More Sound Static Handling of Java Reflection

Yannis Smaragdakis George Kastrinis
George Balatsouras
Department of Informatics, University of Athens
Athens, 15784, Greece
{smaragd,gkastrinis,gbalats}@di.uoa.gr

Martin Bravenboer
LogicBlox Inc.
Two Midtown Plaza
Atlanta, GA 30309, USA
martin.bravenboer@acm.org

Abstract

Reflection is a highly dynamic language feature that poses grave problems for static analyses. In the Java setting, reflection is ubiquitous in large programs. Any handling of reflection will be approximate, and overestimating its reach in a large codebase can be catastrophic for precision and scalability. We present an approach for handling reflection with improved empirical soundness (as measured against prior approaches and dynamic information) in the context of a points-to analysis. Our approach is based on the combination of string-flow and points-to analysis from past literature augmented with (a) substring analysis and modeling of partial string flow through string builder classes; (b) new techniques for analyzing reflective entities based on information available at their use-sites. The resulting analysis is general, without any need for hand-tuning. In experimental comparisons with prior approaches, we demonstrate a combination of both improved soundness (recovering the majority of missing call-graph edges) and increased performance.

1. Introduction

Whole-program static analysis is the engine behind several modern programming facilities for program development and understanding. Compilers, bug detectors, security checkers, modern development environments (with automated refactorings, slicing facilities, and auto-complete functionality), and a myriad other tools routinely employ static analysis machinery. Even the seemingly simple effort of computing a program's call-graph (i.e., which program function can call which other) requires sophisticated analysis in order to achieve precision in a modern language.

Yet, static whole-program analysis suffers in the presence of common dynamic features, especially reflection. When a Java program accesses a class by supplying its name as a run-time string, via the `Class.forName` library call, the static analysis has very few available courses of action: It either needs to conservatively over-approximate (e.g., assume that *any* class can be accessed, possibly limiting the set later, after the returned object is used), or it needs to perform a string analysis that will allow it to infer the contents of the `forName` string argument. Both options can be detrimental to the scalability of the analysis: the conservative over-approximation may never become constrained enough by further instructions to be feasible in practice; precise string analysis is impractical for large programs of realistic size. It is telling that no practical Java program analysis framework in existence handles reflection in a sound way, although other language features are modeled soundly.[1] Thus, the first challenge of static reflection handling is *empirical soundness*: the static analysis is intended to thoroughly cover dynamic behavior. After all, the reason to perform static analysis is to capture more program behaviors than a dynamic execution—the converse is a paradox that puts the value of the static analysis in question. Even if guaranteed full soundness is impractical, it is desirable to capture most actual behavior for the well-behaved reflection patterns encountered in regular, non-adversarial programs.

The second challenge of handling reflection in a static analysis is *scalability*. The online documentation of the IBM WALA library [Fink et al.] concisely summarizes the current state of the practice, for *points-to analysis* in the Java setting.

> *Reflection usage and the size of modern libraries/frameworks make it very difficult to scale flow-insensitive points-to analysis to modern Java programs. For example, with default settings,* WALA*'s pointer analyses cannot handle any program linked against the Java 6 standard libraries, due to extensive reflection in the libraries.*

The same caveats routinely appear in the research literature. Multiple published points-to analysis papers analyze well-known benchmarks with reflection disabled [Smaragdakis et al. 2011; Kastrinis and Smaragdakis 2013b; Ali and Lhoták 2012, 2013].

In this paper, we describe an approach to analyzing reflection in the above setting (points-to analysis, Java). Our approach requires no manual configuration and achieves significantly higher empirical soundness without sacrificing

[1] In our context, *sound* = over-approximate, i.e., guaranteeing that all possible behaviors of reflection operations are modeled.

scalability, for realistic benchmarks and libraries (DaCapo Bach and Java 7).

There are two new technical elements in our work:

- We augment prior algorithms for inter-related reflection and points-to analysis [Livshits et al. 2005; Livshits 2006] with a *substring* analysis, as well as a string flow analysis. The insight behind this treatment is that reflection is often used to dynamically access entities with partially known names, and the dynamic part of the configuration is limited to package prefixes, method name suffixes, etc. Thus, much of the dynamic information (i.e., other substrings contributing to the eventual string representing a class or member name) can be supplanted by mere over-approximation.

  The (simplified) code excerpt shown in Figure 1, found in the *xalan* DaCapo benchmark, demonstrates the need for substring analysis in order to resolve reflective method invocations. The methods shown belong to class `org.apache.xalan.processor.XSLTAttributeDef`, which represents an attribute for an element in an XSLT stylesheet. The method `setAttrVal()` computes and sets the value of this attribute, for a given element (of type `ElemTemplateElement`), via reflection (calls `getClass`, `getMethod`, `invoke`). In order to achieve this, it first has to determine the exact name of the setter method of the element, by calling `getSetterMethodName()`. The attribute contains a field `m_name`, which holds the local name of the attribute without any prefix. The method simply transforms this local name to a setter method by adding a "`set`" prefix, removing dashes, and changing it to camel case. (Reflective calls have to be generic, which explains why patterns such as this, relying on naming conventions and employing some basic string transformation, are common in practice.)

  Note that, in order to resolve the setter method, one needs to track the flow of the "`set`" prefix through the `StringBuffer` object and use it to match against any possible setter methods of `ElemTemplateElement`.

- We introduce new techniques for inferring the result of reflection calls based on how this result is used later in the program. Consider, for instance, a sequence of program statements, possibly remote to each other, yet with values flowing from one to the next:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Object o1 = c2.newInstance();
...      // o2 aliases o1
e = (Event) o2;
```

  Assuming that the cast is intended to succeed, we get information regarding the result of the `newInstance` call, which, in turn, informs the result of the `forName` call. That is, by keeping track of the flow of objects from an original call that accepts strings and produces reflection objects (such as a `forName` or a `getField` call) we infer that this string could have been the name of a subtype of `Event`. This effectively propagates information back to the source of an unknown object.

  Similar reasoning has been employed in other work [Livshits et al. 2005; Li et al. 2014]. Yet our approach generalizes past techniques significantly: we support inter-procedural reasoning, leverage information from strings and not just casts, and introduce "invented" objects that fully materialize at the point of a cast and subsequently aid the analysis modeling.

