stati-

cally known class, *type*, but the target methods at `invoke()` are unknown, just like the case illustrated in Figure 2.8. 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()` call site 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` call site are resolved, the second rule in Scenario P7 can be applied to resolve the target methods. Based on target inference, Elf has found 50 target methods at this call site, out of which 48 are real targets by manual inspection.

3.5 Implementation

We have implemented Elf with context sensitivity in Doop (r160113) [21], a state-of-the-art 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 more methods than prior work [93, 92, 95, 21]. Specifically, Elf handles the first eight side-effect methods listed in Table 2.1, all member-introspecting methods in the reflection API, and five out of the six entry methods, `forName()`, `getClass()`, `loadClass()`, `getComponentType()` and `.class`, shown in Figure 2.4. For the three side-effect methods on `Array`, `Array::newInstance()` is handled similarly as `Class::newInstance()`. We have ignored `Proxy::newProxyInstance()` in Table 2.1, `getProxyClass()` in Figure 2.4 due to the closed-world assumption (Section 3.2).

We have modified the fact generator in Doop by using an intra-procedural

post-dominance algorithm in SOOT [92] to generate the post-dominance facts, e.g., NEWINSTANCECAST and GETCAST in Figure 3.3 (and INVOKECAST not given). Note that ELF (version 0.3) can also output its reflection analysis results with the format that is supported by SOOT, enabling SOOT's clients to use ELF's results directly.

3.6 Evaluation

We evaluate ELF against the reflection analysis implemented in the state-of-the-art Java pointer analysis framework DOOP (r160113) [21]. Being unsound, both analyses make different trade-offs among soundness, precision and scalability. Our evaluation has validated the following hypotheses about our self-inferencing approach in handling reflective code.

Soundness and Precision Trade-offs 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.

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.

Again, for each analysis evaluated (due to its unsoundness), a reflective call site is considered to be resolved as long as one target is resolved from the call site.

3.6.1 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 DOOP's pointer analysis framework. Both analyses for a program are performed in the SSA form of the program generated by SOOT, under 1-call-site context sensitivity implemented in DOOP. For each program, the results presented are obtained from all the analyzed code, in both the application itself and the libraries used.

3.6.2 Soundness and Precision Trade-offs

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 (r160113). 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 DOOP (r160113), it resolves `mtd.invoke(o,args)` 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 inter-procedural assignments exactly this way.

Table 3.1 compares ELF^d and DOOP in terms of their soundness and precision trade-offs made when resolving `invoke()`, `get()` and `set()` calls. Both analyses happen to resolve the same number of reflective call sites. For a program, ELF^d usually discovers the same target methods/fields while avoiding many spurious

[1] DOOP has fixed this unsound handling in its latest version (r5459247); in addition, much more reflective methods are handled in it.

			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

Table 3.1: Comparing ELF^d and DOOP on reflection resolution. According to this particular configuration of ELF, C denotes the same number of resolved side-effect call sites in both analyses and T denotes the number of target methods/fields resolved by either.

ones introduced by DOOP. We have carried out a recall experiment for all the 11 DaCapo benchmarks by using Tamiflex [14] 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 to ELF's precision improvement (over DOOP) is its target propagation component. It is significantly more beneficial to track both constant class names and constant method/field names simultaneously rather than either alone.

3.6.3 Target Propagation vs. Target Inference

To evaluate their individual contributions to the soundness and precision trade-off made, we have included a version of ELF, named ELF^p, in which only target propagation is used. Table 3.2 is an analogue of Table 3.1 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 2.5). 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 3.1).

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.

			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

Table 3.2: Comparing ELF and ELF^p, where C and T are as defined in Table 3.1.

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 3.1) at this `get()` call site in a disciplined manner. By also exploiting the partially known information about target names (such as the common prefix `exposed_`) [74], ELF will only need to resolve the 22 target names starting with `exposed_` at this call site.

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()` call sites with their usage pattern illustrated in Figure 2.10. ELF^p discovers one target from each call site. However, ELF has discovered 553 more, one from one of the two call sites and 552 true targets at the two call sites as discussed in Section 3.4.3. 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 2.9), contributing two each, as explained in Section 3.4.3. Finally, there are two `invoke()` call sites similar to the one illustrated in Figure 2.8. ELF has discovered a total of $2 \times 48 = 96$ true target methods invoked at the two call sites. How to resolve one such `invoke()` call is also discussed in Section 3.4.3.

When analyzing Java programs, a reflection analysis works together with a pointer analysis. Each benefits from precision improvements from the other. For example, if the pointer analysis used from DOOP is 2-call-site-sensitive+heap, then the results for `invoke()` in `hsqldb` (Table 3.2) would be changed to $C = 5$ and $T = 22$ for ELF^p and $C = 8$ and $T = 83$ for ELF.

3.6.4 Effectiveness

Table 3.3 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 [37, 75], 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 [37, 75]), 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 3.3 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 3.1 and 3.2), ELF exhibits smaller numbers in eight programs in terms of all the five metrics and slightly larger ones in the remaining five programs (highlighted in bold font). Thus, these results suggest that ELF appears to strike a good trade-off 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.

	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

Table 3.4: Comparing ELF and DOOP in term of analysis times (secs).

3.6.5 Scalability

Table 3.4 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 [37, 75], `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 [52, 53], i.e., only the solid blue arrows in Figure 3.1 are considered in reflection resolution, 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.

3.7 Discussion

As described in Section 1.1, reflection analysis has to make a trade-off among soundness, precision and scalability. By exploiting and leveraging the self-inferencing property inherent in the reflective code, ELF has the potential to resolve reflection usually more soundly (than the previous string resolution approach) while maintaining good precision, as demonstrated in Section 3.6.

However, in the trade-off, ELF leans more toward precision than soundness, as it resolves a reflective target if and only if both target name and the class type are known (simultaneous presence of two circles in Figure 3.1). Thus, although the self-inferencing property pervasively exists in reflective code (as explained in Section 2.3.7), some true reflective targets may be missed under this strict strategy in some cases. How to improve or even achieve soundness (under some reasonable assumptions) in reflection analysis will be introduced in the next chapter.

		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**

Table 3.3: 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 are highlighted in **bold**.

Chapter 4

Soundness-Guided Reflection Analysis

4.1 Overview

In this chapter, we introduce SOLAR (*Sooner Or LAteR*), the first reflection analysis that is designed to allow its soundness to be reasoned about for Java programs under some reasonable assumptions and produce significantly improved under-approximations otherwise. In both (sound and unsound) settings, SOLAR handles reflection in exactly the same way with three novel aspects in its design, (N1) lazy heap modeling for reflective allocation sites, (N2) collective inference for improving the inferences on related reflective calls, and (N3) automatic identification of "problematic" reflective calls that may threaten the soundness, precision and scalability of the analysis.

What is the rationale behind (N1)? One major challenge faced by reflection analysis is how to handle effectively reflective object creation (by `newInstance()`), which is frequently used in reflection code as shown in Section 2.3.6. In the past [17,

93, 52, 46, 21], many reflectively created objects have been ignored unsoundly. To handle reflective object creation more effectively (and soundly under certain assumptions), we exploit a key observation: usually, a reflectively created object is either used in a regular way or a reflective way. In the former case, the object has to be cast to a specific type first before its further usage as a normal object; in the later case, the object will flow to a reflective call site, e.g., `invoke()` (for reflective method invocations) or `get()/set()` (for reflective field accesses). As a result, SOLAR handles reflective object creation lazily by delaying the creation of objects at their usage points where their class types can be inferred.

In addition, we advocate (N2) in reflection analysis. Earlier, the inferences on individual reflective calls are mostly performed in isolation in order to resolve reflective targets [17, 93, 52, 21, 46]. Here, we emphasize that collective inference, where the inferences on related reflective calls mutually reinforce each other, is significant, due to the partial information available at each reflective call.

Only when (N3) is supported, will it become possible to reason about the soundness of a reflection analysis. When such reasoning is not possible for some programs due to, e.g., native code, SOLAR reduces to an effective under-approximate analysis due to its soundness-guided design. In addition, SOLAR is able to accurately identify where reflection is resolved unsoundly or imprecisely, helping users become aware of the effectiveness of their analysis (as explained in Section 1.2). Based on these identified "problematic" locations, SOLAR can further guide users to iteratively refine the results of their analysis by performing lightweight annotations until their specific requirements are satisfied.

Although there exist programs that we do not expect SOLAR to perform adequately on (e.g., SOLAR is unscalable for `jython` from `DaCapo`), we believe our results are representative of the potential for a considerable improvement over cur-

rent practice. In summary, this chapter makes 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.
- We have developed SOLAR by building on prior research and enhancing it by adopting the three novel aspects (N1) – (N3) in its design.
- We formalize SOLAR as part of a pointer analysis for Java (including the core of its reflection API) and reason about its soundness under some reasonable assumptions. This is the first such formalization for reflection analysis, which is developed to be extended easily to the entire Java reflection API and expected to be useful for future research on reflection analysis.
- We implement SOLAR in DOOP and make it public available at http://www.cse.unsw.edu.au/~corg/solar. To the best of our knowledge, SOLAR is the most comprehensive static analysis tool in handling the Java reflection API. Presently, SOLAR (version 0.1) can also output its reflection analysis results with the format that is supported by SOOT [92], which enables SOOT's clients to use its results directly.
- We evaluate SOLAR against two state-of-the-art reflection analyses, DOOP [17] and ELF [46], with 11 large Java benchmarks/applications, 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. These lightweight annotations are based on the 14 "problematic" program locations, which are automatically identified by SOLAR with no false positives; otherwise, more than 338 annotations are required using the previous approach to achieve the same-level soundness.

The rest of this chapter is organized as follows. Section 4.2 describes the challenges and our insights to achieve the goals of SOLAR. Then we present the methodology of SOLAR in Section 4.3 and formalize SOLAR in Section 4.4. The implementation of SOLAR, including what Java reflection API is handled and how to extend the rules in Section 4.4 to analyze other reflective methods, is introduced in Section 4.5. Then we evaluate SOLAR against the reflection analysis implementation in DOOP and ELF in Section 4.6. Finally, we briefly discuss the advantages and limitations of SOLAR in Section 4.7.

4.2 Challenges and Insights

The main goal of SOLAR is to reason about its soundness under reasonable assumptions and accurately identify the reflective calls in the program which are resolved unsoundly.

We argue that achieving this goal is very challenging because, theoretically, if one reflective call is handled unsoundly, e.g., one true reflective target is not resolved by the analysis, all the code in the program would be potentially unsound. For example, if one true target method (called at an `invoke()` call site) is not considered, we would not be able to identify what values at which points in the

program could be affected by this method. As a result, to draw a safe conclusion, we can only assume that the values at any points in the program may be under-approximated. This situation would deteriorate sharply if *many* reflective calls are resolved unsoundly as they may affect more values at more program points. Such unsoundness would be out of control when it is further spread out by the value flows in the program. As a result, the values at any program points are much more likely to be analyzed unsoundly. In this situation, it is extremely hard to reason about the soundness of reflection analysis. In addition, we cannot accurately identify the unsoundly resolved reflective calls in the program either, since all of them are likely to be handled unsoundly.

Our insights to achieve the goal of SOLAR are as follows.

- First, SOLAR needs to be significantly more sound than existing reflection analysis solutions. In other words, we have to ensure that only few (or no) reflective calls are resolved unsoundly. In this situation, as explained before, it would be significantly easier to control the unsoundness introduced by those unsoundly resolved reflective calls. As a result, if SOLAR reports that some analysis results are sound (e.g., a variable v only points to one heap object), they are probably sound (v will not point to another heap object at runtime). Similarly, if some analysis results are reported as unsound ones, they are likely unsound. This is the key to enable SOLAR to achieve practical precision in terms of both soundness reasoning and unsoundness identification.

- Second, to achieve the significantly more sound goal (as stated in the first point) or even full soundness (under some reasonable assumptions), SOLAR has to maximize its inference power (to infer reflective targets) by leveraging more available clues in the program in a more effective way. Meanwhile, SOLAR must infer reflective targets as precisely as possible; otherwise it would

probably be unscalable by introducing too many false reflective targets (e.g., introducing too many false call graph edges at `invoke()` and `newInstance()` call sites to make the call graph too large) and render itself useless.

- Third, SOLAR should be aware of the conditions under which a reflective target cannot be resolved. In other words, we need to design a set of soundness criteria for different Java reflective methods based on different inference strategies of SOLAR (relying on the inference principle as stated in the second point). If the criteria are not satisfied, SOLAR would be able to mark the corresponding reflective calls as the ones which are resolved unsoundly. On the contrary, if the criteria are satisfied, SOLAR would be able to determine the soundness of the reflection analysis.

4.3 Methodology

In this section, we first define precisely a set of assumptions made (Section 4.3.1). We then use an example to highlight how SOLAR differs fundamentally from the state-of-the-art (Section 4.3.2). Then, we examine its three new components: lazy heap modeling (Section 4.3.3), collective inference (Section 4.3.4) and automatic identification of "problematic" reflective calls (Section 4.3.5). Finally, we briefly show how SOLAR guides users to give lightweight annotations to improve the analysis quality (Section 4.3.6).

