More Sound Static Handling of Java Reflection

Yannis Smaragdakis[1], George Balatsouras[1], George Kastrinis[1], and Martin Bravenboer[2]

[1] University of Athens, Greece
[2] LogicBlox Inc., Atlanta, GA, USA

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. 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.

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 needs to either conservatively over-approximate (e.g., assume that *any* class can be accessed), or 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 programs of realistic size. It is telling that *no practical Java program analysis framework in existence handles reflection soundly* [19], although other language features are modeled soundly.[3]

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

The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-26529-2_26

Full soundness is not practically achievable, but it can still be approximated for the well-behaved reflection patterns encountered in regular, non-adversarial programs. Therefore, it makes sense to treat soundness as a continuous quantity: something to improve on, even though we cannot perfectly reach. To avoid confusion, we use the term *empirical soundness* for the quantification of how much of the dynamic behavior the static analysis covers. Computable metrics of empirical soundness can help quantify how close an analysis is to the fully sound result. Based on such metrics, one can make comparisons (e.g., "more sound") to describe soundness improvements.

The second challenge of handling reflection in a static analysis is *scalability*. The online documentation of the IBM Wala library [8] 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.*

In this paper, we describe an approach to analyzing reflection in the Java points-to analysis setting. Our approach requires no manual configuration and achieves significantly higher empirical soundness without sacrificing scalability, for realistic benchmarks and libraries (DaCapo Bach and Java 7). In experimental comparisons with the recent Elf system [16] (itself improving over the reflection analysis of the Doop framework [6]), 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. Concretely, our work:

- introduces key techniques in static reflection handling that contribute greatly to empirical soundness. The techniques generalize past work from an intra-procedural to an inter-procedural setting and combine it with a string analysis;
- shows how scalability can be addressed with appropriate tuning of the above generalized techniques;
- thoroughly quantifies the empirical soundness of a static points-to analysis, compared to past approaches and to a dynamic analysis;
- is implemented and evaluated on top of an existing open framework (Doop [6]).