```
boolean setAttrVal(..., ElemTemplateElement el) {
   String setterString = getSetterMethodName();
   Object val = processValue(..., el);

   Object[] args = new Object[]{ val };
   Class[] argTypes =
                   new Class[]{val.getClass()};
   Method meth = el
       .getClass()
       .getMethod(setterString, argTypes);
   meth.invoke(el, args);
}

public String getSetterMethodName() {
   StringBuffer outBuf = new StringBuffer();
   outBuf.append("set");

   for (int i = 0; i < m_name.length(); i++) {
      char c = m_name.charAt(i);
      if ('-' == c) {
         i++;
         c = m_name.charAt(i);
         c = Character.toUpperCase(c);
      }
      else if (0 == i) {
         c = Character.toUpperCase(c);
      }
      outBuf.append(c);
   }
   return outBuf.toString();
}
```

Figure 1: Example of reflection leveraging partial strings.

As a result of carefully balancing the soundness and scalability of all techniques, our approach is both more sound and more scalable than past work. In experimental comparisons with the recent ELF system [Li et al. 2014] (itself improving over the reflection analysis of the DOOP framework [Bravenboer and Smaragdakis 2009b]), our algorithm discovers most of the call-graph edges missing (relative to a dynamic analysis) from ELF's reflection analysis. This improvement in empirical soundness is accompanied by *increased* performance relative to ELF, demonstrating that near-sound handling of reflection is often practically possible.

In summary, our work:

- introduces key techniques in static reflection handling that contribute greatly to empirical soundness and to scalability;
- thoroughly quantifies the empirical soundness of a static points-to analysis, compared to past approaches and a dynamic analysis;
- is implemented and evaluated on top of an open, available framework (DOOP [Bravenboer and Smaragdakis 2009b]) that can be improved by others. This can offer a platform for experimentation with sophisticated static reflection handling, possibly also leading to good static handling of other dynamic features (e.g., complex dynamic loading).

2. Background: Joint Reflection and Points-To Analysis

As necessary background, we next present an abstracted model of the inter-related reflection and points-to analysis upon which our approach builds. The model is a light reformulation of the analysis introduced by Livshits et al. [2005]; Livshits [2006]. The main insight of the Livshits et al. approach is that reflection analysis relies on points-to information, because the different key elements of a reflective activity may be dispersed throughout the program. A typical pattern of reflection usage is with code such as:

```
1 String className = ... ;
2 Class c = Class.forName(className);
3 Object o = c.newInstance();
4 String methodName = ... ;
5 Method m = c.getMethod(methodName, ...);
6 m.invoke(o, ...);
```

All of the above statements can occur in distant program locations, across different methods, invoked through virtual calls from multiple sites, etc. Thus, a whole-program analysis with an understanding of heap objects is required to track reflection with any amount of precision. This suggest the idea that reflection analysis can leverage points-to analysis—it is a client for points-to information. At the same time, points-to analysis needs the results of reflection analysis—e.g., to determine which method gets invoked in the last line of the above example, or what objects each of the example's local variables point to. Thus, under the Livshits et al. approach, reflection analysis and points-to analysis become mutually recursive, or effectively a single analysis.

This mutual recursion introduces significant complexity. Fortunately, a large amount of research in points-to analysis has focused on specifying analyses declaratively [Reps 1994; Whaley et al. 2005; Lam et al. 2005; Whaley and Lam 2004; Bravenboer and Smaragdakis 2009b; Kastrinis and Smaragdakis 2013a; Bravenboer and Smaragdakis 2009a; Kastrinis and Smaragdakis 2013b; Naik et al. 2006; Liang and Naik 2011; Guarnieri and Livshits 2009], in the Datalog programming language. Datalog is ideal for encoding mutually recursive logic—recursion is the backbone of the language. Computation in Datalog consists of monotonic logical inferences that apply to produce more facts until fixpoint. A Datalog rule "$\mathbf{C}(z, x) \leftarrow \mathbf{A}(x, y), \mathbf{B}(y, z)$." means that if $\mathbf{A}(x, y)$ and $\mathbf{B}(y, z)$ are both true, then $\mathbf{C}(z, x)$ can be inferred. Livshits et al. expressed their joint reflection and points-to analysis declaratively in Datalog, which is also a good vehicle for our illustration and further changes.

We consider the core of the analysis algorithm, which handles the most common features, illustrated in our above example: creating a reflective object representing a class (a *class object*) given a name string (library method `Class.forName`), creating a new object given a class object (library method `Class.newInstance`), retrieving a reflective method object given a class object and a signature (library method `Class.getMethod`), and reflectively calling a virtual method on an object (library method `Method.invoke`). This treatment ignores several other APIs, which are handled similarly. These include, for instance, fields, constructors, other kinds of method invocations (static, special), reflective access to arrays, other ways to get class objects, and more.

The domains of the analysis include: invocation sites, I; variables, V; heap object abstractions (i.e., allocation sites), H; method signatures, S; types, T; methods, M; natural numbers, N, and strings.

The analysis takes as input the relations (i.e., tables filled with information from the program text) shown in Figure 2. Using these inputs, the Livshits et al. reflection analysis can be expressed as a five-rule addition to any points-to analysis. The rest of the points-to analysis (not shown here—see e.g., [Guarnieri and Livshits 2009; Kastrinis and Smaragdakis 2013b; Whaley et al. 2005]) supplies more rules for computing a relation $\textsc{VarPointsTo}(v : V, h : H)$ and a relation $\textsc{CallGraphEdge}(i : I, m : M)$. Intuitively, the traditional points-to part of the joint analysis is responsible for computing how heap objects flow intra- and inter-procedurally through the program, while the added rules contribute only the reflection handling. We explain the rules below.

$\textsc{ClassObject}(i, t) \leftarrow$
  $\textsc{Call}(i, \texttt{"Class.forName"}), \textsc{ActualArg}(i, 0, p),$
  $\textsc{VarPointsTo}(p, c), \textsc{ConstantForClass}(c, t).$
$\textsc{VarPointsTo}(r, h) \leftarrow$
  $\textsc{ClassObject}(i, t), \textsc{ReifiedClass}(t, h),$
  $\textsc{AssignReturnValue}(i, r).$

The first two rules, above, work jointly: they model a `forName` call, which returns a class object given a string representing the class name. The first rule says that if the first argument (0-th parameter, since `forName` is a static method) of a `forName` call points to an object that is a string constant, then the type corresponding to that constant is retrieved and associated with the invocation site in computed relation $\textsc{ClassObject}$. The second rule then uses $\textsc{Clas-}$

CALL($i : I, s : string$): invocation instruction i in the program is a call to a method with signature s.

ACTUALARG($i : I, n : N, v : V$): at invocation instruction i, the n-th parameter is local variable v.

ASSIGNRETURNVALUE($i : I, v : V$): at invocation instruction i, the value returned from the call is assigned to local variable v.

HEAPTYPE($h : H, t : T$): object h has type t.

LOOKUP($sig : S, t : T, m : M$): inside type t there is a method m with method signature sig.

CONSTANTFORCLASS($h : H, t : T$): class/type t has a name represented by the constant string object h in the program text.

CONSTANTFORMETHOD($h : H, sig : S$): method signature sig has a name represented by the constant string object h in the program text.

REIFIEDCLASS($t : T, h : H$): special object h represents the class object of type t. Such special objects (i.e., unique identifiers representing them in the analysis) are created up-front and are part of the input.

REIFIEDHEAPALLOCATION($i : I, t : T, h : H$): special object h represents objects of type t that are allocated with a `newInstance` call at invocation site i.

REIFIEDMETHOD($sig : S, h : H$): special object h represents the reflection object for method signature sig.

Figure 2: Relations representing the input program and their informal meaning.

SOBJECT: if the result of the `forName` call at invocation site i is assigned to a local variable r, and the reflection object for the type associated with i is h, then r is inferred to point to h.

The above rules could easily be combined into one. However, their split form is more flexible. In later sections we will add more rules for producing **CLASSOBJECT** facts—for instance, instead of constant strings we will have expressions that still get inferred to resolve to an actual type.

VARPOINTSTO(r, h) ←
  CALL(i, `"Class.newInstance"`), **ACTUALARG**($i, 0, v$),
  VARPOINTSTO(v, h_c), **REIFIEDCLASS**(t, h_c),
  ASSIGNRETURNVALUE(i, r),
  REIFIEDHEAPALLOCATION(i, t, h).

The above rule reads: if the receiver object, h_c, of a `newInstance` call is a class object for class t, and the `newInstance` call is assigned to variable r, then make r point to the special allocation site h that designates objects of type t allocated at the `newInstance` call site.

VARPOINTSTO(r, h_m) ←
  CALL(i, `"Class.getMethod"`),
  ACTUALARG($i, 0, b$), **ACTUALARG**($i, 1, p$),
  ASSIGNRETURNVALUE(i, r),
  VARPOINTSTO(b, h_c), **REIFIEDCLASS**(t, h_c),
  VARPOINTSTO(p, c), **CONSTANTFORMETHOD**(c, s),
  LOOKUP($t, s, \_$), **REIFIEDMETHOD**(s, h_m).

The above rule gives semantics to `getMethod` calls. It states that if such a call is made with receiver b (for "base") and first argument p (the string encoding the desired method's signature), and if the analysis has already determined the objects that b and p may point to, then, assuming p points to a string constant encoding a signature, s, that exists inside the type that b points to ("_" stands for "any" value), the variable r holding the result of the `getMethod` call points to the reflective object, h_m, for this method signature.

CALLGRAPHEDGE(i, m) ←
  CALL(i, `"Method.invoke"`),
  ACTUALARG($i, 0, b$), **ACTUALARG**($i, 1, p$),
  VARPOINTSTO(b, h_m), **REIFIEDMETHOD**(s, h_m),
  VARPOINTSTO(p, h), **HEAPTYPE**(h, t),
  LOOKUP(t, s, m).

Finally, all reflection information can contribute to inferring more call-graph edges. The last rule encodes that a new edge can be inferred from the invocation site, i, of a reflective `invoke` call to a method m, if the receiver, b, of the `invoke` (0th parameter) points to a reflective object encoding a method signature, and the argument, p, of the `invoke` (1st parameter) points to an object, h, of a class in which the lookup of the signature produces the method m.

The above five rules are a small part of a realistic implementation of reflection handling, but they offer a faithful model of the core of the analysis—other additions handle more reflective calls and more language types (e.g., arrays) but represent engineering, rather than conceptual handling. Having declarative rules allows easy inspection of changes. Note how much of the logic relies on inter-procedural properties (i.e., **VARPOINTSTO** information), and at the same time produces inter-procedural properties (**VARPOINTSTO** and **CALLGRAPHEDGE**).

3. Techniques for Empirical Soundness

In our work, we define a more liberal reflection inference algorithm that attempts to improve in terms of empirical soundness. For instance, we are trying to reasonably over-approximate the values returned by a `forName` call—the main entry point for dynamic information. We next present our two major techniques.

3.1 Generalizing Reflection Inference via Substring Analysis

An important way of enhancing the empirical soundness of our analysis is via richer string flow. The logic discussed in

Section 2 only captures the case of entire string constants used as parameters to a `forName` call. The parameter of `forName` could be any string expression, however. It is interesting to attempt to deduce whether such an expression can refer to a class name. Similarly, strings representing field and method names are used in reflective calls—we already encountered the `getMethod` call in Section 2. As shown earlier, in the code example of Figure 1, method names are often constructed dynamically, with fixed prefixes, such as "`set`".

In order to estimate what classes, fields, or methods a string expression may represent, we implement substring matching: all strings constants in the program text are tested for prefix and suffix matching against all known class, method, and field names. (We use reasonable heuristics to limit the matches: member prefixes need to be at least 3 characters long, member suffixes at least 5 characters, class suffixes at least 6 characters long. These settings can easily vary but reflect a balance between expected usage and spurious matches.)

The strings that may refer to such entities are handled with more precision than others during analysis. For instance, a points-to analysis (e.g., in the DOOP or WALA frameworks) will typically merge most strings into a single abstract object—otherwise the analysis will incur an overwhelmingly high cost because of tracking numerous string constants. Strings that may represent class/interface, method, or field names are prevented from such merging. Furthermore, the flow of such strings through factory objects is tracked.

String concatenation in Java is typically done through objects of type `StringBuffer` or `StringBuilder`. The source-level concatenation primitive, operator `+`, reduces to operations through such factory objects. To evaluate whether reflection-related substrings may flow into factory objects, we leverage the points-to analysis itself, pretending that an object flow into an `append` method and out of a `toString` method is tantamount to an assignment. The logic is quite straightforward—a simplified version is in the rule below. (The rule assumes we have already computed relation **REFLECTIONOBJECT**$(h : H)$, which lists the string constants that partially match method, field, or class names, as described above. It also takes an extra input relation **STRINGFACTORYVAR**$(v_f : V)$ that captures which variables are of a string factory type.)

VARPOINTSTO$(r, h) \leftarrow$
  CALL$(i_a,$ `"append"`**)**,
  ACTUALARG$(i_a, 0, v_f)$, **ACTUALARG**$(i_a, 1, v)$,
  STRINGFACTORYVAR(v_f),
  CALL$(i_t,$ `"toString"`**)**,
  ACTUALARG$(i_t, 0, u_f)$, **ASSIGNRETURNVALUE**(i_t, r),
  VARPOINTSTO(v_f, h_f), **VARPOINTSTO**(u_f, h_f),
  VARPOINTSTO(v, h), **REFLECTIONOBJECT**(h).

In words: if a call to `append` and a call to `toString` are over the same factory object, h_f, (accessed by different vars, v_f and u_f, at possibly disparate parts of the program) then all the potentially reflection-related objects that are pointed to by the parameter, v, of `append` are inferred to be pointed by the variable r that accepts the result of the `toString` call.

In this way, the flow of partial string expressions through the program is tracked. By then appropriately adjusting the **CONSTANTFORCLASS** and **CONSTANTFORMETHOD** predicates of Section 2 (to also map from partial strings to their matching types) we can estimate which reflective entities can be returned at the site of a `forName` or `getMethod` call. In this way, the joint points-to and reflection analysis is enhanced with substring reasoning without requiring any changes to the base logic of Section 2. Namely, string flow through buffers becomes just an enhancement of the points-to logic, which is already leveraged by reflection analysis.

An interesting aspect of the above approach is that it is easily configurable, in commony desirable ways. Our above rule for handling partial string flow through string factory objects does not concern itself with how string factory objects (h_f) are represented inside the analysis. Indeed, string factory objects are often as numerous as strings themselves, since they are implicitly allocated on every use of the `+` operator over strings in a Java program. Therefore, a pointer analysis will often merge string factory objects.[2] The rule for string flow through factories is unaffected by this treatment. Although precision is lost if all string factory objects are merged into one abstract object, the joint points-to and reflection analysis still computes a fairly precise outcome: "does a partial string that matches some class/method-/field name flow into some string factory's `append` method, and does some string factory's `toString` result flow into a reflection operation?" If both conditions are satisfied, the class/method/field name matched by the partial string is considered to flow into the reflection operation.

3.2 Use-Based Reflection Analysis

Our second technique for statically analyzing reflection calls leverages the way objects returned by reflective calls are later used in the program. We call the approach *use-based reflection analysis* and it integrates two sub-techniques: a *back-propagation* mechanism and an *object invention* mechanism. We discuss these next.

3.2.1 Inter-procedural Back-Propagation

An important observation regarding reflection handling is that it is one of the few parts of a static analysis that are typically *under-approximate* rather than *over-approximate*. A static points-to analysis is primarily a *may* analysis: it computes a conservative over-approximation of the ana-

[2] For instance, this is enabled with the flag `SMUSH_STRINGS` in the WALA framework [Fink et al.] or the flag `MERGE_STRING_BUFFERS` in the DOOP framework. Both flags are on by default for precise (i.e., costly) analyses.

lyzed program's behavior. This is usually impossible to do in the presence of reflection: the analysis cannot know all the values that a string expression can assume. Of course, the analysis could over-approximate such values (e.g., assume that *any* string is possible) but such treatment is catastrophic for precision and scalability: a single reflective call would lead to vast imprecision propagating through the program. No practical whole-program analysis attempts such over-approximation [Livshits et al. 2013]. Instead, analyses choose to purposely treat reflective calls under-approximately: when the arguments of the reflection call are possible to infer, they are taken into account; other potential values are ignored.

Our first use-based reflection analysis technique back-propagates information from the use-site of a reflective result *to the original reflection call that got under-approximated*. Such an under-approximated call could be one of several:

- A `Class.forName` call, as seen earlier: returns a dynamic representation of a class, given a string.
- A `Class.get[Declared]Method` call, as seen earlier: returns a dynamic representation of a method, given a class and a string.
- A `Class.get[Declared]Field` call: returns a dynamic representation of a field, given a class and a string.
- A `Class.get[Declared]Methods` call: returns all (or all public) methods of a class.
- A `Class.get[Declared]Fields` call: returns all (or all public) fields of a class.

Let us consider again the example in the Introduction, showing how the use of a non-reflection object can inform a reflection call's analysis:

```
1 Class c1 = Class.forName(className);
2 ...      // c2 aliases c1
3 Object o1 = c2.newInstance();
4 ...      // o2 aliases o1
5 e = (Event) o2;
```

Typically (e.g., when `className` does not point to a known constant) the `forName` call will be under-approximated (rather than, e.g., assuming it will return any class in the system). The idea is to then treat the cast as a hint: it suggests that the earlier `forName` call should have returned a class object for `Event`. This reasoning, however, should be *inter-procedural* with an understanding of heap behavior. The above statements could be in distant parts of the program (separate methods) and aliasing is part of the conditions in the above pattern. Further, note that the related objects are twice-removed: we see a cast on an *instance* object and need to infer something about the `forName` site that *may* have been used to create the class that got used to allocate that object. This propagation should be as precise as possible: lack of precision will lead to too many class objects returned at the `forName` call site, affecting scalability.

Therefore, we see again the need to employ points-to analysis, this time in order to detect the relationship between cast sites and `forName` sites, so that the latter can be better resolved and we can improve the points-to analysis itself—a mutual recursion pattern. The high-level structure of our technique (for this pattern) is as follows:

- At the site of a `forName` call, create a special, placeholder object (of type `java.lang.Class`), to stand for all unknown objects that the invocation may return.
- The special object flows freely through the points-to analysis, taking full advantage of inter-procedural reasoning facilities.
- At the site of a `newInstance` invocation, if the receiver is our special object, the result of `newInstance` is also a special object (of type `java.lang.Object` this time) that remembers its `forName` origins.
- This second special object also flows freely through the points-to analysis, taking full advantage of inter-procedural reasoning facilities.
- If the second special object (of type `java.lang.Object`) reaches the site of a cast, then the original `forName` invocation is retrieved and augmented to return the cast type or its subtypes as class objects.

The algorithm for the above treatment can be elegantly expressed via rules that are mutually recursive with the base points-to analysis. The rules for the `forName`-`newInstance`-cast pattern are representative. We use extra input relations **ReifiedUnknownForName**$(i : I, h : H)$, and **ReifiedUnknownNewInstance**$(i : I, h : H)$, analogous to our earlier "**Reified...**" relations. The first relation gives, for each `forName` invocation site, i, a special object, h, that identifies the invocation site. The second relation gives a special object, h, that stands for all unknown objects returned by a `newInstance` call, which was, in turn, performed on the special object returned by a `forName` call, at invocation site i. The rules then become:

VarPointsTo$(v, h) \leftarrow$
  Call$(i,$ `"Class.forName"`**)**, **AssignReturnValue**(i, v),
  ReifiedUnknownForName(i, h).

In words: the variable that was assigned the result of a `forName` invocation points to the special object representing all missing objects from this invocation site. In this way, the special object can then propagate through the points-to analysis.

VarPointsTo$(r, h_n) \leftarrow$
  Call$(i_n,$ `"Class.newInstance"`**)**,
  ActualArg$(i_n, 0, v)$, **AssignReturnValue**(i_n, r),
  VarPointsTo(v, h),
  ReifiedUnknownForName(i, h),
  ReifiedUnknownNewInstance(i, h_n).

According to this rule, when analyzing a `newInstance` call, if the receiver is a special object that was produced by a `forName` invocation, i, then the result of the `newInstance` will be another special object (of appropriate type—determined by the contents of

ReifiedUnknownNewInstance) that will identify the original forName call.

The final rule uses input relation **CAST**($v' : V, v : V, t : T$) (with v' being the variable to which the cast result is stored and v the variable being cast) and **SUBTYPE**($t : T, u : T$) with its expected meaning:

CLASSOBJECT$(i, t') \leftarrow$
CAST$(\_, v, t)$, **SUBTYPE**(t', t), **VARPOINTSTO**(v, h_n),
REIFIEDUNKNOWNNEWINSTANCE(i, h_n).

The rule ties the logic together: if a cast to type t is found, where the cast variable points to a special object, h_n, then retrieve the object's forName invocation site, i, and infer that this invocation site returns a class object of type t', where t' is a subtype of t.

Other use-cases. As seen above, the back-propagation logic involves the result of several inter-procedural queries (e.g., points-to information at possibly distant call sites). In fact, there are use-based back-propagation patterns with even longer chains of reasoning. One such is below:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Constructor[] cons1 =
               c2.getConstructors(types);
...      // cons2 aliases cons1
Object o1 = cons2[i].newInstance(args);
...      // o2 aliases o1
e = (Event) o2;
```

In this case, the cast of o2 informs the return value of forName, three reflection calls back!