4.3.1 Assumptions

Besides the previous three common assumptions made on reflection analysis for Java in Section 3.2, SOLAR needs one more practical assumption to allow reflective allocation sites to be modeled lazily.

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 a side-effect method,* `get(v)`, `set(v,...)` *or* `invoke(v,...)`, *where* `v` *points to* `o`, *along every execution path in the program.*

Here, (1) and (2) represent two kinds of usage points at which the class types of object o will be inferred lazily. Specifically, case (1) indicates that o is used as a regular object, and case (2) says that o is used reflectively, i.e., flowing to the first argument of different side-effect calls as a receiver object. This object reachability assumption avoids the rare situation where o is created but never used later. As validated in Section 4.6.2, Assumption 4 is found to hold for almost all reflective allocation sites in real code.

4.3.2 A Motivating Example

Prior Work: Best-Effort Reflection Resolution Livshits et al. [52] introduced the first static reflection analysis for Java. This, together with subsequent extensions [17, 93, 21, 46], including ELF, resolves as much reflection as possible heuristically, without being able to identify reflective calls that potentially affect their soundness, precision and scalability. In particular, their under-approximate handling of `newInstance()` is a significant source of unsoundness. In the absence of reflection, a pointer analysis applies eager heap modeling to an allocation site `v = new A()`, by letting `v` point to an abstract object `o` of type `A` created at this site. However, adopting this eager approach in handling `newInstance()` is ineffective.

Consider Figure. 4.1, where `cName1` is an input string. The prior work cannot handle the `newInstance()` call in line 3 soundly. In [17, 93], line 10 is ignored since `cName1` is unknown. In [52, 53, 46], line 10 is also ignored: they failed to infer the types of the reflectively created objects in line 3 since the cast in line 12 is

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

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

9  void foo(B b, C c, … ) {
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[] {b, c});
16   }
17 }
```

Figure 4.1: An example of reflection usage (abstracted from real code) for comparing prior work and SOLAR.

not intra-procedurally post-dominating on these objects. In recent work [74], the objects created in line 10 (or line 3) are assumed to be of type `A` or its subtypes 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 the `else` branch are ignored. As a result, some target methods that are reflectively called on `v` at the `invoke()` call site in line 15 may not have been discovered.

SOLAR: Soundness-Guided Reflection Resolution Figure 4.2 illustrates the SOLAR design, with its three novel aspects marked by (N1) – (N3), illustrated by our example, and further described in Sections 4.3.3 – 4.3.5, respectively.

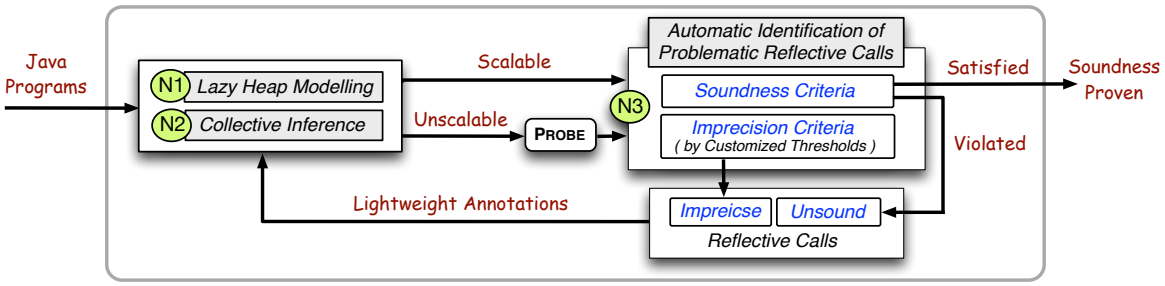


Figure 4.2: SOLAR: a soundness-guided analysis with three novel aspects (N1) – (N3).

SOLAR will model the `newInstance()` call in line 3 lazily due to (N1) as `cName1` is statically unknown, by returning an object o_3^u of an unknown type u. Note that o_3^u flows into two kinds of usage points: the cast operation in line 12 and the `invoke()` call in line 15. In the former case, SOLAR will infer u to be `A` and its subtypes in line 12. In the latter case, SOLAR will infer u based on the information available in line 15 by distinguishing three cases. (1) If `cName2` is known, then SOLAR deduces u from the known class in `cName2`. (2) If `cName2` is unknown but `mName2` is known, then SOLAR deduces u from the known method name in `mName2` and the second argument `new Object[] {b,c}` of the `invoke()` call site. As a result, SOLAR has successfully inferred the class type u in o_3^u due to a collective inference process (N2) emphasized for the first time in reflection analysis here. In SOLAR, inference is performed "collectively", whereby inferences on related reflective calls (in lines 3, 7 and 15) can mutually reinforce each other. (3) If both `cName2` and `mName2` are unknown (given that the types of o_3^u are already unknown), then SOLAR will flag the `invoke()` call in line 15 as being unsoundly resolved due to (N3), detected automatically by verifying one of the soundness criteria, i.e., Condition (4.4) in Section 4.4.5. 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. If SOLAR is unscalable, a simplified version of SOLAR, called PROBE, is enlisted to first identify reflective calls to be annotated so that SOLAR can then proceed scalably.

Discussion If Assumptions 1 – 4 hold, we can establish the **full soundness** of SOLAR by verifying the soundness criteria (stated in Section 4.4.5) at all `invoke()`, `get()` and `set()` reflective call sites in the program. Otherwise, SOLAR reduces to an effective under-approximate analysis (i.e., significantly more sound reflection analysis) due to (N1) – (N3) in its design.

In fact, according to our experience, **empirical soundness** can still be established in a closed-world, which is mainly because Assumptions 2 – 4 are the practical ones. In other words, regardless of the code outside the closed-world, if SOLAR reports that the reflection analysis is sound, it is probably sound and the identified unsoundly resolved reflective calls are probably unsoundly resolved. We argue that such **empirical soundness** is also useful in practice. In addition, it indirectly explains why SOLAR is able to achieve total recall and identify the unsoundly resolved reflective calls very accurately in an extended closed-world (or a simulated open-world) as shown in Section 4.6.

4.3.3 Lazy Heap Modeling

Our Insight Figure. 4.3 illustrates the basic idea behind our lazy heap modeling (LHM). Usually, an object, say `o`, created by `newInstance()` will be later used either regularly or reflectively as shown in Cases (II) and (III) respectively. In Case (II), since the declared type of `o` is `java.lang.Object`, before calling methods or accessing fields as a regular object, it has to be cast to a specific type first. As a result, `o` will flow to some cast operations in this case. In Case (III), `o` is used in a reflective way, i.e., as the first argument of a call to a side-effect method, `invoke()`, `get()` or `set()`, on which the target method (field) is called (accessed). According to our experience, this is especially commonly used in Android apps. For these two cases, we can leverage the information at `o`'s usage sites to infer its type lazily and also make the corresponding effects visible there. As for the side-effects that may be made by `o` during the paths from `newInstance()` call site to its usages sites, we use Case (I) to cover this situation. Now, let us examine these three cases, which are highlighted in Figure 4.3, one by one.

If `cName` at `c = Class.forName(cName)` is unknown, SOLAR will create a `Class`

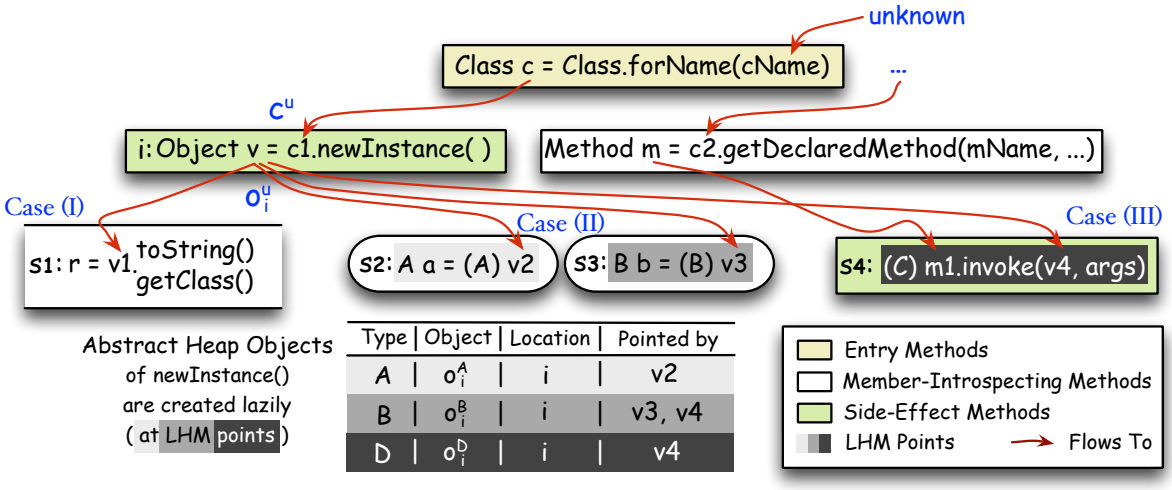


Figure 4.3: Overview of LHM. Suppose `A` and `B` have no subtypes. Suppose further that `m1` is declared in class `D` with one subtype `B`, implying that the dynamic types of the objects pointed by `v4` is `D` or `B`. The three abstract objects, o^A_i, o^B_i and o^D_i, for `newInstance()` are created lazily at the two kinds of LHM (usage) points in Cases (II) and (III).

object c^u that represents this unknown class and assign it to `c`. On discovering that `c1` points to a c^u at an allocation site `i`, `v = c1.newInstance()`, SOLAR will create an abstract object o^u_i of an unknown type for the site to mark it as unresolved yet. Subsequently, o^u_i will flow into Cases (I) – (III).

In Case (I), since the returned type of o^u_i is declared as `java.lang.Object`, then before flowing to a cast operation, the only side-effect that could be made by this object is to call some methods declared in class `java.lang.Object`. In terms of reflection analysis, only the two shown pointer-affecting methods need to be considered. SOLAR handles them soundly, by returning (1) an unknown string for `v1.toString()` and (2) an unknown `Class` object for `v1.getClass()`. Note that `clone()` cannot be called on `v1` of type `java.lang.Object` (without a downcast on `v1` first).

Let us consider Cases (II) and (III), where each statement, say S_x, is called an

LHM point, containing a variable x to which $\texttt{o}_\texttt{i}^u$ flows. In Figure 4.3, $x \in \{\texttt{v2}, \texttt{v3}, \texttt{v4}\}$. Let $lhm(S_x)$ be the set of class types discovered for u at S_x by inferring from the cast operation at S_x as in Case (II) or the information available at a call to `(C) m1.invoke(v4, args)` (e.g., on `C`, `m1` and `args`) as in Case (III). For example, given $\texttt{S2}_\texttt{v2}$: `A a = (A) v2`, $lhm(\texttt{S2}_\texttt{v2})$ contains `A` and its subtypes. To account for the effect of `v = c1.newInstance()` at S_x lazily, we add (conceptually) a statement, `x = new T()`, for every $T \in lhm(S_x)$, before S_x. Thus, $\texttt{o}_\texttt{i}^u$ is finally split into and thus aliased with n distinct abstract objects, $\texttt{o}_\texttt{i}^{T_1}, \ldots, \texttt{o}_\texttt{i}^{T_n}$, where $lhm(S_x) = \{T_1, \ldots, T_n\}$, such that x points to all these new abstract objects.

Figure 4.3 illustrates lazy heap modeling for the case when neither `A` nor `B` has subtypes and the declaring class for `m1` is discovered to be `D`, which has one subtype `B`. Thus, SOLAR will deduce that $lhm(\texttt{S2}_\texttt{v2}) = \{\texttt{A}\}$, $lhm(\texttt{S3}_\texttt{v3}) = \{\texttt{B}\}$ and $lhm(\texttt{S4}_\texttt{v4}) = \{\texttt{B,D}\}$. Hence, the lazy modeling results as shown. Note that in Case (II), $\texttt{o}_\texttt{i}^u$ will not flow to `a` and `b` due to the cast operations.

As `java.lang.Object` contains no fields, all field accesses to $\texttt{o}_\texttt{i}^u$ will only be made on its lazily created objects. Therefore, if the same concrete object represented by $\texttt{o}_\texttt{i}^u$ flows to both S_{x_1} and S_{x_2}, then $lhm(S_{x_1}) \cap lhm(S_{x_2}) \neq \varnothing$. This implies that x_1 and x_2 will point to a common object lazily created. For example, in Figure 4.3, `v3` and `v4` points to $o_\texttt{i}^\texttt{B}$ since $lhm(S3_\texttt{v3}) \cap lhm(S4_\texttt{v4}) = \{\texttt{o}_\texttt{i}^\texttt{B}\}$. Thus, the alias between $x_1.f$ and $x_2.f$ is correctly maintained, where f is a field of $\texttt{o}_\texttt{i}^u$.

Discussion Under Assumption 4, only the three cases in Figure 4.3 need to handle to establish the soundness for modeling `newInstance()` lazily. The rare exception is that $\texttt{o}_\texttt{i}^u$ is created but never used later (where no hints are available). Then to achieve soundness under this exception, the corresponding constructor (of the dynamic type of $\texttt{o}_\texttt{i}^u$) must be annotated to be analyzed statically unless ignoring it will not affect the points-to information to be obtained. Again, as validated in

