Efficient Reflection String Analysis via Graph Coloring

Neville Grech
Dept. of Informatics and Telecommunications, University of Athens, Greece
and Dept. of Computer Science, University of Malta, Malta
me@nevillegrech.com

George Kastrinis
Dept. of Informatics and Telecommunications, University of Athens, Greece
gkastrinis@di.uoa.gr

Yannis Smaragdakis
Dept. of Informatics and Telecommunications, University of Athens, Greece
yannis@smaragd.org

Abstract
Static analyses for reflection and other dynamic language features have recently increased in number and advanced in sophistication. Most such analyses rely on a whole-program model of the flow of strings, through the stack and heap. We show that this global modeling of strings remains a major bottleneck of static analyses and propose a compact encoding, in order to battle unnecessary complexity. In our encoding, strings are maximally merged if they can never serve to differentiate class members in reflection operations. We formulate the problem as an instance of graph coloring and propose a fast polynomial-time algorithm that exploits the unique features of the setting (esp. large cliques, leading to hundreds of colors for realistic programs). The encoding is applied to two different frameworks for string-guided Java reflection analysis from past literature and leads to significant optimization (e.g., a $\sim 2x$ reduction in the number of string-flow inferences), for a whole-program points-to analysis that uses strings.

2012 ACM Subject Classification Software and its engineering → Compilers, Theory of computation → Program analysis, Software and its engineering → General programming languages

Keywords and phrases reflection, static analysis, graph coloring

Digital Object Identifier 10.4230/LIPIcs.ECOOP.2018.26

Acknowledgements We gratefully acknowledge funding by the European Research Council, grant 307334 (SPADE), a Facebook Research and Academic Relations award, and an Oracle Labs collaborative research grant. In addition, the research work disclosed is partially funded by the REACH HIGH Scholars Program – Post-Doctoral Grants. The grant is part-financed by the European Union, Operational Program II, Cohesion Policy 2014-2020 (Investing in human capital to create more opportunities and promote the wellbeing of society - European Social Fund).

1 Introduction

Reflection is a language feature that enables the dynamic discovery of an object’s type structure (e.g., its fields and supported methods) and full access to the object’s state and functionality through dynamically-discovered members. Reflection is not merely one of the dynamic features of a statically-typed language but typically the backbone connecting all dynamic features. For instance, in Java, the most common facility for dynamic code
generation is the *dynamic proxy* pattern [26] (recently estimated to appear in 21% of open-source Java programs [17]), which requires the use of reflection in order to provide a generic implementation of an interface.

Modeling reflection has been a challenge for the static analysis of languages like Java and C#. Ignoring reflection operations during static analysis is one of the top causes of analysis unsoundness [22]. (Or, equivalently, one of the most common assumptions in any claim of soundness for an analysis is the lack of reflection.) Modeling reflection operations statically has attracted much recent research effort [7, 18, 19, 21, 23, 30, 34]. Virtually all of these models have a similar general structure, first explored by Livshits et al. [21, 23]: they model reflection in the context of a whole-program value-flow analysis (such as a points-to analysis, with a full model of the heap) so that the flow of parameters of reflective actions can be approximated. The initial such parameters are merely string values. For instance, a call to `java.lang.Class.getMethod()` takes as argument the name of the method to be looked up dynamically. By examining the flow of string values (which may partially match class, method, or field names) into reflective calls, the analysis can do a first approximation of the effects of reflective actions. This model can then be refined with extra information, such as the use patterns of objects produced via reflection.

Static analysis for reflection, therefore, crucially depends on modeling the flow of string *constants*. For instance, consider a string constant "put" that flows into a string concatenation operation (possibly with statically-unknown parameters), whose result flows to a `getMethod` call. The constant string yields significant information as to which method(s) may be selected dynamically. Such information is typically the differentiator between an infeasibly imprecise static analysis and one that can reliably guess an overapproximation of reflection results.

Tracking strings through a whole-program analysis can be very expensive, however. Type filtering is ineffective, since the `String` type is not elaborated into more detailed subtypes in most languages. This observation holds for a vast array of whole-program static analyses and is surprising in scope. For instance a context-insensitive analysis of the avrora DaCapo-Bach benchmark in the DOOP framework [30] computes a points-to set of 2.9 million strings and just 2 million regular objects. Similar effects are found throughout other static analyses that model reflection: a 0-1-CFA analysis with reflection support on the IBM WALA library [7], over a medium-sized benchmark (`antlr`, from the DaCapo 2006 suite), yields a total points-to set with 6.7 million strings and just 1.7 million non-strings objects (i.e., all other object types together). This effect is surprising: an analysis that only incidentally models the flow of strings (in order to model reflection operations) ends up being dominated by string values, in comparison to all other objects whose flow is the real analysis target.

In this paper, we propose a technique for collapsing string constants so that they impose minimal overhead in a whole-program value-flow analysis, yet retain their ability to act as member selectors for all reflection operations. Specifically, we model the problem of merging *string constants* as a graph coloring problem. Two strings cannot be merged if they can be used as selectors of distinct class members in the same reflection operation. This is denoted by making the strings be neighbor nodes in a *conflict* (a.k.a. *interference*) graph, which we then attempt to color. Any coloring of the graph yields different values for any two neighbors, i.e., conflicting string constants. Colors are then used as values in the whole-program static analysis, instead of string constants. In this way, a color can designate any of a finite set of merged strings.

The graph coloring approach has several nuances, both theoretical and experimental. First, our setting suggests the need for a very fast (certainly polynomial time) coloring algorithm, which can, however, tolerate suboptimal coloring results: The optimal coloring...
still yields numbers of colors up to several hundreds, due to the existence of large cliques
in the interference graph. (The cliques are due to conflicts between all members of a class,
including all members declared in supertypes, all the way up to java.lang.Object. A deep
class hierarchy can easily result in string constants being able to select several hundreds of
members from the same class object.) We present a fast, near-linear-time, coloring algorithm
that performs very well in this setting. As a second consideration, string merging can
theoretically introduce some imprecision, due to spurious data flows in the static analysis.
We find experimentally that no imprecision arises in practice, strongly validating the appeal
of the string merging insight.

The string merging approach is orthogonal to other reflection analysis, string interning,
etc. techniques commonly employed in the literature. Concretely, string merging complements
any standard reflection analysis algorithm: the same reflection algorithm still applies, but
for fewer abstract string constants. Similarly, the technique is agnostic to how compactly
strings are represented at the low level.