Interestingly, the back-propagation analysis can exploit not just cast information but also strings (including partial strings, transparently, per our substring/string-flow analysis of Section 3.1). When retrieving a member from a reflectively discovered class, the string name supplied may contain enough information to disambiguate what this class may be. Consider the pattern:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
Field f = c2.getField(fieldName);
```

In this case, the value of the fieldName string can inform the analysis result for the earlier forName call. We apply this idea to the 4 API calls Class.get[Declared]Method and Class.get[Declared]Field.

Contrasting approaches. Our back-propagating reflection analysis has some close relatives in the literature. Livshits et al. [2005]; Livshits [2006] also examined using future casts as hints for forName calls, as an alternative to regular string inference. Li et al. [2014] generalize the Livshits approach to many more reflection calls. There are, however, important ways in which our techniques differ:

- Our analysis is inter-procedural, whereas other approaches have several intra-procedural elements. In our earlier example, both the Livshits et al. and the Li et al. approaches require for the cast to post-dominate the newInstance call. Thus, the pattern becomes significantly less general:

```
Class c1 = Class.forName(className);
...      // c2 aliases c1
e = (Event) c2.newInstance();
```

The result of such a restriction is that the potential for imprecision is diminished, yet the ability to achieve empirical soundness is also scaled back. There are several cases where the cast will not post-dominate the intermediate reflection call, yet could yield useful information. This is precisely what Livshits et al. encountered experimentally—a direct quote illustrates:

> *The high number of unresolved calls in the JDK is due to the fact that reflection use in libraries tends to be highly generic and it is common to have 'Class.newInstance wrappers'—methods that accept a class name as a string and return an object of that class, which is later cast to an appropriate type in the caller method. Since we rely on intraprocedural post-dominance, resolving these calls is beyond our scope.* [Livshits et al. 2005]

Our approach aims for empirical soundness and uses other mechanisms for controlling the potential imprecision of the back-propagating analysis, as we will see in Section 3.3.

- The ability to back-propagate string information and not just cast information (i.e., exploit the use of get[Declared]{Method,Field} calls to resolve earlier forName calls) has not been exploited in other approaches. This feature also benefits from other elements of our overall analysis, namely substring matching and substring flow analysis (Section 3.1).

Precision vs. Scalability. A final note on the back-propagation technique concerns its precision and scalability. The fully inter-procedural tracking of reflection results is good for achieving empirical soundness: many possibilities are explored and the results are conservatively propagated to the original reflection site, from where they will flow down again to all other dependent reflection calls. However, this can introduce imprecision and, as a result, impact scalability. For instance, a cast may be to a type with numerous subtypes. If all of them are considered to be the result of a common forName call, then the back-propagation inference will be imprecise. If, in turn, this forName call has its result propagated to many program points, then the analysis precision and scalability are likely to suffer. Therefore, it is desirable to control when and how back-propagation will apply. We discuss this topic in Section 3.3, but first we consider alternatives to back-propagation.

3.2.2 Inventing Objects

Our second use-based reflection analysis technique is a forward propagation technique. It consists of inventing objects of the appropriate type at the point of a cast operation that has received the result of a reflection call. Consider again our `forName-newInstance`-cast example:

```
1 Class c1 = Class.forName(className);
2 ...      // c2 aliases c1
3 Object o1 = c2.newInstance();
4 ...      // o2 aliases o1
5 e = (Event) o2;
```

As discussed earlier, one issue with back-propagation is that its results may adversely affect precision. The information will flow back to the site of the `forName` call, and from there to multiple other program points—not just to the point of the cast operation (line 5), or even to the point of the `newInstance` operation (line 3) in the example.

The object invention technique offers the converse compromise. Whenever a special, unknown reflective object flows to the point of a cast, instead of informing the result of `forName`, the technique invents a new, regular object of the right type (`Event`, in this case) that starts its existence at the cast site. The "invented" object does not necessarily abstract actual run-time objects. Instead, it exploits the fact that a points-to analysis is fundamentally a may-analysis: it is designed to possibly yield over-approximate results, in addition to those arising in real executions. Thus, an invented value does not impact the correctness of the analysis (since having extra values in points-to sets is acceptable), yet it may enable it to explore possibilities that would not exist without the invented value. These possibilities are, however, strongly hinted by the existence of a cast in the code, over an object derived from reflection operations.

The algorithm for object invention in the analysis is again recursive with the main points-to logic. We illustrate for the case of `Class.newInstance`, although similar logic applies to reflection calls such as `Constructor.newInstance`, as well as `Method.invoke` and `Field.get`.

As in the back-propagating analysis, we use special placeholder objects. These are represented by input relations **ReifiedMarkerNewInstance**($i : I, h : H$), and **ReifiedInventedObject**($i : I, t : T, h : H$). The first relation gives, for each `newInstance` invocation site, i, a special object, h, that identifies the invocation site. The second relation gives an invented object, h of type t, for each `newInstance` invocation site, i, and type t that appears in a cast. The algorithm is captured by two rules:

VarPointsTo(v, h) ←
  Call(i, "`Class.newInstance`"),
  AssignReturnValue(i, v),
  ReifiedMarkerNewInstance(i, h).

That is, the variable assigned the result of a `newInstance` invocation points to a special object marking that it was produced by a reflection call. The marker object can then propagate through the points-to analysis.

The key part of the algorithm is to then invent an object at a cast site.

VarPointsTo(r, h) ←
  Cast(r, v, t), **VarPointsTo**(v, h_m),
  ReifiedMarkerNewInstance(i, h_m),
  ReifiedInventedObject(i, t, h).

In words, if a cast to a type t is found, and the variable, v, being cast points to an object marking that it was produced by a `newInstance` call, then the variable, r, storing the result of the cast, points to a newly invented object, with the right type, t.

Note that in terms of coverage (i.e., empirical soundness) the object invention approach is subsumed by the back-propagation analysis: if a type is inferred to be produced by an earlier `forName` call, it will flow down to the point of the cast, removing the need for object invention. Nevertheless, the benefit of object invention is that it allows selectively turning off back-propagation while still taking advantage of information from a cast.

3.3 Balancing for Scalability

As discussed in Section 3.2.1, the back-propagating analysis technique is powerful in terms of achieving empirical soundness (i.e., inferring a large number of potential results of a reflection call). At the same time, however, the technique may suffer in precision, since the result of a reflection call is deduced from far away information, which may be highly over-approximate. Conversely, the object invention technique is more precise (since the invented object only starts existing at the point of the cast) but may suffer in terms of soundness. Thus, it can be used to supplement back-propagation when the latter is applied selectively.

To balance the soundness/precision tradeoff of the back-propagating analysis, we employ precision thresholds. Namely, back-propagation is applied only when it is reasonably precise in terms of type information. For instance, if a cast is found, it is used to back-propagate reflective information only when there are up to a constant, c, class types that can satisfy the cast (i.e., at most c subtypes of the cast type). Intuitively, a cast of the form "`(Event)`" is much more informative when `Event` is a class with only a few subclasses, rather than when `Event` is an interface that many tens of classes implement. Similarly, if string information (e.g., a method name) is used to determine what class object could have been returned by a `Class.forName`, the back-propagation takes place only when the string name matches methods of at most d different types. This threshold approach minimizes the potential for noise back-propagating and polluting all subsequent program paths that depend on the original reflection call.

A second technique for employing back-propagation without sacrificing precision and scalability adjusts the flow of special objects (i.e., objects in REIFIEDUNKNOWNFORNAME or REIFIEDUNKNOWNNEWINSTANCE). Although we want such objects to flow inter-procedurally, we can disallow their tracking through the heap (i.e., through objects or arrays), allowing only their flow through local variables. This is consistent with expected inter-procedural usage patterns of reflection results: although such results will likely be returned from methods (cf. the quote from [Livshits et al. 2005] in Section 3.2.1), they are less likely to be stored in heap objects.

We employ both of the above techniques by default in our analysis (with $c = d = 5$). The user can configure their application through input options.

4. Evaluation

We implemented our techniques in the DOOP framework [Bravenboer and Smaragdakis 2009b], together with numerous engineering improvements (i.e., support for more API calls) to DOOP's original reflection handling. The ELF system [Li et al. 2014] implements similar functionality, yet without a focus on empirical soundness: ELF explicitly avoids inferring reflection call targets when it cannot fully disambiguate them.

We perform the default joint points-to and call-graph analysis of DOOP, which is an Andersen-style context-insensitive analysis, with full support for complex Java language features, such as class initialization, exceptions, etc.

The evaluation of our techniques aims to answer three research questions:

RQ1. *Do the presented techniques have reasonable running times?*

RQ2. *Can these techniques have an impact on the soundness of a points-to analysis?*

RQ3. *Does an increase in soundness incur a significant loss in precision?*

Experimental Setup. Our evaluation setting uses the LogicBlox Datalog engine, v.3.9.0, on a Xeon X5650 2.67GHz machine with only one thread running at a time and 24GB of RAM. We have used a JVMTI agent to construct a dynamic call-graph for each analyzed program.