```
Class: java.awt.EventDispatchThread
private boolean handleException (Throwable thrown) {
   ...
   Class c = Class.forName(hd);        // hd unresolved
   Method m = c.getMethod("handle", …);
   Object h = c.newInstance();
   m.invoke(h, new Object[] {thrown});  }
```

Figure 4.4: Collective inference, by which the dynamic types of `h` are inferred from the reflective targets inferred at `invoke()`.

Section 4.6.2, Assumption 4 is found to be very practical.

4.3.4 Collective Inference

SOLAR builds on the prior work [46, 17, 52] by emphasizing on collective inference (on related reflective calls), enabled with new inference rules added.

Let us illustrate its importance with an example in `JDK 1.6.0_45` in Figure. 4.4, where `hd` is an unknown string. Due to the lack of a post-dominant cast for `newInstance()`, the call cannot be resolved in the past. In SOLAR, we will infer the declaring classes of `m` based on the information deduced from the second argument of `invoke()` and the name "`handle`" of the target method (see Example 2 in Section 4.4.4.3 for detail). This also enables the dynamic types of the objects pointed to by `h` to be inferred. Thus, LHM for the `newInstance()` call can be performed. Otherwise, `chart` and `fop` from DaCapo cannot be analyzed effectively. The converse of this inference is equally important, as demonstrated later with a real application in Section 4.6.4.3. To the best of our knowledge, SOLAR is the first to conduct such collective inference by exploiting the connection between `newInstance()` (via LHM) and other side-effect methods, which is particularly useful for the case where a reflectively created object is further used reflectively.

4.3.5 Automatic Identification of "Problematic" Reflective Calls

Intuitively, we mark a reflective call as unsoundly resolved, if and only if all the inference strategies of SOLAR (e.g., LHM and collective inference, etc.) have been tried but fail to work due to some unavailable hints. Let us take the example in Figure 4.3 to illustrate how SOLAR determines whether a `newInstance()` call is resolved unsoundly. All the previous settings for this example remain the same except that the declared type and the name of `m1` at site `s4` are unknown. Note that simply leveraging `args` to infer the types may cause the analysis to be too imprecise to scale. As a result, SOLAR fails to apply collective inference to resolve the type of $\mathtt{o}_i^u$ pointed to by `v4`. Accordingly, Case (III) in LHM cannot be handled soundly and SOLAR will mark the `newInstance()` at site `i` as an unsoundly resolved reflective call. The formal soundness criteria used to identify unsoundly resolved reflective calls, are defined and illustrated in Section 4.4.5.

As for identifying imprecisely resolved reflective calls, SOLAR currently adopts a simple but practical approach: measuring the precision according to the number of the targets resolved at these call sites. SOLAR allows users to specify a threshold value in advance to define the imprecision that can be tolerated.

Like every other reflection analysis, SOLAR may handle a program scalably or unscalably (under a budget). However, there are some fundamental differences. If SOLAR is scalable, SOLAR can automatically identify problematic reflective calls (as described above) that may threaten its soundness and precision, together with their corresponding annotation points, to enable both to be improved with lightweight annotations. If SOLAR is unscalable for a program, we address this scalability issue by analyzing the same program again with a simplified version of SOLAR, denoted PROBE in Figure 4.2. Since PROBE resolves reflection very precisely (by sacrificing

some soundness), its scalability could usually be guaranteed. In addition, PROBE is able to identify the problematic reflective calls using the same approach as SOLAR. Thus with some problematic reflective calls (identified by PROBE) 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 trade-offs among soundness, precision and scalability, so that the scalability of PROBE could be always guaranteed.

4.3.6 Guided Lightweight Annotations

After having identified the problematic reflective calls, SOLAR can guide the users to the program points where hints for annotations are potentially available. As the problematic reflective call sites are the places where side-effect methods are invoked, we can hardly extract the information there to know the target names as they are specified at the corresponding entry and member-introspecting call sites. Here, we call the latter two kinds of call sites, the annotation sites. Since the three kinds of reflective call sites may not be within the same method (due to some wrapper methods used [52]), given an identified problematic side-effect call site, it is hard for users to find out the corresponding annotation sites manually. Thus, SOLAR is designed to automatically track the metaobject flows from a given side-effect call site in a demand-driven way to find out all the related annotation sites.

As the guided-annotation mechanism is based on the accurately identified problematic (e.g., unsound) locations, the number of the required annotations (for soundness) would be significantly less than the one achieved by using the method in [52], which simply asks for annotations when an input string is unknown. This is further validated in Section 4.6.3.

4.4 Formalism

We formalize SOLAR, as illustrated in Figure 4.2, for REFJAVA, which is Java restricted to a core subset of its reflection API. SOLAR is flow-insensitive but context-sensitive. However, our formalization is context-insensitive. We first define REFJAVA (Section 4.4.1), give a road map for the formalism (Section 4.4.2) and present some notations used (Section 4.4.3). We then introduce a set of rules for formulating lazy heap modeling and collective inference (Section 4.4.4). Based on these rules, we formulate a set of soundness criteria (Section 4.4.5) that enables reasoning about the soundness of SOLAR (Section 4.4.6). Finally, we describe how to instantiate PROBE from SOLAR (Section 4.4.7), and handle static class members (Section 4.4.8).

4.4.1 The REFJAVA Language

REFJAVA consists of all Java programs (under Assumptions 1 – 4) except that the Java reflection API is restricted to the seven methods in Figure 3.1. Our formalism is designed to allow its straightforward generalization to the entire Java reflection API. As is standard, a Java program is represented only by five kinds of statements in the SSA form, as shown in Figure 4.7. For simplicity, we assume that all the members (fields or methods) of a class accessed reflectively are its instance members, i.e., `o` $\neq$ `null` in `get(o)`, `set(o,a)` and `invoke(o,...)` in Figure 3.1. We will discuss how to handle static members in Section 4.4.8.

4.4.2 Road Map

As depicted in Figure 4.5, SOLAR's inference system, which consists of five components, works together with a pointer analysis. The arrow $\longleftrightarrow$ between a component

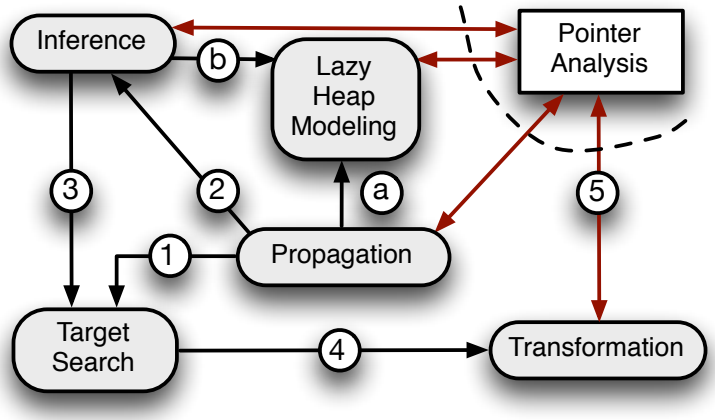


Figure 4.5: SOLAR's inference system: five components and their dependency relationships (depicted by arrows) with pointer analysis.

and the pointer analysis indicates that each is both a producer and consumer of the other.

Let us see how SOLAR resolves `r = `$\mathcal{F}$`.get(o)` in Figure 3.1. If `cName` and `fName` are string constants, *Propagation* will create a `Field` object (pointed to by $\mathcal{F}$) carrying its known class and field information and pass it to *Target Search* (①). If `cName` or `fName` is not a constant, a `Field` object marked as such is created and passed to *Inference* (②), which will infer the missing information and pass a freshly generated `Field` object enriched with the missing information to *Target Search* (③). Then *Target Search* maps a `Field` object to its reflective target f in its declaring class (④). Finally, *Transformation* turns the reflective call `r = `$\mathcal{F}$`.get(o)` into a regular statement `r = o.`f and pass it to the pointer analysis (⑤). *Lazy Heap Modeling* handles `newInstance()` based on the information discovered by *Propagation* (ⓐ) or *Inference* (ⓑ).

4.4.3 Notations

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

class type	$t \in \mathbb{T}$
`Field` object*	$\mathtt{f}_s^t, \mathtt{f}_u^t, \mathtt{f}_s^u, \mathtt{f}_u^u \in \mathbb{FO} = \widehat{\mathbb{T}} \times \mathbb{S}_f$
field/method name	$n_f, n_m \in \mathbb{N}$
field signature*	$s \in \mathbb{S}_f = \widehat{\mathbb{T}} \times \widehat{\mathbb{N}}$
field	$f \in \mathbb{F} = \mathbb{T} \times \mathbb{T} \times \mathbb{N}$
field type*	$s.t_f \in \widehat{\mathbb{T}}$
parameter (types)	$p \in \mathbb{P} = \mathbb{T}^0 \cup \mathbb{T}^1 \cup \mathbb{T}^2 \ldots$
field name*	$s.n_f \in \widehat{\mathbb{N}}$
method	$m \in \mathbb{M} = \mathbb{T} \times \mathbb{T} \times \mathbb{N} \times \mathbb{P}$
`Method` object*	$\mathtt{m}_s^t, \mathtt{m}_u^t, \mathtt{m}_s^u, \mathtt{m}_u^u \in \mathbb{MO} = \widehat{\mathbb{T}} \times \mathbb{S}_m$
local variable	$\mathrm{c}, \mathrm{f}, \mathrm{m} \in \mathbb{V}$
method signature*	$s \in \mathbb{S}_m = \widehat{\mathbb{T}} \times \widehat{\mathbb{N}} \times \widehat{\mathbb{P}}$
Abstract heap object	$o_1^t, o_2^t, \ldots, o_1^u, o_2^u, \cdots \in \mathbb{H}$
return type*	$s.t_r \in \widehat{\mathbb{T}}$
unknown	u
method name*	$s.n_m \in \widehat{\mathbb{N}}$
`Class` object	$\mathtt{c}^t, \mathtt{c}^u \in \mathbb{CO}$
parameter*	$s.p \in \widehat{\mathbb{P}}$

Figure 4.6: Notations ($\widehat{X} = X \cup \{u\}$, where u is an unknown class type or an unknown field/method signature). A superscript '*' marks a domain that contains u.

We will use the notations given in Figure 4.6. $\mathbb{CO}$, $\mathbb{FO}$ and $\mathbb{MO}$ represent the set of `Class`, `Field` and `Method` objects, respectively. In particular, $\mathtt{c}^t$ denotes a `Class` object of a known class t and $\mathtt{c}^u$ a `Class` object of an unknown class u. As illustrated in Figure 4.3, 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 `Field` object, we write $\mathtt{f}_s^t$ if it is a field defined in a known

class t and $\texttt{f}_s^u$ otherwise, with its signature being s. In particular, we write $\texttt{f}_u^-$ for $\texttt{f}_s^-$ in the special case when s is unknown, i.e., $s.t_f = s.n_f = u$. Similarly, $\texttt{m}_s^t$, $\texttt{m}_u^t$, $\texttt{m}_s^u$ and $\texttt{m}_u^u$ are used to represent `Method` objects. We write $\texttt{m}_u^-$ for $\texttt{m}_s^-$ when s is unknown (with the return type $s.t_r$ being ignored), i.e., $s.n_m = s.p = u$.

4.4.4 The SOLAR's Inference System

We present the inference rules used by all the components in Figure 4.5, starting with the pointer analysis and moving to the five components of SOLAR. Due to their cyclic dependencies, the reader is invited to read ahead sometimes, particularly to Section 4.4.4.6 on LHM, before returning back to the current topic.

4.4.4.1 Pointer Analysis

Figure 4.7 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], only the fields of an abstract object o_i^t of a known type t can be accessed. In Java, as explained in Section 4.3.3, the field accesses to o_i^u (of an unknown type) can only be made to the abstract objects of known types created lazily from o_i^u at LHM points.

In [A-CALL] (for non-reflective calls), like the one presented in [81], the function $dispatch(o_i^-, m)$ is used to resolve the virtual dispatch of method m on the receiver object o_i^- to be m'. There are two cases. If $o_i^t \in pt(y)$, we proceed normally as before. For $o_i^u \in pt(y)$, it suffices to restrict m to {`toString()`, `getClass()`}, as explained in Section 4.3.3. We assume that m' has a formal parameter m'_{this} for

$$\frac{i:\ \mathrm{x} = new\ t()}{\{o_i^t\} \in pt(\mathrm{x})}\ \text{[A-New]} \qquad \frac{\mathrm{x} = \mathrm{y}}{pt(\mathrm{y}) \subseteq pt(\mathrm{x})}\ \text{[A-Cpy]}$$

$$\frac{\mathrm{x} = \mathrm{y.f}\ \ o_i^t \in pt(\mathrm{y})}{pt(o_i^t.\mathrm{f}) \subseteq pt(\mathrm{x})}\ \text{[A-Ld]} \qquad \frac{\mathrm{x.f} = \mathrm{y}\ \ o_i^t \in pt(\mathrm{x})}{pt(\mathrm{y}) \subseteq pt(o_i^t.\mathrm{f})}\ \text{[A-St]}$$

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

Figure 4.7: Rules for *Pointer Analysis*.

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'.

4.4.4.2 Propagation

Figure 4.8 gives the rules for handling `forName()`, `getMethod()` and `getField()` calls. Different kinds of `Class`, `Method` and `Field` objects are created depending on whether their string arguments are string constants or not. For these rules, $\mathbb{SC}$ denotes a set of string constants and the function *toClass* creates a `Class` object 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, $\mathtt{c}^t$, $\mathtt{f}_s^t$ and $\mathtt{m}_s^t$ will flow to *Target Search* but all the others, i.e., $\mathtt{c}^u$, $\mathtt{f}_-^u$, $\mathtt{f}_u^-$, $\mathtt{m}_-^u$ and $\mathtt{m}_u^-$ will flow to *Inference*, where the missing information is inferred. During *Propagation*, only the name of a method/field signature s ($s.n_m$ or $s.n_f$) can be discovered but its other parts are unknown: $s.t_r = s.p = s.t_f = u$.