We apply the string merging approach to the Doop and SOLAR program analysis
frameworks, which employ different static analyses for reflection [19,30]. The technique
yields a size reduction of $\sim 2x$ for points-to sets with string values, or a reduction of $\sim 1.5x$
in the total sizes of computed value sets. This translates to proportional savings for any
further analyses as well as a speedup of $\sim 20\%$ for the base points-to analysis. Importantly,
these improvements are orthogonal and assumption-less, shrinking the input of an analysis
and transparently benefiting any analysis algorithm, setup, or analyzed program, with no
pitfalls or drawbacks in other respects.

In all, our work makes the following contributions:

- It identifies a rather surprising problem in whole-program analysis that models reflection:
  string values feature disproportionately in value-flow results, such as points-to sets.
- It proposes the merging of strings that cannot differentiate members in the same class,
  and formulates the problem as an instance of graph coloring.
- It shows that a simple but fast graph coloring algorithm is well-suited to the specifics of
  the problem instance.
- It validates the approach in standard analysis frameworks and quantifies the benefit.

## 2 Illustration and Intuition

We illustrate (in simplified terms) the insight that our approach exploits, with the example
of Figure 1.

A program contains several (10) string constants (“bar”, “fro”, etc.), which can be used
as selectors for fields and methods of all available classes. An undirected conflict graph is
produced, where two constants are neighbors if they are substrings of the names of different
fields or different methods of the same class. (In the full setting, class members also include
all supertype members. However, in this example no type has a non-trivial supertype and
members of the trivial superclass java.lang.Object are ignored for the sake of simplicity.) A
single string constant has the potential to be used via reflection to select members of different
classes—e.g., “bar” can potentially be used to select either Foob:bar() or Tolk:barahir().
Neither possibility can be precluded a priori, since it is the analysis of value flow itself that
will determine how these strings get concatenated with others and to which reflection calls
they flow.

By coloring the conflict graph (in the standard “graph coloring” sense, of different colors
for neighbors), we can merge string constants together without losing any of their ability
to be used as selectors in reflection operators. In the example, an optimal graph coloring
Figure 1 Illustration of approach: sample classes (top) are accessed via reflection, from a program with 10 (sub)string constants (middle). This induces a conflict graph (bottom): two string constants conflict if they can refer to distinct members of the same type (i.e., method or field) inside the same class. The graph is 2-colorable, so just 2 values can be kept instead of the initial 10. An example coloring is shown: bold vs. non-bold strings.

uses just two colors—e.g., consider the bold and non-bold strings as having different colors. Any whole-program static analysis can then be performed with merely two artificial string constant objects, one for each color, in place of the original string constants. The first object effectively denotes either "bar" or "fro" or "bara" or "frotz", while the second stands for either "zork" or "gand" or "foo" or "baz" or "alf" or "gru". Subsequent analysis of a reflection operation will then proceed as before, e.g., knowing that either "bar" or "fro" or "bara" or "frotz" is used as an argument to a getField() call on an Info class object is as good as knowing which exact string constant was used: only field "frotz" can be selected.

In this way, the analysis needs to track the flow of a lot fewer values. The end results of a realistic whole-program analysis stop being unduly dominated by string objects.

There are several subtleties in the general approach. The idea of merging strings that cannot refer to members of the same class is simple in dynamic execution terms. In static analysis, however, there are some complicating factors that we explore in the next sections. Briefly:

- Although most published static analyses for reflection employ common base reasoning (Section 3) there are significant differences. Some algorithms track the flow of substrings (known as substring analysis), others only track strings that completely match class names or member names. Some algorithms use backward reasoning based on information other
than string matches (e.g., examining the further uses of the return value of a reflective operation). The approach is mostly orthogonal to these variations but may need slight adaptation to avoid accidental interference. Additionally, not all string constants should be merged. For instance, string constants that match class names should not be merged: they are the primary selectors in reflection logic. Section 5 discusses such topics in more detail.

Due to independent static analysis imprecision, string merging is not guaranteed to produce identical results as an original analysis that uses full string constants. A string constant can be spuriously computed to flow to a reflection operation over an incompatible class object. Before merging, the spurious flow could fail to find a matching member, whereas after merging it may do so. Such accidental imprecision, although uncommon, can be solved – Section 5 describes solutions as applied to two reflection analysis frameworks. In our experiments (Section 6), we have not found string merging to introduce any practical imprecision in static analysis: precision metrics, such as methods called by each site, remain identical.

3 Static Analysis with Reflection

The context of our work is a standard scheme that integrates Java reflection reasoning in a static analysis that tracks the flow of objects inter-procedurally. We discuss it first (Section 3.1) and then present variations (Section 3.2).

3.1 Inter-Procedural Reflection Analysis

Most static analyses for reflection [18,19,21,23,30,34] employ variations of a scheme originally due to Livshits et al. [21,23]. This standard scheme is based on computing the flow of objects in mutual recursion with the computation of the effects of reflection actions. A value-flow analysis (e.g., a standard points-to analysis, for all reference variables in a program) gives information to the reflection analysis and vice versa. This interplay of analyses is necessary because the different elements of reflective actions can be distributed throughout the program.