We analyze 10 benchmark programs from the DaCapo 9.12-Bach suite [Blackburn et al. 2006], with their default inputs (for the purposes of the dynamic analysis). Other benchmarks could not be executed or analyzed: *tradebeans*/*tradesoap* from 9.12-Bach do not run with our instrumentation agent, hence no dynamic call-graphs can be extracted for comparison. This is a known, independently documented, issue (see `http://sourceforge.net/p/dacapobench/bugs/70/`). We have been unable to meaningfully analyze *fop* and *tomcat*—significant entry points were missed. This suggests either a packaging error at determining what makes up the application and library code of each benchmark (manual repackaging is necessary since no exact boundaries are provided by the DaCapo suite), or the extensive use of dynamic loading, which needs further special handling. We are planning to investigate such instances in the future.

We use Oracle JDK 1.7.0_25 for the analysis. To our knowledge, this is the most modern version of the JDK to have been used in the literature of scalable points-to analysis. In both respects (size of benchmarks and complexity of the JDK) we stress-test the analysis and our techniques, inevitably straining scalability. (For comparison, consider the quote from [Fink et al.] in the Introduction, referring to the inability to analyze realistic benchmarks with reflection under the smaller JDK 1.6.)

Empirical soundness metric. Any unsoundness is, in principle, bad, so a quantification is treacherous. However, following the example of other work [Ali and Lhoták 2012, 2013; Li et al. 2014; Stancu et al. 2014], we quantify the unsoundness of the static analysis in terms of missing call-graph edges, compared to a dynamic call-graph. We consider only call-graph edges originating from application code, since library classes contain a fair amount of non-analyzable native methods. We also filter out some missing edges:

- *Class Initializers.* DOOP only models *which* subset of classes get initialized (without any information about where the initializer gets called from). We filter out edges to class initializer methods, if static analysis indicates that the class has been initialized.
- *Native.* Native code cannot be analyzed. However, some library reflection calls are wrappers for native methods (e.g., `forName()` and `forName0()`). Edges to these methods are, thus, completely extraneous due to our special modeling of their effect.
- *Class Loader.* Method `loadClass()` is invoked by the VM when a class needs to be loaded and `checkPackageAccess()` is invoked right after loading.
- *Synthetic.* Edges involving dynamically generated classes are impossible to obtain by reflection analysis alone, so we eliminate such instances.

Results. Figure 3 plots the results of our experiments, combining both analysis time and empirical unsoundness (in call-graph edges). We compare our techniques to the ELF system [Li et al. 2014], which also attempts to improve reflection analysis for Java. Missing bars labeled "n/a" correspond to analyses that did not terminate in 90mins (5400sec). Each chart plots the missing dynamic call-graph edges that are not discovered by the corresponding static analysis. We use separate bars for the *application-to-application* and *application-to-library* edges.

We show five techniques:

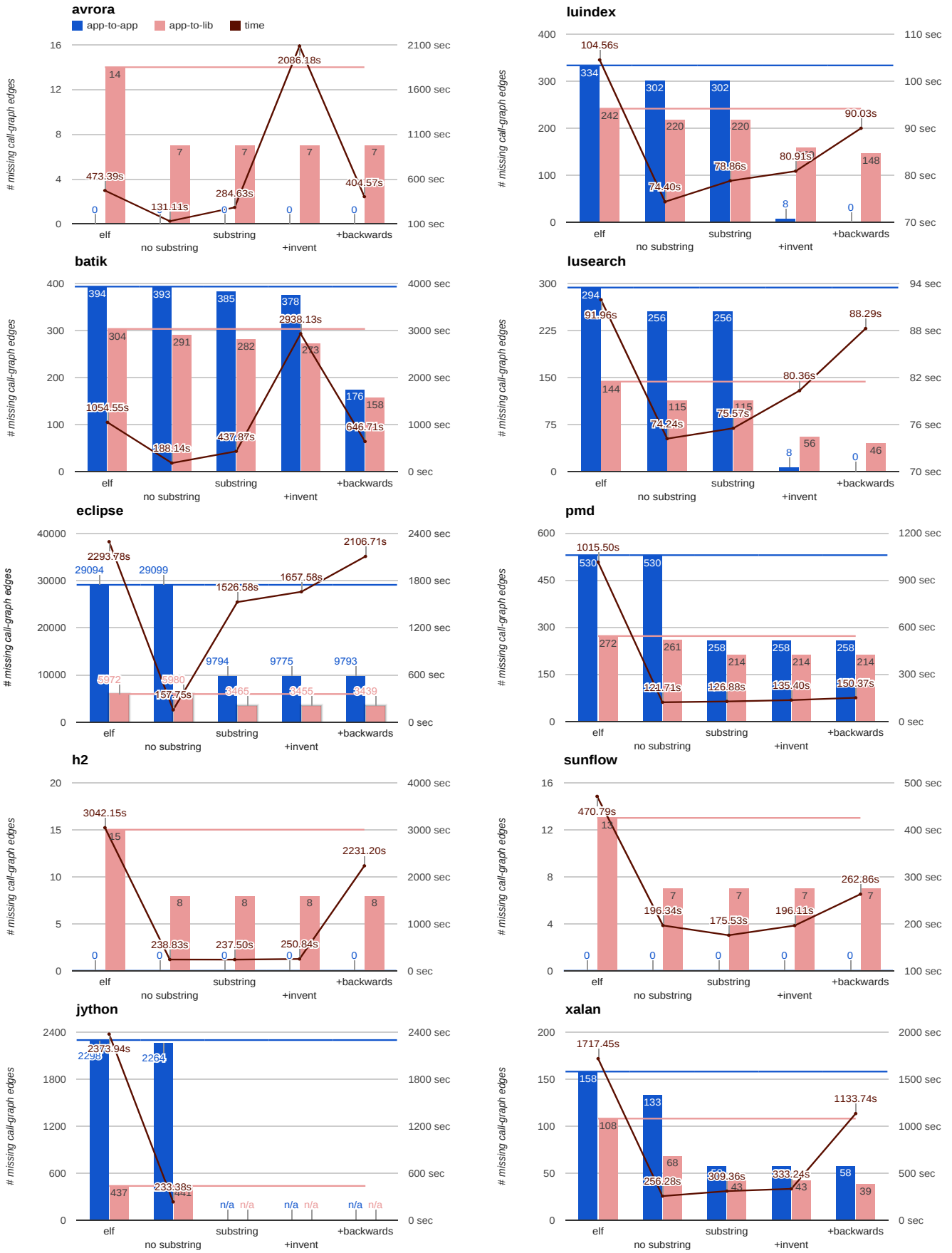

Figure 3: Unsoundness metrics and running time over the DaCapo benchmarks, for the Li et al. ELF system vs. our four techniques. The charts plot one point-and-line (for running time—scale on right axis) and two bars (for missing application-to-application and application-to-library dynamic call-graph edges—scale on left axis). *Lower is better for all metrics.* N/a entries correspond to non-termination.

Total Edges	Benchmarks									
Setting	avrora	batik	eclipse	h2	jython	luindex	lusearch	pmd	sunflow	xalan
elf	19355	31602	10191	38252	19709	4547	4209	8544	4223	35918
no substring	19379	31708	9032	35538	20537	4676	4352	8592	4251	35221
substring	20591	35314	115967	38107	n/a	4682	4362	9533	4285	45160
+invent	26586	47303	116635	38162	n/a	5773	5266	9557	4319	45343
+backwards	20677	37013	117576	43952	n/a	6115	5587	9577	4407	63746
dynamic	4165	8329	40026	4901	13583	3027	1845	4874	2215	6128

Figure 4: Total static and dynamic call-graph edges for the DaCapo 9.12-Bach benchmarks. These include only *application-to-application* and *application-to-library* edges.

1. *Elf.* This is the ELF reflection analysis [Li et al. 2014]. It serves as a baseline for the evaluation of the four settings of our own analysis.
2. *No substring.* This is our reflection analysis, with engineering enhancements over the original DOOP framework, but no analysis of partial strings or their flow.
3. *Substring.* The analysis integrates the substring and substring flow analysis of Section 3.1.
4. *+Invent.* This analysis integrates substring analysis as well as the object invention technique of Section 3.2.2.
5. *+Backwards.*[3] This analysis integrates substring analysis as well as the back-propagation technique of Section 3.2.1.

Our research questions can thus be answered:

RQ1: Do the techniques have reasonable running times? Although our approach has emphasized empirical soundness, it does not sacrifice scalability. All four of our settings were faster than *elf* for almost all benchmarks. Aside from *jython*, for which only the *elf* and *no substring* techniques were able to terminate before timeout, in all other cases *substring* and at least one of *+invent* or *+backwards* outperformed *elf*, while in 7-of-10 benchmarks *all* our techniques outperformed *elf*.

RQ2: Do our techniques impact soundness? In most benchmarks, more than half (to nearly all) of the missing *application-to-application* edges were recovered by at least one technique. The *application-to-library* missing edges also decreased, although not as much. This suggests that our techniques substantially increase the soundness of the analysis. In fact, the *eclipse* benchmark was hardly being analyzed in the past, since most of the dynamic call-graph was missing.

RQ3: Do the techniques sacrifice precision? Soundness could increase by computing a vastly imprecise call-graph. This is not the case for our techniques. Figure 4 lists the total static and dynamic edges being computed. On average, *+backwards* computes the most static edges (about 4.5 times the number of dynamic edges). On the lower end of the spectrum lies *no substring*, with a minimum of 3.4 times the number of dynamic edges being computed.

[3] Note that the *+Backwards* and *+Invent* techniques are independent and not comparable: they are both additions to the substring analysis, but neither includes the other.

5. Related Work

The traditional handling of reflection in static analysis has been through integration of user input or dynamic information. The Tamiflex tool [Bodden et al. 2011] exemplifies the state of the art. The tool observes the reflective calls in an actual execution of the program and rewrites the original code to produce a version without reflection calls. Instead, all original reflection calls become calls that perform identically to the observed execution. This is a practical approach, but results in a blend of dynamic and static analysis. Clearly, the greatest motivation for static analysis is to capture all possible program behaviors. It is unrealistic to expect that uses of reflection will always yield the same results in different dynamic executions—or there would be little reason to have the reflection (as opposed to static code) in the first place. Our approach attempts to restore the benefits of static analysis, with reasonable empirical soundness.