4.4.4.3 Inference

Figure 4.9 gives nine rules to infer reflective targets at `x = (A) m.invoke(y,args)`, `x = (A) f.get(y)`, `f.set(y,x)`, where `A` indicates a post-dominating cast on their

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

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

$$\frac{\mathit{Field}\ \mathrm{f} = \mathrm{c}'.\mathit{getField}(\mathrm{fName}) \quad o_i^{\texttt{String}} \in pt(\mathrm{fName}) \quad \mathtt{c}^- \in pt(\mathrm{c}')}{pt(\mathrm{f}) \supseteq \begin{cases} \{\mathtt{f}_s^t\} & \text{if } \mathtt{c}^- = \mathtt{c}^t \wedge o_i^{\texttt{String}} \in \mathbb{SC} \\ \{\mathtt{f}_u^t\} & \text{if } \mathtt{c}^- = \mathtt{c}^t \wedge o_i^{\texttt{String}} \notin \mathbb{SC} \\ \{\mathtt{f}_s^u\} & \text{if } \mathtt{c}^- = \mathtt{c}^u \wedge o_i^{\texttt{String}} \in \mathbb{SC} \\ \{\mathtt{f}_u^u\} & \text{if } \mathtt{c}^- = \mathtt{c}^u \wedge o_i^{\texttt{String}} \notin \mathbb{SC} \end{cases} \quad \begin{array}{c} s.t_f = u \\ s.n_f = \mathit{val}(o_i^{\texttt{String}}) \end{array}} \text{[P-GetFld]}$$

Figure 4.8: Rules for *Propagation*.

results. If `A = Object`, no such cast exists. These rules fall into three categories. In [I-InvTp], [I-GetTp] and [I-SetTp], we use the types of the objects pointed to by `y` to infer the class type of a method/field. In [I-InvSig], [I-GetSig] and [I-SetSig], we use the information available at a call site (excluding `y`) to infer the descriptor of a method/field signature. In [I-InvS2T], [I-GetS2T] and [I-SetS2T], we use a method/field signature to infer the class type of a method/field.

Some notations used are in order. As is standard, $t <: t'$ holds when t is t' or a subtype of t'. In [I-InvSig], [I-GetSig], [I-InvS2T] and [I-GetS2T], $\ll:$ is used to take advantage of the post-dominating cast `(A)` during inference when `A` is not `Object`. By definition, $u \ll: \texttt{Object}$ holds. If t' is not `Object`, then $t \ll: t'$ holds if and only if $t <: t'$ or $t' <: t$ holds. In [I-InvSig] and [I-InvS2T], the information on `args` is also exploited, where `args` is an array of type `Object[]`, only when it can be analyzed exactly element-wise by an intra-procedural analysis. In this case,

x = (A) m.*invoke*(y, args)

$$\frac{\mathtt{m}^{u}_{\_} \in pt(\mathrm{m})}{pt(\mathrm{m}) \supseteq \{\, \mathtt{m}^{t}_{\_} \mid o^{t}_{i} \in pt(\mathrm{y})\}} \text{ [I-InvTp]}$$

$$\frac{\mathtt{m}^{-}_{u} \in pt(\mathrm{m})}{pt(\mathrm{m}) \supseteq \{\, \mathtt{m}^{-}_{s} \mid s.p \in Ptp(\mathrm{args}),\ s.t_r \ll: \mathsf{A},\ s.n_m = u\}} \text{ [I-InvSig]}$$

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

x = (A) f.*get*(y)

$$\frac{\mathtt{f}^{u}_{\_} \in pt(\mathrm{f})}{pt(\mathrm{f}) \supseteq \{\, \mathtt{f}^{t}_{\_} \mid o^{t}_{i} \in pt(\mathrm{y})\}} \text{ [I-GetTp]}$$

$$\frac{\mathtt{f}^{-}_{u} \in pt(\mathrm{f})}{pt(\mathrm{f}) \supseteq \{\, \mathtt{f}^{-}_{s} \mid s.t_f \ll: \mathsf{A},\ s.n_f = u\}} \text{ [I-GetSig]}$$

$$\frac{\mathtt{f}^{u}_{s} \in pt(\mathrm{f}) \quad o^{u}_{i} \in pt(\mathrm{y}) \quad s.n_f \neq u \quad s.t_f \ll: \mathsf{A}}{pt(\mathrm{f}) \supseteq \{\mathtt{f}^{t}_{s} \mid t \in \mathcal{F}(s.n_f, s.t_f)\}} \text{ [I-GetS2T]}$$

f.*set*(y, x)

$$\frac{\mathtt{f}^{u}_{\_} \in pt(\mathrm{f})}{pt(\mathrm{f}) \supseteq \{\, \mathtt{f}^{t}_{\_} \mid o^{t}_{i} \in pt(\mathrm{y})\}} \text{ [I-SetTp]}$$

$$\frac{\mathtt{f}^{-}_{u} \in pt(\mathrm{f})}{pt(\mathrm{f}) \supseteq \{\, \mathtt{f}^{-}_{s} \mid o^{t}_{j} \in pt(\mathrm{x}),\ \mathtt{t} <: s.t_f, s.n_f = u\}} \text{ [I-SetSig]}$$

$$\frac{\mathtt{f}^{u}_{s} \in pt(\mathrm{f}) \quad o^{u}_{i} \in pt(\mathrm{y}) \quad s.n_f \neq u \quad o^{t'}_{j} \in pt(\mathrm{x}) \quad t' <: s.t_f}{pt(\mathrm{f}) \supseteq \{\mathtt{f}^{t}_{s} \mid t \in \mathcal{F}(s.n_f, s.t_f)\}} \text{ [I-SetS2T]}$$

Figure 4.9: Rules for *Inference*.

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 $Ptp(\mathtt{args}) = P_0 \times \cdots \times P_{n-1}$. Otherwise, $Ptp(\mathtt{args}) = \varnothing$, implying that `args` is ignored as it cannot be exploited effectively during inference.

To maintain precision in [I-InvS2T], [I-GetS2T] and [I-SetS2T], we use a method

(field) signature to infer its classes when both its name and descriptor are known. In [I-InvS2T], the function $\mathcal{M}(s_{t_r}, s.n_m, s.p)$ returns the set of classes where the method with the specified signature s is defined if $s.n_m \neq u$ and $s.p \neq u$, and $\varnothing$ otherwise. The return type of the matching method is ignored if $s.t_r = u$. In [I-GetS2T] and [I-SetS2T], $\mathcal{F}(s.n_f, s.t_f)$ returns the set of classes where the field with the given signature s is defined if $s.n_f \neq u$ and $s.t_f \neq u$, and $\varnothing$ otherwise.

Let us illustrate our rules by considering two examples in Figures 4.1 and 4.4.

Example 1 *Let us modify the reflective allocation site in line 3 to* `c1.newInstance()`*, where* `c1` *represents a known class, named* `A`*, so that* $\texttt{c1}^{\texttt{A}} \in pt(\texttt{c1})$*. By applying [L-KwTp] (introduced later in Figure 4.11) to the modified allocation site,* Solar *will create a new object* $o_3^{\texttt{A}}$*, which will flow to line 10 so that* $o_3^{\texttt{A}} \in pt(\texttt{v})$*. Suppose both* `cName2` *and* `mName2` *point to some unknown strings. When [P-GetMtd] is applied to* `c.getMethod(mName,...)` *in line 7, a* `Method` *object, say,* m_u^u *is created and eventually assigned to* `m` *in line 14. By applying [I-InvTp] to* `m.invoke(v, args)` *in line 15, where* $o_3^{\texttt{A}} \in pt(v)$*,* Solar *deduces that the target method is a member of class* `A`*. Thus, a new object* $m_u^{\texttt{A}}$ *is created and assigned to* $pt(\texttt{m})$*. Given* `args = new Object[] {b, c}`*,* $Ptp(args)$ *is constructed as described earlier. By applying [I-InvSig] to this* `invoke()` *call,* Solar *will add all new* `Method` *objects* $m_s^{\texttt{A}}$ *to* $pt(\texttt{m})$ *such that* $s.p \in Ptp(args)$*, which represent the potential target methods called reflectively at this site.* □

Example 2 *Here,* `hd` *is statically unknown but the string argument of* `getMethod()` *is* `"handle"`*. By applying [P-ForName], [P-GetMtd] and [L-UkwTp] (Figure 4.11) to the* `forName()`*,* `getMethod()` *and* `newInstance()` *calls, respectively, we obtain* $\texttt{c}^u \in pt(\texttt{c})$*,* $m_s^u \in pt(\texttt{m})$ *and* $o_i^u \in pt(\texttt{h})$*, where* s *indicates a signature with a known method name (i.e., "*`handle`*"). Since the second argument of the* `invoke()` *call can also be exactly analyzed,* Solar *will be able to infer the classes* t *where* `"handle"` *is*

defined by applying /I-InvS2T/. *Finally,* Solar *will add all inferred* `Method` *objects* $\mathtt{m}_s^t$ *to* $pt(\mathtt{m})$ *at the call site. Since neither the superscript nor the subscript of* $\mathtt{m}_s^t$ *is* u, *the inference is done and it will be used to find out the targets (represented by it) in* Target Search *(Section 4.4.4.4). Note that* Solar *does not search the targets directly (based on* $\mathtt{m}_s^t$*) and immediately add the resolved target methods in the call graph at this call site. Instead, it divides such operation into different phases: (1) adding* $\mathtt{m}_s^t$ *into* $pt(\mathtt{m})$*, (2) searching the targets (based on the* $\mathtt{m}_s^t$ *in (1)) in* Target Search*, and (3) transforming the reflective call into the corresponding regular call (using the targets found in (2)) in* Transformation. *Such design is for easy extension of the rules. For example, if considering another member-introspecting method* `getDeclaredMethod()`*, we have to rewrite the three rules (about* `invoke()`*) for it redundantly. Our current design can avoid such modification. To achieve the same result, we only need to tune the* MTD *function (defined in Section 4.4.4.4) as described in Section 4.5.* □

4.4.4.4 Target Search

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

$$MTD(\mathtt{m}_s^t) \quad = \quad \bigcup_{t <: t'} mtdLookUp(t', s.t_r, s.n_m, s.p) \tag{4.1}$$

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 in the search (for better precision) and (2) any u that appears in s is treated as a wild card during the search. Similarly, we define $FLD : \mathbb{FO} \rightarrow \mathcal{P}(\mathbb{F})$

for a `Field` object $\texttt{f}_s^t$:

$$FLD(\texttt{f}_s^t) \quad = \quad \bigcup_{t<:t'} fldLookUp(t', s.t_f, s.n_f) \tag{4.2}$$

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

4.4.4.5 Transformation

Figure 4.10 gives the rules used for transforming a reflective call into a regular statement, which will be analyzed by the pointer analysis.

$$\frac{\text{x} = \text{m}.invoke(\text{y, args}) \quad \texttt{m}_\_^t \in pt(\text{m}) \quad m' \in MTD(\texttt{m}_\_^t) \quad o_i^- \in pt(\text{args}) \quad o_j^{t'} \in pt(o_i^-.arr) \quad t'' \text{ is declaring type of } m'_{pk} \quad t' <: t''}{\forall\ 1 \leqslant k \leqslant n : \{o_j^{t'}\} \subseteq pt(\text{arg}_k) \quad \text{x} = \text{y}.m'(\text{arg}_1, ..., \text{arg}_n)} \text{ [T-Inv]}$$

$$\frac{\text{x} = \text{f}.get(\text{y}) \quad \texttt{f}_\_^t \in pt(\text{f}) \quad f \in FLD(\texttt{f}_\_^t)}{\text{x} = \text{y}.f} \text{ [T-Get]}$$

$$\frac{\text{f}.set(\text{y, x}) \quad \texttt{f}_\_^t \in pt(\text{f}) \quad f \in FLD(\texttt{f}_\_^t)}{\text{y}.f = \text{x}} \text{ [T-Set]}$$

Figure 4.10: 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 intra-procedurally in our current implementation. Let $\texttt{arg}_1, \ldots, \texttt{arg}_n$ 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 $pt(\texttt{arg}_k)$ only when $t' <: t''$ holds in order to filter out the objects that cannot be assigned to m'_{pk}. Finally, the reflective target method found can be analyzed by [A-Call] in Figure 4.7.

4.4.4.6 Lazy Heap Modeling

In Figure 4.11, we give all the rules for resolving a `newInstance()` call lazily, as explained in Section 4.3.3.

$$\frac{i : \mathrm{o} = \mathrm{c}'.newInstance() \quad \mathtt{c}^t \in pt(\mathrm{c}')}{\{o_i^t\} \subseteq pt(\mathrm{o})} \text{ [L-KwTp]}$$

$$\frac{i : \mathrm{o} = \mathrm{c}'.newInstance() \quad \mathtt{c}^u \in pt(\mathrm{c}')}{\{o_i^u\} \subseteq pt(\mathrm{o})} \text{ [L-UkwTp]}$$

$$\frac{\mathsf{A}\ a = (\mathsf{A})\,\mathrm{x} \quad o_i^u \in pt(\mathrm{x}) \quad t <: \mathsf{A}}{\{o_i^t\} \subseteq pt(a)} \text{ [L-Cast]}$$

$$\frac{\mathrm{x} = \mathrm{m}.invoke(\mathrm{y}, ...) \quad o_i^u \in pt(\mathrm{y}) \quad \mathtt{m}^t_{\_} \in pt(\mathrm{m}) \quad t' \ll: t}{\{o_i^{t'}\} \subseteq pt(\mathrm{y})} \text{ [L-Inv]}$$

$$\frac{\mathrm{x} = \mathrm{f}.get(\mathrm{y}) \;/\; \mathrm{f}.set(\mathrm{y}, \mathrm{x}) \quad o_i^u \in pt(\mathrm{y}) \quad \mathtt{f}^t_{\_} \in pt(\mathrm{f}) \quad t' \ll: t}{\{o_i^{t'}\} \subseteq pt(\mathrm{y})} \text{ [L-GSet]}$$

Figure 4.11: Rules for *Lazy Heap Modeling*.

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 in Figure 4.3, 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 its Case (II) by applying [L-Cast] (with o_i^u blocked from flowing from `x` to `a`) and its Case (III) by applying [L-Inv] and [L-GSet]. Note that in [L-Cast], `A` is assumed not to be `Object`.

4.4.5 Soundness Criteria

RefJava consists of four side-effect methods as shown in Figure 3.1. Solar is sound if their calls are resolved soundly under Assumptions 1 – 4. Due to Assumption 4 illustrated in Figure 4.3, there is no need to consider `newInstance()` since it is soundly resolved if `invoke()`, `get()` and `set()` are. For convenience, we define:

$$AllKwn(v) \quad = \quad \nexists\, o_i^u \in pt(v) \tag{4.3}$$

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

Figure 4.9 gives a set of nine rules for `(A) m.invoke(y, args)`, `(A) f.get(y)` and `f.set(y, x)`. For the `Method` (`Field`) objects $\mathtt{m}_s^t$ ($\mathtt{f}_s^t$) 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], [I-GetSig] and [I-SetSig].

For the `Method` (`Field`) objects $\mathtt{m}_s^u$ ($\mathtt{f}_s^u$) with unknown class types u, the targets accessed are inferred by applying the remaining six rules in Figure 4.9. Let us consider a call to `(A) m.invoke(y, args)`. Solar attempts to infer the missing classes of its `Method` objects in two ways, by applying [I-InvTp] and [I-InvS2T]. Such a call is soundly resolved if the following condition holds:

$$SC(\mathtt{m.invoke(y,args)}) = AllKwn(\mathtt{y}) \vee\ \forall\, \mathtt{m}_s^u \in pt(\mathtt{m}) : s.n_m \neq u \wedge Ptp(\mathtt{args}) \neq \varnothing \tag{4.4}$$

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` object $\mathtt{m}_{\_}^u \in pt(m)$ 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` object $\mathtt{m}_s^u \in pt(m)$ 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).