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. [18,20]. 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:

```
String className = ... ;
Class c = Class.forName(className);
Object o = c.newInstance();
String methodName = ... ;
Method m = c.getMethod(methodName, ...);
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 [5, 6, 10, 13–15, 17, 22, 23, 25, 26], 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 is representative and 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 1. 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., [10, 14, 25]) supplies more rules for computing a

Call$(i : I, s : string)$: instruction i is an invocation to a method with signature s.
ActualArg$(i : I, n : N, v : V)$: at invocation i, the n-th parameter is local var v.
AssignRetValue$(i : I, v : V)$: at invocation i, the value returned is assigned to local variable v.
HeapType$(h : H, t : T)$: object h has type t.
Lookup$(sig : S, t : T, m : M)$: in type t there exists method m with 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 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.

Fig. 1: Relations representing the input program and their informal meaning.

relation **VarPointsTo**$(v : V, h : H)$ and a relation **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.

ClassObject$(i, t) \leftarrow$
  Call$(i,$ `"Class.forName"`$)$, **ActualArg**$(i, 0, p)$,
  VarPointsTo(p, c), **ConstantForClass**(c, t).
VarPointsTo$(r, h) \leftarrow$
  ClassObject(i, t), **ReifiedClass**(t, h), **AssignRetValue**(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 **ClassObject**. The second rule then uses **ClassObject**: if the result of the `forName` call at instruction 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$(r, h) \leftarrow$
  Call$(i,$ `"Class.newInstance"`$)$, **ActualArg**$(i, 0, v)$, **VarPointsTo**(v, h_c),
  ReifiedClass(t, h_c), **AssignRetValue**(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 (i.e., invented) allocation site h that designates objects of type t allocated at the `newInstance` call site.

VarPointsTo$(r, h_m) \leftarrow$
  Call$(i,$ `"Class.getMethod"`$)$, **ActualArg**$(i, 0, b)$, **ActualArg**$(i, 1, p)$,
  AssignRetValue(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$(i, m) \leftarrow$
  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.

3 Techniques for Empirical Soundness

We next present our main techniques for higher empirical soundness.

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.

In order to estimate what classes, fields, or methods a string expression may represent, we implement *substring matching*: all string constants in the program text are tested for prefix and suffix matching against known class, method, and field names. (We use tunable thresholds to limit the matches: e.g., member prefixes, resp. suffixes, need to be at least 3, resp. 5, characters long.)

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 `StringBuffer` or `StringBuilder` objects. The common concatenation operator, `+`, reduces to calls over 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. A simplified version of the logic 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$(i_a, \texttt{"append"})$, **ActualArg**$(i_a, 0, v_f)$, **ActualArg**$(i_a, 1, v)$,
  StringFactoryVar(v_f), **Call**$(i_t, \texttt{"toString"})$, **AssignRetValue**$(i_t, r)$,
  ActualArg$(i_t, 0, u_f)$, **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. 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 commonly 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, with the appropriate user-selectable flag.[4] The rule for string flow through factories is unaffected by

[4] E.g., `SMUSH_STRINGS` in WALA [8] and `MERGE_STRING_BUFFERS` in DOOP.

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 is called *use-based reflection analysis* and it integrates two sub-techniques: a *back-propagation* mechanism and a (forward) *object invention* mechanism.

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* [19]. 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 can be a `Class.forName`, `Class.get[Declared]Method`, `Class.get[Declared]Field`, etc. call, which returns a dynamic representation of a class, method, or field, given a string name.

The example below, which we will refer to repeatedly in later sections, shows 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 marker 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 **ReifiedForName**$(i : I, h : H)$, and **ReifiedNewInstance**$(i : I, h : H)$, 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$(i, \texttt{"Class.forName"})$, **AssignRetValue**$(i, v)$, **ReifiedForName**$(i, h)$.

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$(i_n, \texttt{"Class.newInstance"})$, **ActualArg**$(i_n, 0, v)$, **VarPointsTo**$(v, h)$,
  AssignRetValue(i_n, r), **ReifiedForName**(i, h), **ReifiedNewInstance**(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 `ReifiedNewInstance`) that will identify the `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), **ReifiedNewInstance**(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. In the case below, the cast of `o2` informs the return value of `forName`, three reflection calls back!

```
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;
```

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. [18,20] also examined using future casts as hints for `forName` calls, as an alternative to regular string inference. Li et al. [16] generalize the Livshits approach to many more reflection calls. There are, however, important ways in which our techniques differ:

- Our analysis generalizes the pattern significantly. In our earlier example, from the beginning of this section, both the Li et al. and the Livshits et al. approaches require for the cast to not only occur in the same method as the `newInstance` call but also to post-dominate it! This restricts the pattern to an intra-procedural and fairly specific setting, reducing its generality:

```
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.* [20]

- We generalize back-propagation to string information and not just cast information (i.e., we exploit the use of `get[Declared]{Method,Field}` calls to resolve earlier `forName` calls). This feature also benefits from other elements of our overall analysis, namely substring matching and substring flow analysis (Section 3.1). For instance, by having more information on what are the possible strings passed to a `getMethod` call, we are more likely to determine the return value of a `getClass`, on which the `getMethod` was called.