Interesting work on static treatments of reflection is often in the context of dynamic languages, where resolving reflective invocations is a necessity. Furr et al. [2009] offer an analysis of how dynamic features are used in the Ruby language. Their observations are similar to ours: dynamic features (reflection in our case) are often used either with sets of constant arguments (in order to avoid writing verbose, formulaic code), or with known prefixes/suffixes (e.g., to re-locate within the file system).

Madsen et al. [2013] employ a use-based analysis technique in the context of Javascript. When objects are retrieved from unknown code (typically libraries) the analysis infers the object's properties from the way it is used in the client. In principle, this is a similar approach to our use-based techniques (both object invention and back-propagation) although the technical specifics differ.

Ali and Lhoták [2012, 2013] offer comparisons of dynamic and static call-graph edge metrics. They discover hundreds of missing edges in several of the DaCapo 2006-10-MR2 benchmarks. However, their experiments do not integrate the vastly improved support for reflection (e.g., mod-

eling of `Object.getClass`) offered by ELF or our current work. Our experiments are substantially more representative in regards to the actual empirical soundness of a joint reflection and pointer analysis. Stancu et al. [2014] present an empirical study that compares profiling data with a points-to static analysis, but do not support reflection. They target only the most reflection-light benchmarks of the DaCapo 9.12-Bach suite (*avrora, luindex, and lusearch*), and patch the code to avoid reflection entirely.

6. Conclusions

Reflection is of key importance, yet very hard to handle in static analysis. We presented powerful techniques that elegantly extend declarative reasoning over reflection calls and inter-procedural object flow.

References

K. Ali and O. Lhoták. Application-only call graph construction. In *Proc. of the 26th European Conf. on Object-Oriented Programming*, ECOOP '12, pages 688–712. Springer, 2012. ISBN 978-3-642-31056-0. doi: 10.1007/978-3-642-31057-7_30.

K. Ali and O. Lhoták. Averroes: Whole-program analysis without the whole program. In *Proc. of the 27th European Conf. on Object-Oriented Programming*, ECOOP '13, pages 378–400. Springer, 2013. ISBN 978-3-642-39037-1. doi: 10.1007/978-3-642-39038-8_16.

S. M. Blackburn, R. Garner, C. Hoffmann, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. L. Hosking, M. Jump, H. B. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanovic, T. VanDrunen, D. von Dincklage, and B. Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Proc. of the 21st Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '06, pages 169–190, New York, NY, USA, 2006. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167488.

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. In *Proc. of the 33rd International Conf. on Software Engineering*, ICSE '11, pages 241–250, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0445-0. doi: 10.1145/1985793.1985827.

M. Bravenboer and Y. Smaragdakis. Exception analysis and points-to analysis: Better together. In *Proc. of the 18th International Symp. on Software Testing and Analysis*, ISSTA '09, pages 1–12, New York, NY, USA, 2009a. ACM. ISBN 978-1-60558-338-9. doi: 10.1145/1572272.1572274.

M. Bravenboer and Y. Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, New York, NY, USA, 2009b. ACM. ISBN 978-1-60558-766-0.

S. J. Fink et al. WALA UserGuide: PointerAnalysis. http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis.

M. Furr, J. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *Proc. of the 24th Annual ACM SIGPLAN Conf. on Object Oriented Programming, Systems, Languages, and Applications*, OOPSLA '09, pages 283–300, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-766-0. doi: 10.1145/1640089.1640110.

S. Guarnieri and B. Livshits. GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In *Proc. of the 18th USENIX Security Symposium*, SSYM' 09, pages 151–168, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855768.1855778.

G. Kastrinis and Y. Smaragdakis. Efficient and effective handling of exceptions in Java points-to analysis. In *Proc. of the 22nd International Conf. on Compiler Construction*, CC '13, pages 41–60. Springer, 2013a. ISBN 978-3-642-37050-2. doi: 10.1007/978-3-642-37051-9_3.

G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '13, New York, NY, USA, 2013b. ACM. ISBN 978-1-4503-2014-6.

M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, and C. Unkel. Context-sensitive program analysis as database queries. In *Proc. of the 24th Symp. on Principles of Database Systems*, PODS '05, pages 1–12, New York, NY, USA, 2005. ACM. ISBN 1-59593-062-0. doi: 10.1145/1065167.1065169.

Y. Li, T. Tan, Y. Sui, and J. Xue. Self-inferencing reflection resolution for Java. In *Proc. of the 28th European Conf. on Object-Oriented Programming*, ECOOP '14, pages 27–53. Springer, 2014. ISBN 978-3-662-44201-2.

P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '11, pages 590–601, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0663-8. doi: 10.1145/1993498.1993567.

B. Livshits. *Improving Software Security with Precise Static and Runtime Analysis*. PhD thesis, Stanford University, December 2006.

B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, APLAS '05, pages 139–

160. Springer, 2005. ISBN 3-540-29735-9. doi: 10.1007/11575467_11.

B. Livshits, M. Sridharan, Y. Smaragdakis, and O. Lhoták. In defense of unsoundness. *PLDI FIT 2013 presentation*, http://www.soundiness.org/, June 2013.

M. Madsen, B. Livshits, and M. Fanning. Practical static analysis of JavaScript applications in the presence of frameworks and libraries. In *Proc. of the ACM SIGSOFT International Symp. on the Foundations of Software Engineering*, FSE '13, pages 499–509. ACM, 2013. ISBN 978-1-4503-2237-9. doi: 10.1145/2491411.2491417.

M. Naik, A. Aiken, and J. Whaley. Effective static race detection for java. In *Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '06, pages 308–319, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1134018.

T. W. Reps. Demand interprocedural program analysis using logic databases. In R. Ramakrishnan, editor, *Applications of Logic Databases*, pages 163–196. Kluwer Academic Publishers, 1994.

Y. Smaragdakis, M. Bravenboer, and O. Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proc. of the 38th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0490-0.

C. Stancu, C. Wimmer, S. Brunthaler, P. Larsen, and M. Franz. Comparing points-to static analysis with runtime recorded profiling data. In *Proc. of the 2014 International Conf. on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools*, PPPJ '14, pages 157–168. ACM, 2014. ISBN 978-1-4503-2926-2. doi: 10.1145/2647508.2647524.

J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the 2004 ACM SIGPLAN Conf. on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM. ISBN 1-58113-807-5. doi: 10.1145/996841.996859.

J. Whaley, D. Avots, M. Carbin, and M. S. Lam. Using Datalog with binary decision diagrams for program analysis. In *Proc. of the 3rd Asian Symp. on Programming Languages and Systems*, APLAS '05, pages 97–118. Springer, 2005. ISBN 3-540-29735-9. doi: 10.1007/11575467_8.