Finally, the soundness criteria for `get()` and `set()` are derived similarly:

$$SC(\mathtt{(A)\,f.get(y)}) = AllKwn(\mathtt{y}) \ \vee \ \forall\, \mathtt{f}_s^u \in pt(\mathtt{f}) : s.n_f \neq u \wedge \mathtt{A} \neq \mathtt{Object} \quad (4.5)$$

$$SC(\mathtt{f.set(y,x)}) = AllKwn(\mathtt{y}) \ \vee \ \forall\, \mathtt{f}_s^u \in pt(\mathtt{f}) : s.n_f \neq u \wedge AllKwn(\mathtt{x}) \quad (4.6)$$

In (4.5), applying [I-GetTp] ([I-GetS2T]) resolves a `get()` call soundly if its first (second) disjunct holds. In (4.6), applying [I-SetTp] ([I-SetS2T]) resolves a `set()` call soundly if its first (second) disjunct holds. By observing [T-Set], we see why $AllKwn(\mathtt{x})$ is needed to reason about the soundness of [I-SetS2T].

4.4.6 Soundness Proof

We prove the soundness of Solar for RefJava subject to our soundness criteria (4.4) – (4.6) under Assumptions 1 – 4. We do so by taking advantage of the well-established soundness of Andersen's pointer analysis (Figure 4.7) stated below.

Lemma 1 Solar *is sound for* RefJava *with its reflection API ignored.*

If we know the class types of all targets accessed at a reflective call but possibly nothing about their signatures, Solar can over-approximate its target method/-fields in *Target Search*. Hence, the following lemma holds.

Lemma 2 Solar *is sound for* RefJava$_c$*, the set of all* RefJava *programs in which* `cName` *is a string constant in every* `Class.forName(cName)` *call.*

PROOF SKETCH. By [P-FORNAME], the `Class` objects at all `Class.forName(cName)` calls are created from known class types. By Lemma 1, this has four implications. (1) LHM is not needed. For the rules in Figure 4.11, only [L-KWTP] is relevant, enabling every `c'.newInstance()` call to be resolved soundly as a set of regular `new t()` calls, for all `Class` objects $\mathtt{c}^t \in pt(\mathtt{c}')$. (2) In [P-GETMTD], the `Method` objects $\mathtt{m}^t_{\_}$ of all class types t accessed at a `getMethod()` call are created, where $\mathtt{m}^t_{\_}$ symbolizes over-approximately all target methods in $MTD(\mathtt{m}^t_{\_})$. The same sound approximation for `getField()` is made by [P-GETFLD]. (3) For the nine rules in Figure 4.9, only [I-INVSIG], [I-GETSIG] and [I-SETSIG] are applicable, since $\nexists \mathtt{m}^u_{\_} \in \mathtt{m}$, $\nexists \mathtt{f}^u_{\_} \in \mathtt{f}$ and $\nexists o^u_i \in \mathtt{y}$, in order to further refine their underlying method or field signatures. (4) In Figure 4.10, the set of reflective targets at each call site is over-approximated. By noting Lemma 1 again and Assumptions 1 – 3, SOLAR is sound for REFJAVA$_c$. □

SOLAR is sound subject to (4.4) – (4.6) under Assumptions 1 – 4.

Theorem 1 SOLAR *is sound for* REFJAVA *if* $SC(c)$ *holds at every reflective call* c *of the form* `(A) m.invoke(y, args)`, `(A) f.get(y)` *or* `f.set(y, x)`.

PROOF SKETCH. For reasons of symmetry, we focus only on a call, c, to `(A) m.invoke(y, args)`. $SC(c)$ is given in (4.4). If its first disjunct is true, then all abstract objects flowing into `y` are created either from known classes soundly by [L-KWTP], or lazily from unknown classes as illustrated in Figure 4.3, by applying initially [L-UKWUP] and later [L-CAST] (for Case (II)) and [L-INV] (for Case (III)), but soundly under Assumption 4. If its second disjunct is true, then SOLAR can always infer the missing class types t in a `Method` object $\mathtt{m}^u_s$ pointed to by $pt(\mathtt{m})$ to over-approximate the set of its target methods as $MTD(\mathtt{m}^t_s)$. This takes us back

to a situation equivalent to the one proved already in Lemma 2. Thus, SOLAR is sound for REFJAVA if the stated hypothesis holds. □

4.4.7 PROBE

For evaluation purposes, we instantiate PROBE, as shown in Figure 4.2, from SOLAR as follows. In PROBE, we refrain from performing SOLAR's LHM (by retaining [L-UKWTP] but ignoring [L-CAST], [L-GSET] and [L-INV]) and abandon some of SOLAR's sophisticated inference rules (by disabling [I-INVS2T], [I-GETS2T] and [I-SETS2T]).

In *Target Search*, PROBE will restrict itself to only `Method` objects $\mathtt{m}_s^t$ and `Field` objects $\mathtt{f}_s^t$, where the signature s is at least partially known.

4.4.8 Static Class Members

To handle static class members, our rules can be simply modified. In Figure 4.9, `y = null`. [I-INVTP], [I-GETTP] and [I-SETTP] are not needed (by assuming $pt(\mathtt{null}) = \varnothing$). In (4.4) – (4.6), the first disjunct is removed in each case. [I-INVS2T], [I-GETS2T] and [I-SETS2T] are modified with $o_i^u \in pt(\mathtt{y})$ being replaced by $\mathtt{y} = \mathtt{null}$. The rules in Figure 4.10 are modified to deal with static members. In Figure 4.11, [L-GSET] and [L-INV] are no longer relevant. The other rules remain applicable. The static initializers for the classes in the closed world are analyzed. This can happen at, say loads/stores for static fields as is the standard but also when some classes are discovered in [P-FORNAME], [L-CAST], [L-GSET] and [L-INV].

4.5 Implementation

SOLAR, as shown in Figure 4.5, works together with a pointer analysis. We have implemented SOLAR on top of DOOP [21], a state-of-the-art pointer analysis frame-

work for Java, by leveraging the open-source code of ELF (Chapter 3). We will compare SOLAR later with ELF and the reflection analysis provided in DOOP (also referred to as DOOP, made clear by the context in which it is used). Like DOOP and ELF, SOLAR is also implemented in the Datalog language. Presently, SOLAR consists of 303 Datalog rules written in about 2800 lines of code. As far as we know, SOLAR is the most comprehensive reflection analysis for the Java reflection API, by handling more methods than the prior work [95, 93, 21], as discussed below.

Below we extend our formalism for REFJAVA to handle the other methods in Java reflection API, divided into entry, member-introspecting and side-effect methods. For details, we refer to the open-source of SOLAR.

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

Member-Introspecting Methods In addition to the two included in REFJAVA, there are 10 more as described in Chapter 2. Let us consider `getDeclaredMethod()`, `getDeclaredMethods()` and `getMethods()` as the other ones are dealt with similarly. For a `Method` object resolved to be $\mathtt{m}_s^t$ at a `getDeclaredMethod()` call by *Propagation*, the search for building $MTD(\mathtt{m}_s^t)$ is confined to class t (the only change in the rules). Otherwise, the search is done as described in Section 4.4.4.4.

For `ms = c'.getMethods()`, which returns an array of `Method` objects, it is analyzed similarly as `m = c'.getMethod(mName)` in Figure 4.8. We first create a placeholder array object $\texttt{ms}_{ph}$ so that $\texttt{ms}_{ph} \in pt(\texttt{ms})$. As `getMethods()` is parameterless, and then insert a new `Method` object $\texttt{m}_u^t$ ($\texttt{m}_u^u$) into $pt(\texttt{ms}_{ph}.arr)$ for every $\texttt{c}^t$ ($\texttt{c}^u$) in $pt(\texttt{c}')$. The underlying pointer analysis will take care of how eventually an element of such an array flows into, say `m` in `m.invoke()`. Then the rules in Figure 4.9 are applicable. The rule for `getDeclareMethods()` is similar.

Side-Effect Methods In addition to the four in RefJava, the following four side-effect methods are also analyzed. `Constructor.newInstance()` is handled as `Class.newInstance()` except that its array argument is handled exactly as in `invoke()`. Its call sites are also modeled lazily. `Array.newInstance()`, `Array.get()` and `Array.set()` are handled as in Chapter 2. Presently, Solar does not handle `Proxy.newProxyInstance()` in its implementation. However, it could be analyzed according to its semantics as described in Section 2.3.1.

4.6 Evaluation

We evaluate Solar by comparing it (primarily) with two state-of-the-art under-approximate reflection analyses, Doop [21] and Elf [46] (introduced in Chapter 3). In some programs, Assumptions 1 – 4 may not hold (as discussed below). Thus, Solar is also treated as being under-approximate. Due to its soundness-guided design, Solar can yield significantly better under-approximations than Doop and Elf. 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 and ELF, while maintaining nearly the same precision?
- **RQ4.** How does SOLAR scale in analyzing large reflection-rich applications?

Below we describe our experimental setup, revisit our assumptions, answer these RQs in that order, and finally, provide a number of insights on our analysis.

4.6.1 Experimental Setup

All the three reflection analyses, the one provided in DOOP, ELF and SOLAR, are compared by running each together with the same DOOP pointer analysis framework [21]. For this framework, we have used its stable release (r160113). Although its beta release (r5459247-beta) handles a larger part of the Java reflection API, we did not use it as it discovers fewer reflective targets than its stable release in our recall experiment (Table 4.1); one of the main reasons is because it ignores the reflective targets whose class types are in the libraries (for efficiency reasons).

The three analyses operate on the SSA form of a program emitted by SOOT [92], 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`. The other 4 small DaCapo benchmarks are excluded since reflection is much less used.

4.6.2 Assumptions

When analyzing 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 [66, 69]. 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 [14]. For the DaCapo benchmarks, their three standard inputs are used. For `avrora` and `checkstyle`, their default test cases are used. For `findbugs`, one Java program is developed as its input since no default ones are available. For `freecs`, a server requiring user interactions, we only initialize it as the input in order to ensure repeatability. For a class in the closed-world of a program, SOLAR analyzes its static initializer at the outset if it is dynamically loaded and proceeds as discussed in Section 4.4.8 otherwise. 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 intra-procedural 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.

4.6.3 RQ1: Full Automation

Figure 4.12 compares SOLAR and existing reflection analyses [17, 93, 21, 52, 46, 14] 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.

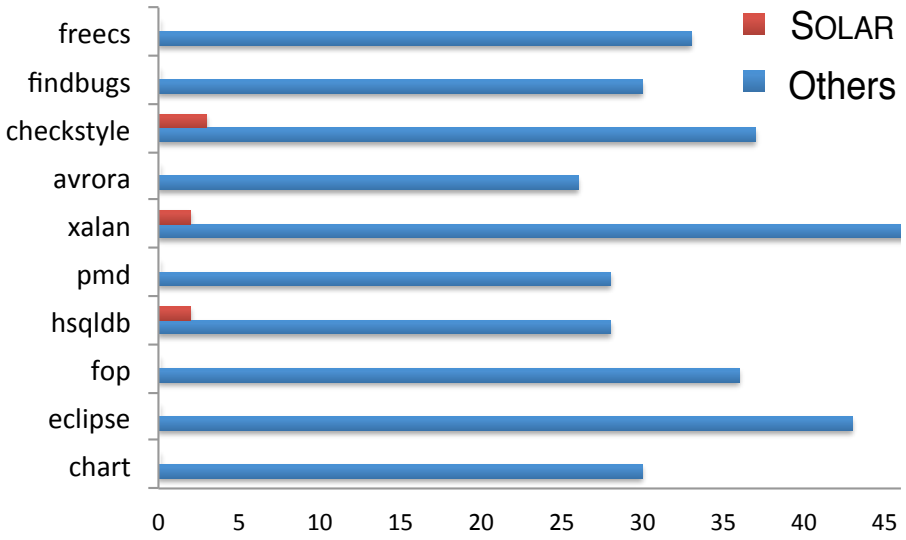


Figure 4.12: The number of annotations required for improving the soundness of unsoundly resolved reflective calls.

SOLAR analyzes 7 out of the 11 programs scalably with full automation. For `hsqldb`, `xalan` and `checkstyle`, SOLAR is unscalable (under 3 hours). With PROBE, a total of 14 reflective calls are flagged as being potentially unsoundly/imprecisely resolved. After 7 annotations, 2 in `hsqldb`, 2 in `xalan` and 3 in `checkstyle`, SOLAR is scalable, as discussed in Section 4.6.4. However, SOLAR is unscalable (under 3 hours) for `jython`, an interpreter (from `DaCapo`) for Python in which the Java libraries and application code are invoked reflectively from the Python code. Neither are DOOP and ELF.

"Others" cannot identify which reflective calls may be unsoundly resolved; they may improve their soundness by adopting the annotation approach suggested in [52]. Generally, this approach requires users to annotate the string arguments of calls to entry and member-introspecting methods if they are not constant. To reduce the annotation effort, e.g., for a `clz = Class.forName(cName)` call with `cName` being an input string, the intra-procedural post-dominating cast on the result of a

`clz.newInstance()` call is exploited to infer the class types of `clz`.

As shown in Figure 4.12, "Others" will require a total of 338 annotations initially and possibly more in the subsequent iterations (when more code is discovered). As discussed in Section 4.3.5, SOLAR's annotation approach is also iterative. However, for these programs, SOLAR requires a total of 7 annotations in only one iteration.

SOLAR outperforms "Others" due to its powerful inference system for performing reflection resolution and effective mechanism in identifying unsoundness as explained in Section 4.2.

4.6.4 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 accurately 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 Figure 4.12, with 2 `hsqldb`, 2 `xalan` and 3 in `checkstyle` as explained below.

4.6.4.1 `hsqldb`

Figure 4.13 shows the unsound and imprecise lists automatically generated by PROBE, together with the suggested annotation points (found by tracing value flow as explained in Section 4.3.6). All the call sites to the same method are numbered from 0.

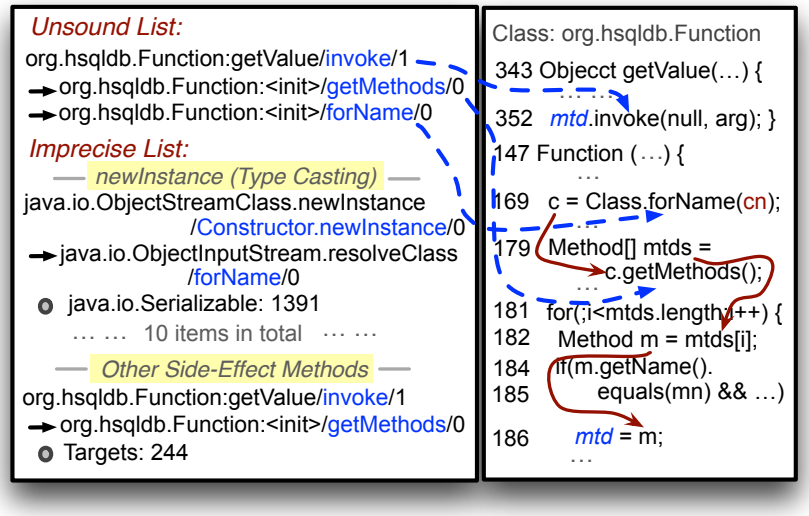


Figure 4.13: Probing `hsqldb`.

The unsound list contains one `invoke()`, with its relevant code contained in class `org.hsqldb.Function` as shown. After PROBE has finished, `mMethod` in line 352 points to a `Method` object $\mathtt{m}_u^u$ 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 like:

```
org.hsqldb.Function.<init>/java.lang.Class.forName/0 java.lang.Math
org.hsqldb.Function.<init>/java.lang.Class.forName/0 org.hsqldb.Library
```

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_i^u 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 `Class.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.

4.6.4.2 `xalan`

PROBE reports 12 unsoundly resolved `invoke()` call sites. 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.

4.6.4.3 `checkstyle`

PROBE reports no unsoundly resolved call. To see why SOLAR is unscalable, we examine one `invoke()` call in line 1773 of Figure 4.14 found automatically by PROBE that stands out as being possibly imprecisely resolved.

There are 962 target methods inferred at this call site. PROBE highlights its corresponding member-introspecting method `clz.getMethods()` (in line 1294) and

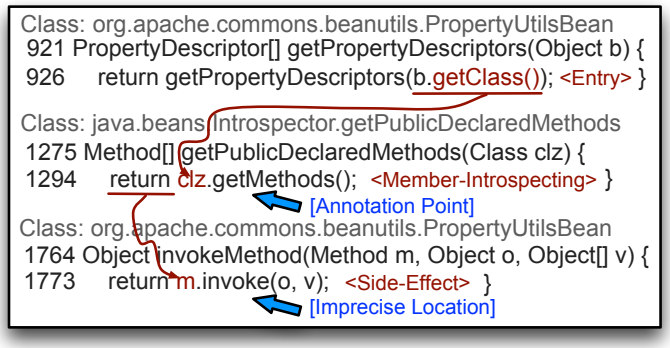


Figure 4.14: Probing `checkstyle`.

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 [17, 93, 52, 46] 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 Figure 4.1). However, SOLAR has handled this `invoke()` call soundly by using LHM, highlighting once again the importance of collective inference in reflection analysis.

Discussion Again, SOLAR's ability of accurately identifying the problematic locations is still useful even if users choose not to annotate for them. According to SOLAR's unsound or imprecise outputs, users are able to sense the significance of the effects caused by reflection, on their analysis clients. As a result, they will be more confident or careful about their analysis results as explained in Section 1.2.

4.6.5 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 calls to side-effect methods 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 Figure 4.12 are used by all the three analyses.

4.6.5.1 Recall

Table 4.1 compares the recall of DOOP, ELF and SOLAR. Note that DOOP is unscalable for `chart` and `hsqldb` (under 3 hours). We use TAMIFLEX [14], a practical dynamic reflection analysis tool to find the targets accessed at reflective calls in our programs under the inputs described in Section 4.6.2. SOLAR is the only one to achieve total recall, for all reflective targets, including both methods and

		chart		eclipse		fop		hsqldb		pmd		xalan		avrora		checkstyle		findbugs		freecs		**Total**	
		Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf	Rc	Tf
c.new	Doop	–		7		8		–		5		27		46		6		6		4		109	
	Elf	21	22	24	43	12	15	9	10	8	15	42	45	46	53	9	72	10	15	10	12	191	302
	Solar	22		37		13		10		13		43		53		72		15		12		290	
ct.new	Doop	–		1		0		–		0		0		0		1		8		2		12	
	Elf	2	3	3	4	0	1	1	2	0	1	0	1	0	0	1	24	8	115	2	2	17	153
	Solar	2		3		0		1		0		0		0		24		115		2		147	
invoke	Doop	–		0		0		–		0		0		0		0		0		0		0	
	Elf	2	2	7	10	1	1	0	0	0	7	5	32	0	0	5	28	1	1	1	55	22	136
	Solar	2		7		1		0		7		32		0		28		1		55		133	
get/set	Doop	–		8		8		–		8		8		0		8		8		8		56	
	Elf	8	8	1039	1039	8	8	8	8	8	8	8	8	0	0	8	8	8	8	8	8	1103	1103
	Solar	8		1039		8		8		8		8		0		8		8		8		1103	

Table 4.1: Recall comparison. Tf denotes the number of targets found by TamiFlex. Rc denotes the number of such (true) targets also discovered by each reflection analysis. There are two types of `newInstance()`: c.new (in `Class`) and ct.new (in `Constructor`).

fields, accessed. There are two cases highlighted in gray and yellow in Table 4.1, where our closed-world assumption is violated. For each DaCapo benchmark, TamiFlex uses a harness that is different from the one in the `xdeps` version provided for static analysis. Thus, Solar "misses" one target for a `newInstance()` call that exists only in the TamiFlex's harness. For `eclipse`, `fop`, `pmd` and `xalan`, Solar "misses" a few target methods due to the differences in a few classes (e.g., `org.apache.xerces.parsers.SAXParser`), where different versions in the two Solar and TamiFlex settings are used.

For practical applications, a reflection analysis must handle `newInstance()` and `invoke()` well in order to build a program's call graph accurately.

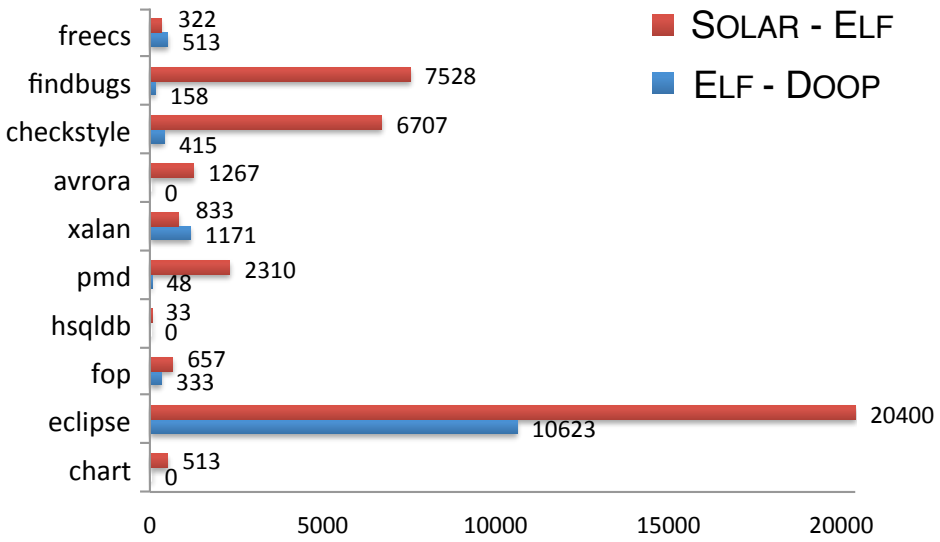


Figure 4.15: More true caller-callee relations discovered in recall by Solar than Elf, denoted Solar−Elf, and by Elf than Doop, denoted Elf−Doop.

Figure 4.15 compares Doop, Elf and Solar in terms of true caller-callee relations discovered. These numbers are statically calculated and obtained by an instrumental tool written in terms of Javassist [34]. According to Table 4.1, 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*

		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

Table 4.2: 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.

caller-callee relations found for the 10 programs under the inputs described in Section 4.6.2. 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.

Let us examine `eclipse` and `checkstyle`. We start with `eclipse`. According to Table 4.1, ELF finds 26 more target methods than DOOP, causing 10623 more true caller-callee relations to be discovered. Due to LHM, SOLAR finds 13 more target methods at `newInstance()` calls than ELF, resulting in 20400 additional true caller-callee relations to be introduced. Let us consider now `checkstyle`. Due to collective inference, SOLAR has resolved 23 more target methods at `invoke()` calls than ELF, resulting in 2437 more true caller-callee relations (i.e., over a third of the total number of such relations, 6707) to be discovered.

For `get()/set()`, SOLAR (and ELF) resolves more fields (of type `String`) than DOOP in `eclipse`, causing 7 more true caller-callee relations to be found.

4.6.5.2 Precision

Table 4.2 compares the analysis precision of DOOP, ELF and SOLAR with two popular clients. Despite achieving better recall (Table 4.1), which results in more true caller-callee relations to be discovered (Figure 4.15), 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.

4.6.6 RQ4: Efficiency

Table 4.3 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 analyzing `hsqldb`, `xalan` and `checkstyle`, SOLAR requires some lightweight annotations. Their analysis times are the ones consumed by SOLAR on analyzing the annotated programs. Note that these annotated programs are also used by DOOP and ELF (to ensure a fair comparison).

	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

Table 4.3: Comparing SOLAR, ELF and DOOP in terms of analysis times (secs).

4.6.7 On the Practical Effectiveness of SOLAR

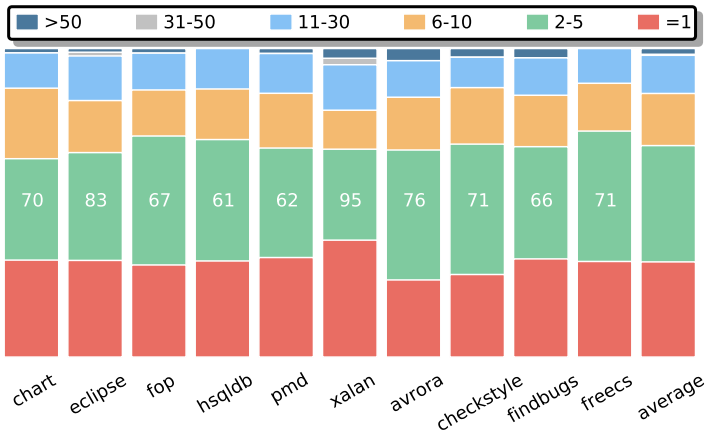


Figure 4.16: Percentage distribution for the number of types inferred at cast-related LHM points, with the total on each bar.

Let us recall the three novel design aspects of SOLAR (Figure 4.2) that have made it practically effective. First, Assumption 4 holds usually in applications (Section 4.6.2). LHM handles reflective object creation well (Figure 4.3), particularly

at the cast operations on `newInstance()`. As shown in Figure 4.16, the number of inferred types in a program is 1 ($\leqslant 10$) at 30.8% (85.4%) of its cast-related LHM points. Some types (e.g., `java.io.Serializable`) are quite wide, giving rise to more than 50 inferred types each, but they are rare, appearing at an average of about 1.9% cast-related LHM points in a program. Second, SOLAR's collective inference is fairly powerful and precise (as demonstrated by the real-world cases in Figures 4.4 and 4.14). Third, SOLAR is capable of accurately finding "problematic" reflective calls for lightweight annotations (Section 4.6.4).

4.7 Discussion

SOLAR is the first reflection analysis that allows its soundness to be reasoned about when some reasonable assumptions are met (Section 4.3.1) and yields significantly more sound results (than previous approaches) otherwise (Section 4.6). In addition, SOLAR has been demonstrated to be able to (1) accurately identify the reflective calls which are resolved unsoundly or imprecisely and (2) guide lightweight annotations to be made to improve the quality of the analysis, which is useful for many applications in practice as explained in Chapter 1.

However, since SOLAR tries to maximally enhance its inference system to achieve its goal as explained in Section 4.2, some inference strategies may be aggressive for some programs, resulting in too many reflective targets inferred. Such over-approximation may render SOLAR unscalable (for some large Java programs) and need users to get involved: accepting PROBE's precise, scalable but less sound results, or giving annotations to enable SOLAR to meet their specific high-quality requirements in terms of soundness (or precision).

Chapter 5

Related Work

5.1 Static Reflection Analysis

Livshits et al. [52] introduce the first static reflection analysis for Java. Reflection analysis works as part of a pointer analysis, with each being both the producer and consumer of the other. In addition, the string values (which are used to specify the reflective targets) are resolved if they are string constants or the ones involving simple string manipulations such as `StringBuffer.append()`. When the string values are unknown, the reflective targets created at some `newInstance()` call sites (say i), could still be resolved (by the cast types) when the cast operations, which intra-procedurally post-dominate i, are available. Many modern pointer analysis frameworks such as Doop [21], Wala [93], Chord [19], adopt a similar approach to analyze Java reflection in their implementations, and many following reflection analyses [17, 74, 46, 47], including both Elf and Solar, are inspired by it.

Besides [52], the design and implementation of Elf benefit greatly from the reflection analysis implementation in Doop (r160113). Generally, its reflection handling can be seen as analogous to adding a sophisticated analysis similar as [52]

but in conjunction with a context-sensitive pointer analysis. In addition, Doop considers more Java features than [52] (e.g., distinguishing instance from static field operations) when handling reflection.

However, there are significant differences between ELF and these two previous analyses, which enable ELF to be more effective. ELF is the first one to adopt the self-inferencing property (discovered in our study in Section 2.3.7), which exploits and leverages much richer hints inherent in the code to resolve reflection. In Figure 3.1, except the dashed red arrow at `newInstance()`, all the other seven dashed blue and red arrows are newly introduced in ELF (recall that the dashed arrows represent *Target Inference* and the solid arrows represent *Target Propagation*). In addition, these two previous analyses only use the solid arrows to analyze reflection: [52] uses the blue solid arrows and DOOP (r160113) uses the red solid arrows (see Figure 3.1). Such simple analysis strategies are totally changed in ELF by combining both solid and our new dashed arrows in different ways, enabling the generation of many new and useful inference strategies. As a result, many reflective targets (at different side-effect call sites) can be resolved effectively even if both class and member names (i.e., those string values) are unknown statically. Note that the current DOOP release (r5459247-beta) have handled much more reflection API than its old stable version (r160113), and some reflection analysis strategies are also improved and introduced in a recent paper [74].