Inventing Objects. Our approach introduces an alternative use-based reflection analysis technique, which works as a *forward propagation* technique (in contrast to the earlier back-propagation). 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 usual `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;
```

A major issue with our earlier back-propagation technique 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 will enable it to explore possibilities that might 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 marker 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$(v, h) \leftarrow$
  Call$(i,$ `"Class.newInstance"`$)$, **AssignRetValue**(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$(r, h) \leftarrow$
  Cast(r, v, t), **VarPointsTo**(v, h_m),
  ReifiedMarkerNewInstance(i, h_m), **ReifiedInventedObject**(i, t, h).

In words, if a variable, v, is cast to a type t and points to a marker object that 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 empirical soundness the object invention approach is weaker than 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. (Conversely, inventing objects at the cast site will not catch all cases covered by back-propagation, since the special object of the back-propagation analysis may never flow to a cast.) Nevertheless, back-propagation is often less scalable. Thus, the benefit of object invention is that it allows to selectively turn off back-propagation while still taking advantage of information from a cast.

3.3 Balancing for Scalability

Consider again our inter-procedural back-propagating analysis technique relative to prior, intra-procedural techniques. Our approach explicitly aims for empirical soundness (i.e., to infer all 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, our 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 **ReifiedForName** or **ReifiedNewInstance**). 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 [20] in Section 3.2), 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 [6], together with numerous improvements (i.e., complete API support) to Doop's reflection handling. Following the Elf study [16], 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. Our techniques are orthogonal to the context-sensitivity used, and can be applied to all analyses in the Doop framework. In general, nothing in our modeling of reflection limits either context- or flow-sensitivity.

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 [3], 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 (no application-library boundaries are provided by the DaCapo suite), or the extensive use of dynamic loading, which needs further special handling.

We use Oracle JDK 1.7.0_25 for the analysis. (For comparison, consider the quote from [8] in the Introduction, refers to the smaller JDK 1.6.)

Empirical soundness metric. We quantify the empirical unsoundness of the static analysis in terms of missing call-graph edges, compared to the dynamic call-graph. Call-graph construction is one of the best-known clients of points-to analysis [1,2,16] and has the added benefit of quantifying how much code the analysis truly reaches. We compare the call-graph edges found by our static analysis to a dynamic call-graph—a comparison also found in other recent work [24]. For a sound static analysis, no edge should occur dynamically but not predicted statically. However, this is not the case in practice, due to the unsound handling of dynamic features, as discussed in the Introduction.

Results. Figure 2 plots the results of our experiments, combining both analysis time and empirical unsoundness (in call-graph edges). 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. Library-to-library edges are also computed but they are not comparable in static vs. dynamic analysis due to native calls. We filter out edges to implicit methods (static initializers, `loadClass()`) that are not statically modeled. We show five techniques:

1. *Elf.* This is the ELF reflection analysis [16], which also attempts to improve reflection analysis for Java.
2. *No substring.* 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.
5. *+Backwards.*[5] This analysis integrates substring analysis as well as the back-propagation technique of Section 3.2.

It is important to note that, by design, our techniques do not enhance the precision of an analysis, only its empirical soundness. Thus, the techniques only find *more* edges: they cover more of the program. This improvement appears as a reduction in the figures ("lower is better") only because the number plotted is the *difference* in the missing edges compared to the dynamic analysis.

As can be seen, our techniques substantially increase the soundness of the analysis. In most benchmarks, more than half (to nearly all) of the missing *application-to-application* edges are recovered by at least one technique. The *application-to-library* missing edges also decreased, although not as much. In fact, the *eclipse* benchmark was hardly being analyzed in the past, since most of the dynamic call-graph was missing.

[5] The *+Backwards* and *+Invent* techniques are both additions to the substring analysis, but neither includes the other.

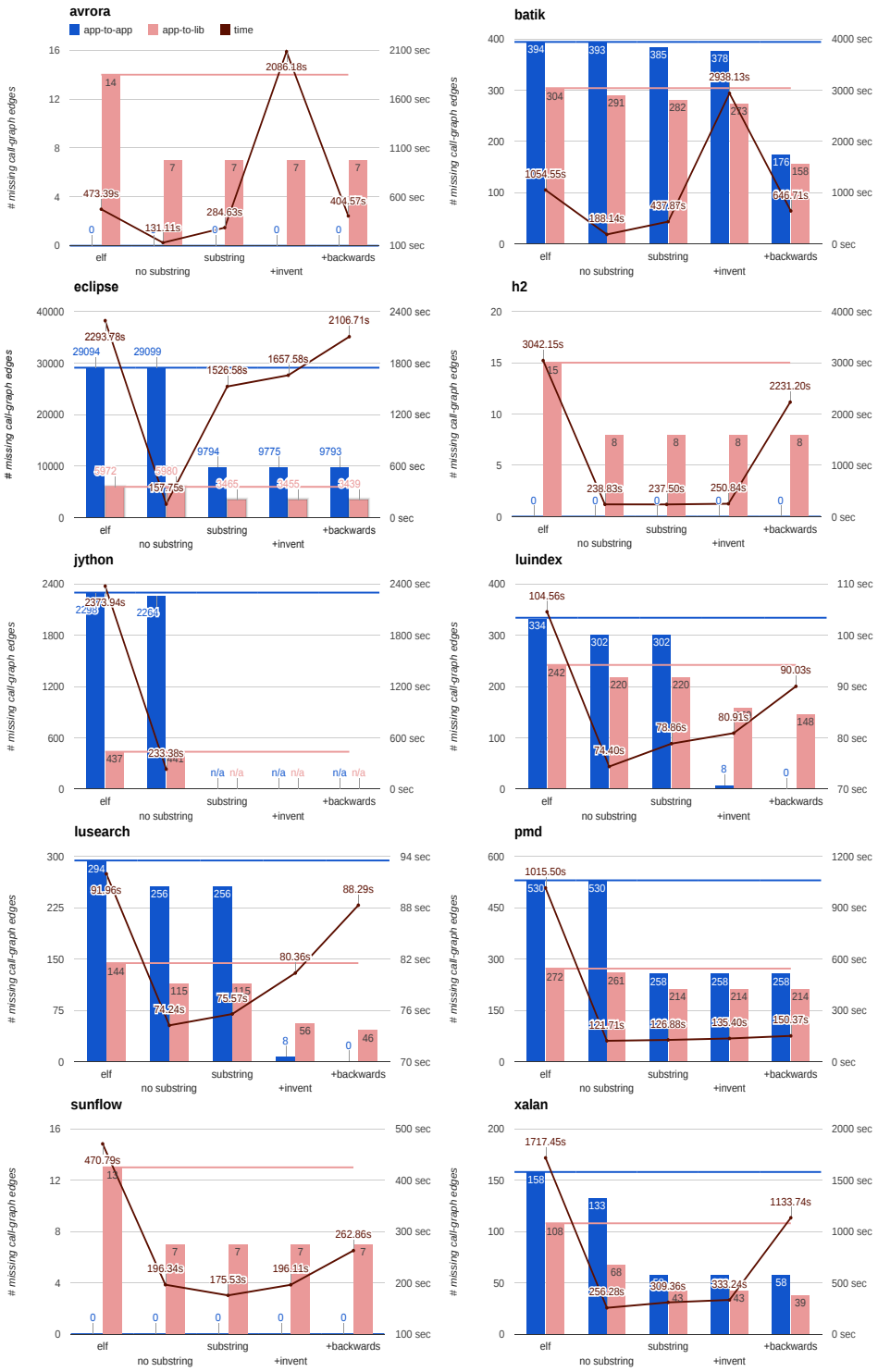

Fig. 2: Unsoundness metrics (two bars: missing call-graph edges app-to-app and app-to-lib) and analysis time (line) over the DaCapo benchmarks. *Lower is better for all.* For missing bars ("n/a"), the analysis did not terminate in 90mins.

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

Fig. 3: 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.

Furthermore, although our approach emphasizes empirical soundness, it does not sacrifice scalability. All four of our settings are faster than Elf for almost all benchmarks. Aside from *jython*, for which only the Elf and *no substring* techniques are 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. This is due to achieving scalability using the threshold techniques of Section 3.3 instead of by sacrificing some empirical soundness, as Elf does. (A major design feature of Elf is that it explicitly avoids inferring reflection call targets when it cannot fully disambiguate them.)

For completeness, we also show a sanity-checking metric over our analyses. Empirical soundness could increase by computing a vastly imprecise call-graph. This is not the case for our techniques. Figure 3 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.

In pragmatic terms, a user of our analysis should use flags to pick the technique that yields more soundness without sacrificing scalability, for the given input program. This is a familiar approach—e.g., it also applies to picking the exact flavor and depth of context-sensitivity.

5 Related Work

The traditional handling of reflection in static analysis has been through integration of user input or dynamic information. The Tamiflex tool [4] 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. 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.

An alternative approach is that of Hirzel et al. [11, 12], where an online pointer analysis is used to deal with reflection and dynamic loading by monitoring their run-time occurrence, recording their results, and running the analysis again, incrementally. This approach is quite interesting when applicable. However, maintaining and running a precise static analysis during program run time is often not realistic (e.g., for expensive context-sensitive analyses). Furthermore, the approach does not offer the off-line soundness guarantees one may expect from static analysis: it is not possible to ask questions regarding all methods that may ever be called via reflection, only the ones that have been called so far.