Consider a typical pattern of reflection use:

```java
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. This suggests 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.

This mutual recursion is typically captured in simple logical rules in the Datalog language. Datalog is ideal in that a) its computation model is based on the recursive specification of logical relations; b) it has already been used in a large amount of static analysis research work (e.g., [4,11,14,16,20,25,27,29,32,33]).
V is a set of variables
H is a set of heap object abstractions
S is a set of method signatures (including name)
I is a set of instructions (e.g., invocation sites)
T is a set of class types
N is the set of natural numbers

\[ \text{CALL}(i: I, sig: S) \] # instruction \( i \) is a call \( \text{sig}(\ldots) \).
\[ \text{ASSIGNRetValue}(i: I, v: V) \] # instruction \( i \) is a return \( v \).
\[ \text{ACTUALArg}(i: I, n: N, v: V) \] # the \( n \)-th parameter of call instruction \( i \) is local var \( v \).
\[ \text{HEAPType}(h: H, t: T) \] # object \( h \) has type \( t \).
\[ \text{LOOKUP}(\text{sig}: S, t: T, \text{m}: M) \] # in type \( t \) there is a method \( m \) with signature \( \text{sig} \).
\[ \text{CONSTANTForClass}(h: H, t: T) \] # string object \( h \) matches name of class/type \( t \).
\[ \text{CONSTANTForMethod}(h: H, \text{sig}: S) \] # string object \( h \) matches name of method \( \text{sig} \).
\[ \text{REIFIEDClass}(t: T, h: H) \] # special object \( h \) represents the class object of type \( t \).
\[ \text{REIFIEDObject}(i: I, t: T, h: H) \] # special object \( h \) represents objects of type \( t \) allocated with a \text{newInstance} call at invocation site \( i \).
\[ \text{REIFIEDMethod}(\text{sig}: S, h: H) \] # special object \( h \) represents the reflection object for method signature \( \text{sig} \).

\[ \text{VARPointsTo}(v, h) \] \text{\textarrow} \[ \text{CALL}(i, "\text{Class.forName"}, \text{ACTUALArg}(i, 0, p), \text{ASSIGNRetValue}(i, r), \text{VARPointsTo}(p, c), \text{CONSTANTForClass}(c, t), \text{REIFIEDClass}(t, h)). \]

The first rule models a \text{Class.forName} call, which returns a class object given a string representing the class name. The rule says that if the first argument (0-th parameter, since \text{Class.forName} is a static method) of a \text{Class.forName} call points to an object that is a string constant, then the types, \( t \), that match that constant are retrieved. Assuming the result of the \text{Class.forName} call at instruction \( i \) is assigned to a local variable \( r \), and the reflection object for a \( t \) is \( h \), \( r \) is inferred to point to \( h \).
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.

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.

Finally, all reflection information can contribute to inferring more call-graph edges. The last rule encodes that an 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 method $m$.

3.2 Variations

Several enhancements and variations of this general reflection analysis scheme have been employed in past work. We summarize them below, since we will need to refer to them in later sections.

- Complex mechanisms for substring flow and matching can be used, instead of matching full class and method names. Consider the first of the previous rules, handling `Class.forName` calls. The rule uses predicate `ConstantForClass`, which could well encode a substring match instead of a full match of the class name. The complication will then be to propagate strings through concatenation operations, so that the `VarPointsTo` relation (the main value-flow relation of the analysis) computes the flow of substrings throughout the program. (That is, when a string constant is in a points-to set, this signifies that the analysis computes that the run-time value is the result of concatenating some prefix and suffix to the string constant.) This tends to put more pressure on the size of the points-to set as string information is allowed to flow through more avenues.

- The base rules show a forward analysis, where string values need to be completely determined for the result of a reflective operation to be modeled. Every one of the four rules is predicated on a past `VarPointsTo` result that establishes that a parameter points to a certain string that matches other rule conditions. An alternative is to analyze how the results of reflective operations are used in further code—i.e., to perform a backward analysis. There are many sources of such backward analysis information [18,19,30]. For
a simple case [21], if the result of a `c.newInstance` call is cast to a type `T`, then `T` gives a strong hint on the value of reflective class object `c`. This is particularly useful when `c` has been produced by using external sources (e.g., strings from a file or the network) so that its value cannot be determined by normal forward analysis. The same technique can also be used to get higher precision, by cross-validating the inferences of forward and backward reflection analysis before allowing them to affect the rest of the analysis.

The base scheme shown (see 3rd rule of previous section) uses strings as method selectors only when the analysis has determined the containing class. This may not be necessary, however: a string matching a method name may be descriptive enough to determine both the method and the containing class. Such reasoning is typically performed under further qualifications of precision. For instance, the rule may only fire if the method name matches very few methods in the whole program and/or if the method name is a constant that is close to the reflective operation (e.g., in the same method), to avoid imprecision.

### 4 String Merging via Coloring

Our approach consists of merging string constants that occur in the program text. As illustrated in Section 2, strings can be merged if they cannot serve to distinguish members of the same class. This agrees with the main forward logic of static reflection analysis, as seen in the 3rd rule of Section 3.1: a string denoting a member name is only used for known class objects. (Exceptions are discussed in Section 5.)

The string-merging approach operates before inter-procedural analysis is performed. Effectively, the analysis input is pre-processed so that the domain `H` of heap object abstractions gets shrunk: abstract objects representing string constants are merged, while all others remain unchanged. The inter-procedural reflection and value-flow (i.e., points-to) analysis then proceeds unchanged, over the optimized domain. The pre-processing also entails correctly updating all input predicates (e.g., `ConstantForMethod`) so that a merged string value can serve to select all members that its constituents can represent.

The more interesting aspects of the technique concern how the string-merging decision is made. As we saw, the problem can be viewed as an instance of standard graph coloring, on an undirected interference/conflict graph with nodes representing all string constants in the program text. Two strings are neighbors iff they match distinct members (of the same kind, i.e., both fields or both methods) in the same class.

Optimal graph coloring is an NP-hard problem. There are, however, strong reasons why our setting is a good fit for algorithms that yield more colors (i.e., do suboptimal merging) but are very fast.

The minimum number of colors required is large. The input graph contains large cliques of nodes, because of classes with several hundreds of members. Such classes arise due to inheritance hierarchies, since inherited members of a class need to be taken into account for reflective lookups. All strings matching members of such a class form a clique: any two of them are connected in the interference graph, so all of them need to be kept distinct. Even just considering classes in the Java system libraries, cliques of size in the hundreds arise.

Based on this observation, the best that an optimal algorithm can hope to achieve is a reduction of string constant values from the several thousands (as typical in a large Java program) to the few hundreds. Therefore, a sub-optimal coloring can still capture a lot of the benefit—even twice as many colors as the optimal number represent a substantial reduction in the number of string constants that the static analysis needs to track throughout the program code.
The benefit from string merging is not proportional to the reduction in the number of string constants. The benefit is, instead, reflected well by the reduction of the size of the \texttt{VarPointsTo} relation. For instance, if the number of string constants tracked by the analysis drops by a factor of 100 (i.e., on average 100 original string constants are mapped to each color and merged) the ensuing reduction in the complexity of the string-tracking analysis will be much more modest—typically well below 10. Benefits arise only when merged strings would have been inferred to be members of the same points-to set. However, most points-to sets in an analysis are small, containing at most a handful of members. In the worst case, if a variable were to point to a single string constant in the original analysis, no benefit would arise: the string constant would merely be replaced by a merged representative, but the points-to set would still be of size 1.

Per the above, the benefit of optimal graph coloring is tiny in our setting. Conversely, the speed of the coloring algorithm is crucial. String merging (i.e., graph coloring) is a pre-processing step, whose running time burdens the static analysis itself.

Accordingly, we employ a near-linear-time greedy algorithm for coloring the interference graph of string constants. The coloring (or “numbering”) algorithm specification is simple:

1. Compile an undirected graph \( G = \{V, E\} \) where \( V \) represents the original string constants and \( E \) represents the conflicts between string constants.
2. Apply any total ordering relation \( \leq \) defined over \( V \) to each edge \( E \) and direct each edge according to this. This produces a directed acyclic graph \( G' = \{V, E'\} \).
3. For each \( v \in V \), its color is established by computing the maximum distance from any root (i.e., node in \( V \) with zero in-degree) in \( G' \).

In practice the ordering \( \leq \) of strings can be arbitrary (e.g., lexicographic, or numeric by internal index, or random id) and an efficient implementation to establish the maximum distance to a root is to examine strings in a topological order over \( G' \). Therefore each string \( s \) in \( V \) is examined only after all its predecessors \( t \). We give string \( s \) color (i.e., number) \( i \), where \( i \) is one higher than the maximum color of any predecessor.

The numbering step trivially gives different numbers to conflicting string constants: for every two conflicting nodes, one will be below the other in the total ordering, so they cannot have the same maximum distance from a root node. Using a standard implementation of topological sorting with a min-heap data structure, the resulting algorithm runs in time \( O(e \cdot \log(n)) \), where \( e = |E| \) (the number of graph edges) and \( n = |V| \) (the number of string constants). (Each edge needs to be traversed upon examination of its source node, in order to update the color bound of the edge’s target node.)

In the worst case, this numbering algorithm would yield suboptimal colorings—e.g., if the greatest element of one clique, per the \( \leq \) ordering, is the least element of the next, the two cliques cannot reuse colors. However, such adversarial input is unlikely if one picks ordering \( \leq \) randomly. Even a lexicographic ordering \( \leq \) of strings yields consistently good results due to the nature of our setting: the strings are used for reflection analysis, therefore they need to match original, unobfuscated names of class members. (Reflection on obfuscated member names is not possible, since all string manipulation—e.g., concatenation or substring matching—gets invalidated.) Human-written member names are distributed fairly well lexicographically, so that different classes have members at dispersed points in the global ordering.

Figure 3 shows the above logic in Datalog with stratified aggregation (all ranges of min/max aggregators are computed before the aggregation needs to take place). We reuse relations from Section 3.1 and again only refer to methods—extending to also include field
members is trivial. The final rule performs an arbitrary aggregation/choice of a representative string constant, among the ones in the input, to stand for all string constants with the same color in the reduced input domain, $H$. This is a simplified version of the logic used in our actual implementation.

Note that the straightforward realization of these rules in a Datalog engine will likely have worst-case quadratic complexity, instead of the $O(e \cdot \log(n))$ bound for the algorithm implemented with a heap data structure. In practice, this effect can be mitigated with standard Datalog optimization techniques. Although the \texttt{LessThan} relation is worst-case quadratic, it is also local, since a string will only conflict with a small number of others, i.e., up to a constant. We can define a more economical intermediate relation, \texttt{ImmediatelyLessThan}, based on \texttt{LessThan}, and compute \texttt{ColorAtLeast} more efficiently using it.

With either a direct implementation or minimal optimization effort of Datalog rules, the running time of the algorithm for sets of string constants in realistic Java applications is virtually instantaneous, i.e., entirely negligible compared to subsequent analysis time.

\section{5 Practical Applications of Technique}

The essence of our technique, described in the previous section, is a good illustration of the principles, but it ignores several realistic semantic complications. Additional development is needed to integrate this technique to existing real world analyses without sacrificing soundness and precision.
5.1 Combining Forward with Backward Analyses to Counter Imprecision

String merging, when combined with (unavoidable) static analysis imprecision, is not guaranteed to produce identical results as an original analysis that uses full string constants. String, Method or Class constants could be spuriously computed to flow to a reflection operation producing statically inferred behavior that was not meant to happen at runtime. Before merging string constants, the spurious flow could fail to find matching spurious members, whereas after merging it may do so. Here, backward analyses [18,19,21,30] come into play to correct virtually all precision issues.