In [74], the authors propose to leverage partial string information to resolve the reflective targets. As explained in Section 2.3.3, many string arguments involve complicated string manipulations and the values of them are hard to resolve statically. However, in some cases, the substring method in [74] could make it possible to leverage these partial string values, as a part of the self-inferencing property (Section 2.3.7), to help infer the reflective targets. Combining this method with

some sophisticated string analysis approaches such as that in [20], is expected to generate more effective results, which is worth pursuing in the future work.

In addition, to achieve more sound results, Smaragdakis et al. [74] present two analysis strategies: (1) leveraging both intra- and inter-procedural cast operations to infer the class types at the corresponding `Class.forName()` call sites, and (2) leveraging the member names at the member-introspecting call sites (i.e., the blue circles formed by *Target Propagation* in Figure 3.1) to infer the class types at the corresponding `Class.forName()` call sites. However, in some cases, both strategies may render the analysis itself too imprecise, which may affect scalability. For strategy (2), it would suffer from the precision loss when the member names are common so that the related class names cannot be distinguished precisely (see the cases in Section 3.6.5 for details). For both strategies (1) and (2), the class types (at a `Class.forName()` call site), which are back-propagated (inferred) from some member-introspecting call sites (cast sites), may further pollute the analysis precision at the other program points.

To reduce such imprecision, the authors of [74] further develop an approach, called inventing objects, to create objects at the `newInstance()` call sites (according to the types at the related cast sites), rather than the corresponding `forName()` call sites. This method is similar to the approach for handling Case (II) in SOLAR's LHM (Figure 4.3). Since none of the above approaches consider the soundness boundary when analyzing reflection, like all previous work [52, 17, 21, 93, 46], [74] can not behave like SOLAR to reason about soundness and accurately identify unsoundness either.

Barros et al. [4] present a static analysis for analyzing the reflection and intents in Android apps. Their soundness is in terms of the clients which use the reflection analysis. For example, if a client is to check whether some sensitive data leaks, and if

a reflective call `invoke()` cannot be resolved, then the analysis would conservatively assume that the `invoke()` call site may return some sensitive data, which is sound regarding the client. However, the reflection analysis itself is still unsound as the previous work [52, 17, 21, 93, 46].

Generally, the reflection inference in [4] can be seen as a subset of the one in ELF (also SOLAR), as it only leverages partial self-inferencing property (introduced in Section 2.3.7). For instance, regarding the arguments in an `invoke()` call, only the number of the arguments are considered and the types of the arguments are ignored. Although this may not cause practical negative consequences in Android apps [4], it would make the reflection inference imprecise for analyzing Java applications. For example, consider the case in `Eclipse` in Figure 2.8. If the types of the arguments `parameters` in line 175 are ignored, many (false) target methods would be inferred as the methods with only one argument are very common (even if in one class).

Despite recent advances [47, 74, 46], a sophisticated reflection analysis does not co-exist well with a sophisticated (more precise) pointer analysis, since the latter is usually unscalable for large programs [52, 47, 74, 46]. If a scalable but imprecise pointer analysis is used instead, the reflection analysis may introduce many false call graph edges, making its underlying client applications to be too imprecise to be practically useful. This problem could be alleviated by using a recently proposed program slicing approach, called program tailoring [48]. Briefly, program tailoring accepts a sequential criterion (e.g., a sequence of reflective call sites: `forName()`→`getMethod()`→`invoke()`) and generates a sound tailored program, in which the statements are only relevant to the given sequential criterion. In other words, the tailored program comprises the statements in all possible execution paths passing through the sequence(s) in the given order. As a result, a more precise (but less scalable) pointer analysis may be scalable when the tailored

program (with smaller size) is analyzed, resulting a more precise result resolved at the given reflective call site (e.g., the above `invoke()` call site). These reflective call sites could be the problematic ones generated by SOLAR as described in Section 4.3.5 and demonstrated in [48].

5.2 Dynamic Reflection Analysis

Hirzel et al. [31] propose 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. [14] suggest 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 and SOLAR are complementary to TAMIFLEX by resolving reflective calls statically rather than dynamically.

5.3 Others

Braux and Noyé [16] provide 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 and SOLAR can be viewed as tools for inferring such constraints automatically.

To increase code coverage, some static analysis tools [21, 95] 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. [79] introduce a framework that exploits domain knowledge to automatically generate a specification of framework-related behaviors (e.g., reflection usage) by processing both application code and configuration files. Both ELF and SOLAR 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 [14, 31] work in the presence of both dynamic class loading and reflection. Nguyen and Xue [66, 97] introduce an inter-procedural side-effect analysis for open-world Java programs (by allowing dynamic class loading but disallowing reflection). Like other static reflection analyses [21, 52, 93, 74], ELF and SOLAR can presently analyze closed-world Java programs only.

Chapter 6

Conclusions

Reflection analysis is a long-standing hard open problem. In the past years, almost all the research papers regard it as a separate assumption and most static analysis tools either handle it partially or totally ignore it. In the program analysis community, people hold to the common opinion that "*reflection is a dynamic feature, so how could it be handled effectively in static analysis?*" A further question, "*what if the string inputs are read from external files, how can you know their values, statically?*" would probably remove any further counter arguments. In this case, this thesis has produced evidence that effective static analysis for handling Java reflection is feasible and we make a step forward in tackling the problem by making the following contributions.

In Chapter 2, we conduct a study on real-world Java programs to understand how reflection is used in practice. Many useful findings are summarized, particularly the self-inferencing property, which can be exploited and leveraged to resolve reflection statically but in an effective way. Then the above "*what if the string inputs are read from external files, how can you know their values, statically?*" question can be simply answered by "*you do not need to know the string values*

specifying the names of the reflective targets; instead, you can infer the reflective targets at their usage points by the self-inferencing property".

This is exactly what ELF accomplishes in Chapter 3. Evaluation shows that ELF is able to infer many true reflective targets precisely even if the input strings are unknown; in addition, it makes a good trade-off among soundness, precision and scalability. However, a precise reflection analysis with acceptable soundness (usually more sound than the string resolution approach) like ELF, may not fulfill the requirements of certain clients. In these clients, significantly more sound results are required; in addition, the ability of being aware of unsound/imprecise places would be very helpful in practice as discussed in Section 1.2. Moreover, if the reflection analysis itself could be tuned (iteratively) by providing no or few annotations, the requirements of some clients (e.g., some bug/security analyses or verification tools), which demand high-quality analysis results, could be satisfied.

The above goals, although challenging, could be achieved in one reflection analysis, called SOLAR as introduced in Chapter 4. In fact, the design insight of SOLAR is simple as explained in Section 4.2 and the results of evaluation (on 11 large Java programs) are quite promising as shown in Section 4.6. Briefly, SOLAR is the only one to achieve total recall and is significantly more sound than the other two state-of-the-art reflection analyses, ELF and DOOP (r160113 and r5459247-beta), subject to only 7 guided annotations in 3 programs. In addition, SOLAR accurately identifies 14 unsoundly/imprecisely resolved reflective call sites (in total), with no false positives. Although SOLAR has limitations as discussed in Section 4.7, we believe our results are representative of the potential for a considerable improvement over current practice.

Bibliography

[1] K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. ECOOP, pages 378–400, 2013.

[2] N. Allen, B. Scholz, and P. Krishnan. Staged points-to analysis for large code bases. CC, pages 131–150, 2015.

[3] 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, pages 259–269, 2014.

[4] P. Barros, R. Just, S. Millstein, P. Vines, W. Dietl, M. d'Amorim, and M. D. Ernst. Static analysis of implicit control flow: Resolving Java reflection and Android intents. ASE, pages 669–679, 2015.

[5] O. Bastani, S. Anand, and A. Aiken. Interactively verifying absence of explicit information flows in Android apps. OOPSLA, pages 299–315, 2015.

[6] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using BDDs. PLDI, pages 103–114, 2003.

[7] S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khang, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel,

A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. OOPSLA, pages 169–190, 2006.

[8] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of andersen's analysis in practice. SAS, pages 60–76, 2011.

[9] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Thresher: Precise refutations for heap reachability. PLDI '13, pages 275–286, 2013.

[10] S. Blackshear, B.-Y. E. Chang, and M. Sridharan. Selective control-flow abstraction via jumping. OOPSLA, pages 163–182, 2015.

[11] S. Blackshear, A. Gendreau, and B.-Y. E. Chang. Droidel: A general approach to Android framework modeling. SOAP, pages 19–25, 2015.