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. [9] 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 or with known prefixes/-suffixes (e.g., to re-locate within the file system).

Madsen et al. [21] 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. The conceptual precursor to both approaches is the work on reflection by Livshits et al. [18, 20], which has been extensively discussed and contrasted throughout the paper (see Sections 2, and 3.2).

Advanced techniques for string analysis have been presented by Christensen et al. [7]. They analyze complex string expressions and abstract them via a context-free grammar that is then widened to a regular language. Reflection is one of their examples but they only apply it to small benchmarks.

Stancu et al. [24] empirically compare profiling data with a points-to static analysis. However, they target only the most reflection-light benchmarks of the DaCapo 9.12-Bach suite and patch the code to avoid reflection entirely.

6 Conclusions

Highly dynamic features, such as reflection and dynamic loading, are the bane of static analysis. These features are not only hard to analyze well, but also ubiquitous in practice, thus limiting the practical impact of static analysis. We presented techniques for static reflection handling in Java program analysis. Our techniques build on top of state-of-the-art handling of reflection in Java, by elegantly extending declarative reasoning over reflection calls and inter-procedural object flow. Our main emphasis has been in achieving higher empirical soundness, i.e., in having the static analysis truly model observed dynamic behaviors.

Although full soundness is infeasible for a realistic analysis, it is possible to produce general techniques that enhance the ability to analyze reflection calls.

Although our techniques improve on the problem of handling reflection, further work is necessary to achieve good scalability and empirical soundness for complex programs. Furthermore, our work has not addressed another major and commonly used dynamic feature: dynamic loading. Continued work will hopefully make such language features a lot more feasible to analyze statically.

Acknowledgments. We gratefully acknowledge funding by the European Research Council under grant 307334 (SPADE).

References

1. Ali, K., Lhoták, O.: Application-only call graph construction. In: Proc. of the 26th European Conf. on Object-Oriented Programming. pp. 688–712. ECOOP '12, Springer (2012)
2. Ali, K., Lhoták, O.: Averroes: Whole-program analysis without the whole program. In: Proc. of the 27th European Conf. on Object-Oriented Programming. pp. 378–400. ECOOP '13, Springer (2013)
3. Blackburn, S.M., Garner, R., Hoffmann, C., Khan, A.M., McKinley, K.S., Bentzur, R., Diwan, A., Feinberg, D., Frampton, D., Guyer, S.Z., Hirzel, M., Hosking, A.L., Jump, M., Lee, H.B., Moss, J.E.B., Phansalkar, A., Stefanovic, D., VanDrunen, T., von Dincklage, D., Wiedermann, B.: 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. pp. 169–190. OOPSLA '06, ACM, New York, NY, USA (2006)
4. Bodden, E., Sewe, A., Sinschek, J., Oueslati, H., Mezini, M.: Taming reflection: Aiding static analysis in the presence of reflection and custom class loaders. In: Proc. of the 33rd International Conf. on Software Engineering. pp. 241–250. ICSE '11, ACM, New York, NY, USA (2011)
5. Bravenboer, M., Smaragdakis, Y.: Exception analysis and points-to analysis: Better together. In: Proc. of the 18th International Symp. on Software Testing and Analysis. pp. 1–12. ISSTA '09, ACM, New York, NY, USA (2009)
6. Bravenboer, M., Smaragdakis, Y.: 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, ACM, New York, NY, USA (2009)