```java
String a = "zork"; // i.e. {gru, alf, baz, foo, gand, zork};
Class cls = unknown() ? Foob.getClass() : Info.getClass();
Method m = cls.getMethod(a); // m = zork or baz ?
String s = (String) m.invoke(); // m = zork!
```

In the example above, in the original program, variable `a` is assigned to string "zork" at line 1. Assuming a class structure as presented in Figure 1, our technique substitutes "zork" with another object that represents any of the following strings: "gru", "alf", "baz", "foo", "gand", "zork". At line 3, a conditional assignment with unknown predicate causes the static analysis to consider that `m` could either be class `Foob` or class `Info`. At line 5, in the original program we get method `zork` from the classes pointed by variable `cls`. Unfortunately, in the transformed program, the representative of "zork" matches both `Info.zork` and `Foob.baz`. Although some imprecision is introduced here, the analysis has means to reverse this. Since the method in `m` is invoked in the next line and the return value is cast, the analysis infers that `m` would not contain `Foob.baz` but just `Info.zork`, which is the only of the two methods that returns a `String`. (The astute reader will note that this is not a 100% sound treatment, however real world reflection analysis tools need to manage and balance precision, soundness and scalability.) In this way, a backward analysis serves the role of cross-validating forward analysis results to negate imprecision. Similarly, since `Info.zork` is only defined in class `Info`, the backward analysis also informs the forward analysis of class constants to infer that variable `cls` only contains class `Info`.

In practice, backward analyses like the ones demonstrated in this example are necessary to maintain a precise analysis whether or not our string coloring technique is applied, and indeed both state-of-the-art static reflection analysis frameworks on which our technique was applied (DOOP [4] and SOLAR [19]) enable these enhancements by default.

5.2 High Confidence Inferences

Although in practice most reflection inferences involve forward and backward analyses, this is not always the case. In DOOP, string constants that originate locally and flow to a reflective operation sink locally are treated as high-confidence inferences, and thus do not require confirmation from backward analysis. For instance, we can take the following example:

```java
String a = in.readline(); // not statically known
Class cls = Class.forName(a);
Method m = cls.getMethod("zork"); // cls must be Info for this to work
```

In the original program, we are not able to statically determine the string passed to variable `a` on line 1. However, if the analysis can determine with high confidence that a
method in this unknown class matches "zork" then this inference is used to determine that cls points to the class Info. Merging "zork" with other strings interferes with this mechanism. By nature, high confidence inferences need to be carried out under strict, syntactically apparent conditions, thus limiting their applicability. These same conditions can also be picked up in the string merging pre-analysis. For instance, in DOOP, only strings that originate in the same method where a matching reflective operation is performed are used for high confidence inferences. A solution we implemented for this scenario is to perform a more selective string merging—if a string can flow to a local reflective operation, that string is not merged.

5.3 Selective Unsoundness

Selective unsoundness in the design of static reflection analysis can also cause challenges to our technique. For instance, typical reflection analyses exclude strings that are not meaningful enough to determine class names. A common substring (for instance “Impl”) can match several classes. Using such strings to resolve class names naturally leads to imprecision in the analysis. A sensible heuristic in this case is to not perform static reflection analysis on strings that match more than some arbitrary number of classes. When strings are represented by their color, each color encodes multiple strings, matching methods or fields. (Note that a string can sometimes match both a method and a class.) Therefore the string coloring technique is at odds with the selective unsoundness heuristic: a merged string may be filtered out in other analysis reasoning. In this case, the solution we adopted was to not merge string constants if they can match classes. Alternatively, one can include strings that match classes to conflict graphs and have these all in the same clique—each color at most would represent one string that matches a class.

Design decisions and heuristics similar to these are present throughout real-world reflection analyses, which implies that any analysis optimization that may introduce imprecision could also introduce unsoundness.

6 Evaluation

We implemented our string coloring algorithm and applied our general technique to the most recent development version of the DOOP [4] static analysis framework. We have also applied our technique to the SOLAR pointer and reflection analysis framework [19]. Both DOOP and SOLAR are full-featured and handle most complex semantic aspects of the Java language, such as reflection, implicit initialization, exceptions, and more.

On both frameworks, we only needed to perform minimal modifications to their logic to apply this technique. Most of the modifications that we made are optimizations and additional indexing in the reflection logic. These simple optimizations were applied after we discovered that the additional load on the backward analyses necessitated better indexes. These modifications were applied to both the baseline configuration and the string coloring configuration, so both configurations benefited from these performance improvements. On both frameworks, we compare the performance of the analysis with and without string coloring enabled.

This evaluation intends to answer the following research questions:

RQ1 Does the presented technique enhance the efficiency of static analysis?
RQ2 Does the technique compromise the quality of the static analysis? In terms of:
   RQ2.1 Soundness.
   RQ2.2 Precision.
RQ3 Does our fast string coloring algorithm perform as-predicted, in terms of coloring effectiveness and its translation to string-merging effectiveness?

To answer these research questions, we employ the following metrics:

**Var points-to.** The size of the VarPointsTo relation, on both application and library code. This metric strongly correlates with relevant static analysis time. This is by far the largest relation that is produced as output by the analysis, and describes what stack variables point to. The cost of any further use of analysis results is likely to be highly correlated to the size of VarPointsTo.

**Heap points-to.** The cumulative size of all heap relations, i.e., instance field points-to, static field points-to and array index points-to. These form the second largest relation produced by the analysis, and describes what heap objects point-to.

**Relevant static analysis time.** The time required to run our static pointer analysis. This includes the time to run the graph coloring algorithm and all associated overheads when this is enabled.

**Call graph size.** We compare the sizes of call graphs before and after our optimization is applied. A smaller call graph indicates unsoundness, while a larger indicates imprecision and can answer RQ2.

**Var points-to string.** The size of the var points-to relation subset containing only strings as heap objects.

**Size of largest clique.** The size of the largest clique in the string conflict graph.

**Number of colors.** The total number of colors applied to the string conflict graph.

The metrics are established through the use of existing tools, applied to benchmarks from the DaCapo 9.12-Bach Java benchmark suite [2] in the case of the DOOP framework. In the case of the SOLAR framework, we use the same subset of the DaCapo 2006 benchmarks used in the original SOLAR evaluation [19]. Both static analysis frameworks use the PA-Datalog engine, a publicly available, stripped-down version of the commercial LogicBlox Datalog engine. Both frameworks are run with full-featured static handling of reflection. All our run times are established on an idle machine with an Intel Xeon E5-2687W v4 3.00GHz with up to 512 GB of RAM. All experiments had a cutoff time of 6 hours. (This is merely a practical time-budgeting limit. We have occasionally let several instances of heavy analyses run for longer but have not observed analyses that terminate in under 10 hours if they do not terminate in 6.) Timings reported are from a single run, but repeat runs show very low variation (up to about 5%, typically much lower).

Notably, all analyses operate on an already economical representation of string constants. Strings are interned and represented in all relations via a 22-bit identifier. (This means that the maximum string pool size is $2^{22} = 4194304$, which is still three orders of magnitude larger than the number of string constants arising in our benchmarks, as we shall see in Figure 10.) Furthermore, all experiments are conducted with all other standard string merging approaches enabled: all strings that cannot be used for reflection (i.e., that do not match class, method, or field names) are merged into a single, nondescript abstract string.

The DOOP framework is flexible with respect to context sensitivity, so we configure it with several flavors:

- **insens:** No context sensitivity.
- **1call:** A 1-call-site sensitive analysis (heap insensitive). This is also known as 1-CFA [28].
- **2type+H:** A 2-type-sensitive analysis [31] with a 1-type-sensitive heap.
- **2obj+H:** A 2-object sensitive analysis with 1-object sensitive heap.
The SOLAR framework is configured to use a selective 2-type-sensitive+heap. This adds call-site sensitivity for static methods to a 2-type-sensitive analysis. This is the kind of context sensitivity that SOLAR is tuned for [19].

6.1 RQ1: Performance and Efficiency Gains

As shown in Figure 4, the technique achieves an average analysis speedup of about 20% on the Doop framework. The running time includes all overheads (including pre-processing of the input) and shows benefits throughout all analysis configurations. This captures well the overall deployment mode of the string-merging optimization: the benefit is orthogonal to any other optimizations or analysis options, consistently applicable, and without adding potential downsides.