[12] E. Bodden. Efficient hybrid typestate analysis by determining continuation-equivalent states. ICSE, pages 5–14, 2010.

[13] E. Bodden, L. Hendren, and O. Lhoták. A staged static program analysis to improve the performance of runtime monitoring. ECOOP, pages 525–549, 2007.

[14] 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, pages 241–250, 2011.

[15] G. Bracha and D. Ungar. Mirrors: Design principles for meta-level facilities of object-oriented programming languages. OOPSLA, pages 331–344, 2004.

[16] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. PEPM, pages 2–11, 2000.

[17] M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. OOPSLA, pages 243–262, 2009.

[18] S. Chiba. Load-time structural reflection in Java. ECOOP, pages 313–336, 2000.

[19] Chord. A program analysis platform for Java. http://www.cc.gatech.edu/~naik/chord.html.

[20] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. SAS, pages 1–18, 2003.

[21] DOOP. A sophisticated framework for Java pointer analysis. http://doop.program-analysis.org.

[22] D. R. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI, pages 1–16, 2000.

[23] D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. SOSP, pages 57–72, 2001.

[24] M. D. Ernst, R. Just, S. Millstein, W. Dietl, S. Pernsteiner, F. Roesner, K. Koscher, P. Barros, R. Bhoraskar, S. Han, P. Vines, and E. X. Wu. Collaborative verification of information flow for a high-assurance app store. CCS, pages 1092–1104, 2014.

[25] Y. Feng, S. Anand, I. Dillig, and A. Aiken. Apposcopy: semantics-based detection of Android malware through static analysis. FSE, pages 576–587, 2014.

[26] Y. Feng, X. Wang, I. Dillig, and T. Dillig. Bottom-up context-sensitive pointer analysis for Java. APLAS, pages 465–484, 2015.

[27] I. R. Forman and N. Forman. *Java Reflection in Action*. Manning Publications Co., 2004.

[28] D. P. Friedman and M. Wand. Reification: Reflection without metaphysics. In *Proceedings of the 1984 ACM Symposium on LISP and Functional Programming*, LFP, pages 348–355, 1984.

[29] B. Hardekopf and C. Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. PLDI, pages 290–299, 2007.

[30] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. CGO, pages 289–298, 2011.

[31] M. Hirzel, D. V. Dincklage, A. Diwan, and M. Hind. Fast online pointer analysis. *ACM Trans. Program. Lang. Syst.*, 29(2), 2007.

[32] W. Huang, Y. Dong, and A. Milanova. Type-based taint analysis for Java web applications. FASE, pages 140–154, 2014.

[33] W. Huang, Y. Dong, A. Milanova, and J. Dolby. Scalable and precise taint analysis for Android. ISSTA, pages 106–117, 2015.

[34] Javassist. A Java bytecode manipulation framework. http://www.javassist.org.

[35] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and detecting real-world performance bugs. PLDI, pages 77–88, 2012.

[36] G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. CC, pages 41–60, 2013.

[37] G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. PLDI, pages 423–434, 2013.

[38] C. Kirkegaard and A. Møller. Static analysis for Java servlets and JSP. SAS, pages 336–352, 2006.

[39] T. Kwon and Z. Su. Static detection of unsafe component loadings. CC, pages 122–143, 2012.

[40] O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. POPL, pages 3–16, 2011.

[41] O. Lhoták and L. Hendren. Scaling Java points-to analysis using Spark. CC, pages 153–169, 2003.

[42] O. Lhoták and L. Hendren. Context-sensitive points-to analysis: is it worth it? CC, pages 47–64, 2006.

[43] H. Li, X. Rival, and B. E. Chang. Shape analysis for unstructured sharing. SAS, pages 90–108, 2015.

[44] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. L. Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. IccTA: Detecting inter-component privacy leaks in Android apps. ICSE, pages 280–291, 2015.

[45] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. FSE, pages 343–353, 2011.

[46] Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. ECOOP, pages 27–53, 2014.

[47] Y. Li, T. Tan, and J. Xue. Effective soundness-guided reflection analysis. SAS, pages 162–180, 2015.

[48] Y. Li, T. Tan, Y. Zhang, and J. Xue. Program tailoring: Slicing by sequential criteria. ECOOP, 2016.

[49] J. Liu and X. Rival. Abstraction of arrays based on non contiguous partitions. VMCAI, pages 282–299, 2015.

[50] B. Livshits and M. S. Lam. Finding security vulnerabilities in Java applications with static analysis. USENIX Security Symposium, pages 271–286, 2005.

[51] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: A manifesto. *Commun. ACM*, 58(2):44–46, 2015.

[52] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. APLAS, pages 139–160, 2005.

[53] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. Technical report, Stanford University, 2005.

[54] Y. Lu, L. Shang, X. Xie, and J. Xue. An incremental points-to analysis with CFL-reachability. CC, pages 61–81, 2013.

[55] M. Madsen and A. Møller. Sparse dataflow analysis with pointers and reachability. SAS, pages 201–218, 2014.

[56] P. Maes. Concepts and experiments in computational reflection. OOPSLA, pages 147–155, 1987.

[57] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. FSE, pages 63–72, 2004.

[58] A. Milanova, W. Huang, and Y. Dong. CFL-reachability and context-sensitive integrity types. PPPJ, pages 99–109, 2014.

[59] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. ISSTA, pages 1–11, 2002.

[60] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, 2005.

[61] A. Møller and M. I. Schwartzbach. Static program analysis, 2015. Department of Computer Science, Aarhus University, `http://cs.au.dk/~amoeller/spa/`.

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

[63] M. Naik and A. Aiken. Conditional must not aliasing for static race detection. POPL, pages 327–338, 2007.

[64] M. Naik, A. Aiken, and J. Whaley. Effective static race detection for Java. PLDI, pages 308–319, 2006.

[65] M. Naik, C. Park, K. Sen, and D. Gay. Effective static deadlock detection. ICSE, pages 386–396, 2009.

[66] P. H. Nguyen and J. Xue. Interprocedural side-effect analysis and optimisation in the presence of dynamic class loading. ACSC, pages 9–18, 2005.

[67] M. Rapoport, O. Lhoták, and F. Tip. Precise data flow analysis in the presence of correlated method calls. SAS, pages 54–71, 2015.

[68] V. Rastogi, Y. Chen, and X. Jiang. DroidChameleon: Evaluating Android anti-malware against transformation attacks. ASIA CCS, pages 329–334, 2013.

[69] J. Sawin and A. Rountev. Improving static resolution of dynamic class loading in Java using dynamically gathered environment information. *Automated Software Engg.*, pages 357–381, 2009.

[70] B. Scholz, H. Jordan, P. Subotic, and T. Westmann. On fast large-scale program analysis in Datalog. CC, pages 196–206, 2016.

[71] L. Shang, Y. Lu, and J. Xue. Fast and precise points-to analysis with incremental CFL-reachability summarisation. ASE, pages 270–273, 2012.

[72] Y. Smaragdakis and G. Balatsouras. Pointer analysis. *Foundations and Trends in Programming Languages*, pages 1–69, 2015.

[73] Y. Smaragdakis, G. Balatsouras, and G. Kastrinis. Set-based pre-processing for points-to analysis. OOPSLA, pages 253–270, 2013.

[74] Y. Smaragdakis, G. Balatsouras, G. Kastrinis, and M. Bravenboer. More sound static handling of Java reflection. APLAS, pages 485–503, 2015.

[75] Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: understanding object-sensitivity. POPL, pages 17–30, 2011.

[76] Y. Smaragdakis, J. Evans, C. Sadowski, J. Yi, and C. Flanagan. Sound predictive race detection in polynomial time. POPL, pages 387–400, 2012.

[77] Y. Smaragdakis, G. Kastrinis, and G. Balatsouras. Introspective analysis: Context-sensitivity, across the board. PLDI, pages 485–495, 2014.

[78] B. Smith. Reflection and semantics in a procedural language. *Ph.D. thesis, Massachusetts Institute of Technology*, 1982.

[79] M. Sridharan, S. Artzi, M. Pistoia, S. Guarnieri, O. Tripp, and R. Berg. F4F: Taint analysis of framework-based web applications. OOPSLA, pages 1053–1068, 2011.

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

[81] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav. Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. 2013.

[82] M. Sridharan, S. J. Fink, and R. Bodik. Thin slicing. PLDI, pages 112–122, 2007.

[83] M. Sridharan, D. Gopan, L. Shan, and R. Bodík. Demand-driven points-to analysis for Java. OOPSLA, pages 59–76, 2005.

[84] Y. Su, D. Ye, and J. Xue. Parallel pointer analysis with CFL-reachability. ICPP, pages 451–460, 2014.

[85] Y. Sui, P. Di, and J. Xue. Sparse flow-sensitive pointer analysis for multi-threaded programs. CGO, pages 160–170, 2016.

[86] Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. CGO, pages 1–11, 2013.

[87] Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. ISSTA, pages 254–264, 2012.

[88] Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: scalable path-sensitive pointer analysis on full-sparse SSA. APLAS, pages 155–171, 2011.

[89] F. Sun, L. Xu, and Z. Su. Static detection of access control vulnerabilities in web applications. USENIX Security, pages 11–27, 2011.

[90] T. Tan, Y. Li, and J. Xue. Making k-object-sensitive pointer analysis more precise with still k-limiting. SAS, 2016.

[91] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. PLDI, pages 87–97, 2009.

[92] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. CASCON, pages 1–13, 1999.

[93] WALA. T.J. Watson libraries for analysis. http://wala.sf.net.

[94] G. Wassermann and Z. Su. Static detection of cross-site scripting vulnerabilities. ICSE, pages 171–180, 2008.

[95] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. PLDI, pages 131–144, 2004.

[96] X. Xiao and C. Zhang. Geometric encoding: Forging the high performance context sensitive points-to analysis for Java. ISSTA, pages 188–198, 2011.

[97] J. Xue and P. H. Nguyen. Completeness analysis for incomplete object-oriented programs. CC, pages 271–286, 2005.

[98] D. Yan, G. Xu, and A. Rountev. Demand-driven context-sensitive alias analysis for Java. ISSTA, pages 155–165, 2011.

[99] D. Ye, Y. Su, Y. Sui, and J. Xue. WPBOUND: enforcing spatial memory safety efficiently at runtime with weakest preconditions. ISSRE, pages 88–99, 2014.

[100] D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined values with static value-flow analysis. CGO, pages 154–164, 2014.

[101] S. Ye, Y. Sui, and J. Xue. Region-based selective flow-sensitive pointer analysis. SAS, pages 319–336, 2014.

[102] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. CGO, pages 218–229, 2010.

[103] Q. Zhang, M. R. Lyu, H. Yuan, and Z. Su. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. PLDI, pages 435–446, 2013.

[104] S. Zhang, H. Lü, and M. D. Ernst. Finding errors in multithreaded GUI applications. ISSTA, pages 243–253, 2012.

[105] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in Datalog. PLDI, pages 239–248, 2014.

[106] Y. Zhauniarovich, M. Ahmad, O. Gadyatskaya, B. Crispo, and F. Massacci. StaDynA: Addressing the problem of dynamic code updates in the security analysis of Android applications. CODASPY, pages 37–48, 2015.