7. Christensen, A.S., Møller, A., Schwartzbach, M.I.: Precise analysis of string expressions. In: Proc. of the 10th International Symp. on Static Analysis. pp. 1–18. SAS '03, Springer (2003)
8. Fink, S.J., et al.: WALA UserGuide: PointerAnalysis. `http://wala.sourceforge.net/wiki/index.php/UserGuide:PointerAnalysis`
9. Furr, M., An, J.D., Foster, J.S.: 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. pp. 283–300. OOPSLA '09, ACM, New York, NY, USA (2009)
10. Guarnieri, S., Livshits, B.: GateKeeper: mostly static enforcement of security and reliability policies for Javascript code. In: Proc. of the 18th USENIX Security Symposium. pp. 151–168. SSYM' 09, USENIX Association, Berkeley, CA, USA (2009), `http://dl.acm.org/citation.cfm?id=1855768.1855778`

11. Hirzel, M., von Dincklage, D., Diwan, A., Hind, M.: Fast online pointer analysis. ACM Trans. Program. Lang. Syst. 29(2) (2007)
12. Hirzel, M., Diwan, A., Hind, M.: Pointer analysis in the presence of dynamic class loading. In: Proc. of the 18th European Conf. on Object-Oriented Programming. pp. 96–122. ECOOP '04, Springer (2004)
13. Kastrinis, G., Smaragdakis, Y.: Efficient and effective handling of exceptions in Java points-to analysis. In: Proc. of the 22nd International Conf. on Compiler Construction. pp. 41–60. CC '13, Springer (2013)
14. Kastrinis, G., Smaragdakis, Y.: Hybrid context-sensitivity for points-to analysis. In: Proc. of the 2013 ACM SIGPLAN Conf. on Programming Language Design and Implementation. PLDI '13, ACM, New York, NY, USA (2013)
15. Lam, M.S., Whaley, J., Livshits, V.B., Martin, M.C., Avots, D., Carbin, M., Unkel, C.: Context-sensitive program analysis as database queries. In: Proc. of the 24th Symp. on Principles of Database Systems. pp. 1–12. PODS '05, ACM, New York, NY, USA (2005)
16. Li, Y., Tan, T., Sui, Y., Xue, J.: Self-inferencing reflection resolution for Java. In: Proc. of the 28th European Conf. on Object-Oriented Programming. pp. 27–53. ECOOP '14, Springer (2014)
17. Liang, P., Naik, M.: Scaling abstraction refinement via pruning. In: Proc. of the 2011 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 590–601. PLDI '11, ACM, New York, NY, USA (2011)
18. Livshits, B.: Improving Software Security with Precise Static and Runtime Analysis. Ph.D. thesis, Stanford University (December 2006)
19. Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.Y.E., Guyer, S.Z., Khedker, U.P., Møller, A., Vardoulakis, D.: In defense of soundiness: A manifesto. Commun. ACM 58(2), 44–46 (Jan 2015), `http://doi.acm.org/10.1145/2644805`
20. Livshits, B., Whaley, J., Lam, M.S.: Reflection analysis for Java. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 139–160. APLAS '05, Springer (2005)
21. Madsen, M., Livshits, B., Fanning, M.: 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. pp. 499–509. FSE '13, ACM (2013)
22. Naik, M., Aiken, A., Whaley, J.: Effective static race detection for java. In: Proc. of the 2006 ACM SIGPLAN Conf. on Programming Language Design and Implementation. pp. 308–319. PLDI '06, ACM, New York, NY, USA (2006)
23. Reps, T.W.: Demand interprocedural program analysis using logic databases. In: Ramakrishnan, R. (ed.) Applications of Logic Databases, pp. 163–196. Kluwer Academic Publishers (1994)
24. Stancu, C., Wimmer, C., Brunthaler, S., Larsen, P., Franz, M.: 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. pp. 157–168. PPPJ '14, ACM (2014)
25. Whaley, J., Avots, D., Carbin, M., Lam, M.S.: Using Datalog with binary decision diagrams for program analysis. In: Proc. of the 3rd Asian Symp. on Programming Languages and Systems. pp. 97–118. APLAS '05, Springer (2005)
26. Whaley, J., Lam, M.S.: 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. pp. 131–144. PLDI '04, ACM, New York, NY, USA (2004)