Running time reduction of the main analysis is only one aspect of the benefit, however. Memory is often a bottleneck for analyzing applications. Figure 5 demonstrates the reduction in memory footprint of the whole analysis database. Our approach shows a larger benefit for this metric, especially with larger analyses, such as the 2-object-sensitive+heap analysis (as there are fewer constant overheads).

Since points-to analysis is mainly used as a general substrate by higher-level analysis clients, the benefit to these is the overall size reduction of the data they import: the points-to sets for local variables (var points-to, Figure 6) and for heap object references (heap points-to, Figure 7). The sizes of these relations typically drop by factors of 1.5x or higher, across all benchmarks and different context sensitivities.

Similar results can be seen for the SOLAR reflection analysis framework, in Figure 8. Notice that, overall, the technique yields slightly less benefit for SOLAR. This is mostly attributed to the fact that SOLAR does not perform substring analysis—only strings fully matching member names are tracked by the analysis. Core reflection analysis coverage improvements such as substring analysis increase the size of the points-to set substantially since strings are allowed to flow in and out through string factory operations such as StringBuilder.append and StringBuilder.toString respectively.

6.2 RQ2: Precision and Soundness

Throughout our evaluation, the string coloring technique compromises neither precision nor soundness, since either of these conditions hold in both implementations:

- Forward and backward analyses in reflection must agree with each other—this drastically improves precision in reflection analysis, whether or not our technique is applied.
- High-confidence inferences (not requiring both forward and backward analyses) are limited by some condition that allows a preprocessing step to select which strings should be merged and which should not.

Experimentally, we have verified that both precision and soundness are preserved when our technique is enabled. One (of the many) metrics we employ to quantify precision and soundness is the number of call-graph edges (projected context-insensitively). A smaller call-graph is due to unsoundness, while a larger one is due to imprecision. Call-graph edge numbers are shown in Figure 9, and, as we can see, remain virtually identical.

6.3 RQ3: Effectiveness of Coloring Algorithm and String Merging

The graph coloring algorithm of Section 4 is very inexpensive. Figure 10 reports running times for the DaCapo Bach benchmarks, at less than a second to run on average. These
We also need to evaluate experimentally how effective the algorithm is, in terms of the number of colors it produces. Figure 10 shows this number of colors, also giving the size of the largest clique in the string-conflict graph (which is a lower bound even for optimal coloring) and the total number of string constants, i.e., nodes in the graph. The algorithm achieves significant reduction factors (mean 6.5x, i.e., 6.5 strings on average are merged into one) leaving little benefit for an algorithm that achieves tighter coloring. The largest clique for most benchmarks is the same (size 176), which is an artifact of the way the DaCapo benchmarks are packaged, with common libraries and a common harness among many benchmarks.

Contrasting Figure 10 with the earlier Figures 6-7 illustrates the numbers involved in our setting: relatively few string constants (in the thousands) result in tens of millions of extra points-to facts for the analysis, since they propagate to several points-to sets each.
### Figure 5 Memory footprint (in KB). Empty values indicate the analysis did not terminate within six hours.

<table>
<thead>
<tr>
<th>Tool</th>
<th>insens</th>
<th>1call</th>
<th>2type+H</th>
<th>2obj+H</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>2419</td>
<td>4639</td>
<td>2401</td>
<td>2923</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.17x</td>
<td>1.19x</td>
<td>1.24x</td>
</tr>
<tr>
<td>batik</td>
<td>2798</td>
<td>4851</td>
<td>6622</td>
<td>8361</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.20x</td>
<td>1.31x</td>
<td>1.27x</td>
</tr>
<tr>
<td>eclipse</td>
<td>3006</td>
<td>4930</td>
<td>3540</td>
<td>6090</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.14x</td>
<td>1.19x</td>
<td>1.13x</td>
</tr>
<tr>
<td>h2</td>
<td>3100</td>
<td>7309</td>
<td>14722</td>
<td>46377</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.37x</td>
<td>1.97x</td>
<td>1.44x</td>
</tr>
<tr>
<td>jython</td>
<td>42659</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.14x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>luindex</td>
<td>941</td>
<td>1502</td>
<td>979</td>
<td>1362</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.18x</td>
<td>1.22x</td>
<td>1.14x</td>
</tr>
<tr>
<td>lusearch</td>
<td>941</td>
<td>1505</td>
<td>974</td>
<td>1354</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.16x</td>
<td>1.20x</td>
<td>1.13x</td>
</tr>
<tr>
<td>pmd</td>
<td>1772</td>
<td>2877</td>
<td>1928</td>
<td>2903</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.18x</td>
<td>1.28x</td>
<td>1.13x</td>
</tr>
<tr>
<td>sunflow</td>
<td>1813</td>
<td>2949</td>
<td>1919</td>
<td>2145</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.16x</td>
<td>1.18x</td>
<td>1.14x</td>
</tr>
<tr>
<td>xalan</td>
<td>3751</td>
<td>7920</td>
<td>6730</td>
<td>12245</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.35x</td>
<td>1.58x</td>
<td>1.38x</td>
</tr>
<tr>
<td>average</td>
<td>1.20x</td>
<td>1.35x</td>
<td>1.22x</td>
<td>1.27x</td>
</tr>
</tbody>
</table>
As discussed in Section 4, merging string constants does not translate into proportional shrinking of string points-to sets, because many original points-to sets would not contain multiple merged strings. (The median points-to set size is 1 for many analysis settings, which allows no shrinking in most cases!) To quantify the actual reduction in string-flow inferences, we show in Figure 11 the change (for the DOOP framework) in the size of points-to sets containing strings, i.e., the number of total tuples in the VARPointsTo relation where the target object is a string. (This includes points-to inferences where the string object is not a constant but a completely unknown run-time value. However, the majority of the tuples concern variables that points to constant strings. This does not necessarily mean that the variable is inferred to have a constant string value, just that the constant string is a substring of the run-time value.)

As Figure 11 shows, string merging significantly shrinks the number of point-to inferences for strings, by roughly a factor of 2, thus capturing the majority of the potential benefit. Again, the reduction is mostly consistent throughout all the benchmarks analyzed under different context sensitivities. Interestingly, the overall reductions in points-to set sizes come

---

**Figure 6** Var points-to size (in thousands). Empty values indicate the analysis did not terminate within six hours.
Table

<table>
<thead>
<tr>
<th></th>
<th>insens</th>
<th>1call</th>
<th>2type+H</th>
<th>2obj+H</th>
</tr>
</thead>
<tbody>
<tr>
<td>avora</td>
<td>original</td>
<td>3756</td>
<td>2041</td>
<td>864</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.53x</td>
<td>1.49x</td>
<td>1.54x</td>
</tr>
<tr>
<td>batik</td>
<td>original</td>
<td>3025</td>
<td>1921</td>
<td>2956</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.84x</td>
<td>1.72x</td>
<td>1.46x</td>
</tr>
<tr>
<td>eclipse</td>
<td>original</td>
<td>2894</td>
<td>1789</td>
<td>1329</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.53x</td>
<td>1.44x</td>
<td>1.36x</td>
</tr>
<tr>
<td>r2</td>
<td>original</td>
<td>5252</td>
<td>2722</td>
<td>2965</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>2.26x</td>
<td>2.97x</td>
<td>1.58x</td>
</tr>
<tr>
<td>jython</td>
<td>original</td>
<td>45002</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.17x</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>luindex</td>
<td>original</td>
<td>924</td>
<td>569</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.75x</td>
<td>1.49x</td>
<td>1.54x</td>
</tr>
<tr>
<td>lusearch</td>
<td>original</td>
<td>952</td>
<td>569</td>
<td>309</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.73x</td>
<td>1.47x</td>
<td>1.49x</td>
</tr>
<tr>
<td>pmd</td>
<td>original</td>
<td>1562</td>
<td>1034</td>
<td>876</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.89x</td>
<td>1.74x</td>
<td>1.26x</td>
</tr>
<tr>
<td>sunflow</td>
<td>original</td>
<td>2210</td>
<td>1073</td>
<td>607</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>1.65x</td>
<td>1.48x</td>
<td>1.49x</td>
</tr>
<tr>
<td>xalan</td>
<td>original</td>
<td>6002</td>
<td>4096</td>
<td>5857</td>
</tr>
<tr>
<td></td>
<td>ratio</td>
<td>2.01x</td>
<td>2.04x</td>
<td>1.60x</td>
</tr>
<tr>
<td>average ratio</td>
<td>1.74x</td>
<td>1.76x</td>
<td>1.48x</td>
<td>1.32x</td>
</tr>
</tbody>
</table>

Figure 7 Heap points-to (in thousands). Empty values indicate the analysis did not terminate within six hours.

about due to different reasons in context-insensitive versus highly context-sensitive settings. In a context-insensitive setting, strings flow less precisely, so there are more opportunities for merged strings to appear in the same variables. In a highly context-sensitive setting, strings also make up some of the context components, and so we also see fewer references for other object types in the points-to set. We also see that strings are flowing with more precision so there is slightly less gain due to merging different strings in the same variables. Overall, these factors seem to balance themselves out and we see a consistent reduction of the points-to set for all levels of context sensitivity.

7 Related Work

There are two directions of related work: specialized graph coloring algorithms and static analyses for reflection. The former are less related to our approach, for a variety of reasons. First, our graphs have no recognized special properties. Second, most specialized graph coloring algorithms are still trying for (near-)optimal coloring, since they apply to a setting
where tight coloring yields benefits. We have the luxury of a setting where some additional number of colors makes hardly any difference in the overall benefit. Therefore, coloring simplicity and efficiency become paramount.

Some of the best-known graph coloring results with applications in programming languages can be found in the register allocation literature. Gupta et al. [12] give a fast coloring algorithm based on clique separators, i.e., cliques whose removal would disconnect the graph. In a relatively recent and prominent representative of specialized graph coloring approaches, Hack and Goos [13] give an optimal algorithm for register allocation of SSA-form programs, by showing that such programs have chordal interference graphs.

Fully analyzing reflection has been attracting attention for a long time. Multiple approaches have been proposed in the past in an effort to tackle the efficiency, soundness, and precision concerns of such an analysis.

As discussed in Section 3.1, Livshits et al. introduced the idea that reflection analysis and pointer analysis have to work together in order to be effective [21, 23]. They also identify points in a given program where user input affects the resolution of reflective targets and subsequently give the user the option to provide appropriate specifications for the aforementioned points. Additionally, they provide an automated, more conservative and sometimes less precise approach in which type casts applied to the results of reflective allocations are used in order to infer the possible values of said allocated objects.

Other work [18, 24, 30] builds on the latter concept and introduces more sophisticated backward or use-based analyses in the context of Javascript and Java respectively. When objects are retrieved from unknown code (including through reflection), the analysis tries to infer the object’s properties based on the way that it is used in the code text (generalizing to more language constructs other than type casts, e.g., string literals used in a Class.getField invocation).

Backward and forward analysis techniques are combined in great variation. For instance, Smaragdakis et al. [30] generalize the backward-information pattern by allowing for inference to arbitrarily cross method boundaries. This backward propagation technique might have adverse effects on precision under certain conditions. To that end, the authors also introduced a forward propagation approach in which type casts on unknown reflection object are used to invent a new object of the correct type at that point that will flow normally in subsequent code. This is a converse compromise since it will not affect the properties of the unknown reflection object.
Li et al. [19] developed SOLAR in which they apply three design novelties. Firstly, they use a lazy heap modeling on reflective allocation sites. Secondly, they introduce a collective inference for related reflective calls. Finally, they have in place an automatic identification of problematic reflective calls that potentially could threaten their analysis in terms of soundness, precision and scalability.

Techniques have also been proposed in order to tackle the scalability issues of a full fledged string analysis that is usually part of a reflection analysis. The most common practice is to merge string literals found in the program text into a single object (e.g., SMUSH\_STRINGS in WALA [7]). The exception to this are string constants that are a possible match with class, method or field names and so could potentially appear in a reflective call. Those literals need to be analyzed with the normal precision of the analysis at hand.

Aydin et al. [1] and Bultan [5] introduced sophisticated string analyses in the context of web applications. The former developed a constraint solver that, given a string constraint, constructs an automaton that accepts all solutions that satisfy the constraint. The latter approach extracts client- and server-side input validation and sanitization functions and models them as deterministic finite automata (DFA) using symbolic fixpoint computations.

<table>
<thead>
<tr>
<th></th>
<th>insens</th>
<th>1call</th>
<th>2type+H</th>
<th>2obj+H</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>115445</td>
<td>109226</td>
<td>97504</td>
<td>96618</td>
</tr>
<tr>
<td>coloring</td>
<td>115295</td>
<td>109076</td>
<td>97519</td>
<td>96633</td>
</tr>
<tr>
<td>difference</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>&lt;0.1%</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>batik</td>
<td>135479</td>
<td>128730</td>
<td>121031</td>
<td>120086</td>
</tr>
<tr>
<td>coloring</td>
<td>135479</td>
<td>128730</td>
<td>121031</td>
<td>120086</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>eclipse</td>
<td>94375</td>
<td>88553</td>
<td>76758</td>
<td>76212</td>
</tr>
<tr>
<td>coloring</td>
<td>94375</td>
<td>88553</td>
<td>76758</td>
<td>76212</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>h2</td>
<td>126378</td>
<td>115731</td>
<td>109814</td>
<td>108127</td>
</tr>
<tr>
<td>coloring</td>
<td>126378</td>
<td>115731</td>
<td>109814</td>
<td>108127</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>jython</td>
<td>627883</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>coloring</td>
<td>627883</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>difference</td>
<td>&lt;0.1%</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>l2index</td>
<td>64580</td>
<td>60435</td>
<td>55802</td>
<td>55808</td>
</tr>
<tr>
<td>coloring</td>
<td>64580</td>
<td>60435</td>
<td>55802</td>
<td>55808</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>eclipse</td>
<td>64748</td>
<td>60391</td>
<td>55551</td>
<td>55572</td>
</tr>
<tr>
<td>coloring</td>
<td>64748</td>
<td>60391</td>
<td>55551</td>
<td>55572</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>pmd</td>
<td>74154</td>
<td>69904</td>
<td>63704</td>
<td>63100</td>
</tr>
<tr>
<td>coloring</td>
<td>74154</td>
<td>69904</td>
<td>63704</td>
<td>63100</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
<tr>
<td>sunflow</td>
<td>100394</td>
<td>94964</td>
<td>84669</td>
<td>84062</td>
</tr>
<tr>
<td>coloring</td>
<td>100394</td>
<td>94964</td>
<td>84669</td>
<td>84062</td>
</tr>
<tr>
<td>difference</td>
<td>&lt;1%</td>
<td>&lt;1%</td>
<td>&lt;0.1%</td>
<td>&lt;0.1%</td>
</tr>
<tr>
<td>xalan</td>
<td>119042</td>
<td>109191</td>
<td>99748</td>
<td>98263</td>
</tr>
<tr>
<td>coloring</td>
<td>119042</td>
<td>109191</td>
<td>99748</td>
<td>98263</td>
</tr>
<tr>
<td>difference</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
<td>nil</td>
</tr>
</tbody>
</table>

**Figure 9** Number of call-graph edges. Empty values indicate the analysis did not terminate within six hours.
with the aim of identifying errors in input validation and sanitization code. A different advanced technique for string analysis has been presented by Christensen et al. [6]. 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. In the context of analyzing Android applications, DROIDSAFE [8] employs the JSA String Analyzer [6]. JSA is a flow-sensitive and context-insensitive static analysis that includes a model of common operations on Java’s String type. For a given string reference, the analysis computes a multi-level automaton representing all possible string values. DROIDSAFE uses JSA as a first pass (only on application code) to resolve values for string references that are arguments to the Android API. It, subsequently, converts each resolved automaton to a regular expression that represents the possible values of a string value. Generally, more precise string analyses are better suited for sensitive semantic domains and more localized application, rather than whole-program reflection analysis and arbitrary substring flow over the heap and call stack.

Traditionally, an alternative approach of handling reflection in static analysis has been the integration of user input or dynamic information along the facts inferred in a static way. A state-of-the-art example in that direction is the Tamiflex tool [3] which observes the reflective calls in an actual execution of the program and rewrites the original code to a version without reflection calls. Another hybrid dynamic-static technique is presented by Grech et al. [9] in HeapDL. The tool gathers dynamic information in the form of heap snapshots taken during program execution. Subsequently, such dynamic information is supplied to a static analysis to enhance its capabilities, substantially counteracting unsoundness with minimum intrusion to the analysis logic. Another application of this approach is to improve the scalability [10] of the analysis by replacing static reasoning with dynamic information. The use of these tools [9,10] significantly increases the number of reflective string constants in the analysis environment, which makes the techniques presented in this paper even more effective. Although, hybrid static-dynamic techniques are a practical approach, it is unrealistic to expect that reflection will yield the same results in different program executions, given that such a runtime variability is the fuel of any reflective feature.
### Table 1: String var points-to (in thousands). Empty values indicate the analysis did not terminate within six hours.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Original</th>
<th>Coloring</th>
<th>Ratio</th>
<th>Ratio</th>
<th>Ratio</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>avrora</td>
<td>10390</td>
<td>36174</td>
<td>19470</td>
<td>10567</td>
<td>17706</td>
<td>9099</td>
</tr>
<tr>
<td></td>
<td>2.20x</td>
<td>1.98x</td>
<td>1.84x</td>
<td>1.95x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>batik</td>
<td>13544</td>
<td>45280</td>
<td>55379</td>
<td>29231</td>
<td>38459</td>
<td>17332</td>
</tr>
<tr>
<td></td>
<td>2.26x</td>
<td>2.11x</td>
<td>1.89x</td>
<td>2.22x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>eclipse</td>
<td>12324</td>
<td>45265</td>
<td>26222</td>
<td>18278</td>
<td>47104</td>
<td>28001</td>
</tr>
<tr>
<td></td>
<td>1.66x</td>
<td>1.54x</td>
<td>1.43x</td>
<td>1.68x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>h2</td>
<td>19114</td>
<td>102602</td>
<td>148192</td>
<td>121101</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.23x</td>
<td>3.53x</td>
<td>2.23x</td>
<td>2.36x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jython</td>
<td>37436</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>10764</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>luindex</td>
<td>5111</td>
<td>15461</td>
<td>5869</td>
<td>6231</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3.48x</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mindex</td>
<td>5301</td>
<td>15475</td>
<td>5863</td>
<td>6080</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.97x</td>
<td>1.61x</td>
<td>1.84x</td>
<td>1.92x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pmd</td>
<td>8550</td>
<td>29199</td>
<td>12067</td>
<td>12456</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1.93x</td>
<td>1.57x</td>
<td>1.76x</td>
<td>1.75x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sunflow</td>
<td>25998</td>
<td>105773</td>
<td>84695</td>
<td>113149</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.36x</td>
<td>2.24x</td>
<td>1.69x</td>
<td>1.65x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xalan</td>
<td>11503</td>
<td>47244</td>
<td>50050</td>
<td>68774</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2.31x</td>
<td>2.02x</td>
<td>1.77x</td>
<td>1.90x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 8 Conclusion and Future Work

Reflection analysis has become a mainstream feature of modern whole-program analysis tools. Since applications and libraries in the Java ecosystem use reflection for generality and configurability, it is necessary to use reflection analysis to get a good level of program analysis coverage. On the other hand, reflection analysis is also responsible for the main performance bottleneck in whole-program analysis tools. It needs to statically track string constants that can refer to class members, which tend to dominate analyses’ points-to sets. There are not many statically-detectable hints within the semantics of the language to limit string flow.

The approach presented in this paper improves the performance of a static analysis by maximally encoding reflection string constants using graph coloring. Our technique compiles an interference graph of strings, and colors this graph using a fast, almost linear-time algorithm as a simple preprocessing step to encode string constants prior to static analysis. We find that string merging using graph coloring is an uncompromising technique...
for addressing some of the inefficiency of static analyses, for all kinds of context-sensitive and context-insensitive analyses and across multiple reflection analysis approaches.

With some adaptations, our technique can lend itself to other similar applications. For instance the technique could apply to the analysis of dynamically typed languages (where many more operations are encoded with reflection-like functionality) or to more specific domains such as the tracking of string constants matching intents in Android applications.

References


Oracle. Proxy (Java Platform SE 8), 2016. URL: http://docs.oracle.com/javase/8/docs/api/java/lang/reflect/Proxy.html.


