Combined Static and Dynamic Mutability Analysis

Shay Artzi   Adam Kieżun   David Glasser   Michael D. Ernst
MIT Computer Science and Artificial Intelligence Laboratory
{artzi,akiezun,glasser,mernst}@csail.mit.edu

Abstract

Knowing which method parameters may be mutated during a method's execution is useful for many software engineering tasks. We present an approach to discovering parameter reference immutability, in which several lightweight, scalable analyses are combined in stages, with each stage refining the overall result. The resulting analysis is scalable and combines the strengths of its component analyses. As one of the component analyses, we present a novel, dynamic mutability analysis and show how its results can be improved by random input generation. Experimental results on programs of up to 185 kLOC show that, compared to previous approaches, our approach increases both scalability and overall accuracy.

1. Introduction

Knowing which method parameters are accessed in a read-only way, and which ones may be mutated, is useful in many software engineering tasks, such as modeling [8], verification [37, 9], compiler optimizations [11, 33], program transformations such as refactoring [20], test input generation [2], regression oracle creation [23, 40], invariant detection [18], specification mining [13], program slicing [39], and program comprehension [17, 15].

Previous work has employed static analysis techniques to detect *immutable* parameters. Computing accurate static analysis approximations threatens scalability, and imprecise approximations can lead to weak results. Dynamic analyses offer an attractive complement to static approaches, both in not using approximations and in detecting *mutable* parameters.

This paper presents an approach to mutability detection that combines the strengths of static and dynamic analyses. In our approach, different analyses are combined in stages, forming a "pipeline", with each stage refining the overall result. The resulting analysis is more accurate and more scalable than previous approaches.

This paper makes the following contributions:

- The first staged analysis approach for discovering parameter mutability. Our staged approach is unusual in that it combines static and dynamic stages and it explicitly represents analysis incompleteness. The framework is sound, but an unsound analysis may be used as a component, and we examine the tradeoffs involved.
- Mutability analyses. We present a novel dynamic analysis that scales well, yields accurate results (it has a sound mode as well as optional heuristics), and complements other analyses. We extend the dynamic analysis with random input generation, which improves the analysis results by increasing code coverage. We explore a new point in the space of static techniques with a simple but effective static analysis.
- Evaluation. We have implemented our framework and analyses for Java, and we investigate the costs and benefits of various sound and unsound techniques, including both our own and that of Sălcianu and Rinard [34]. Our results show that a well-designed collection of fast, simple analyses can outperform a sophisticated analysis in both scalability and accuracy.

The remainder of this paper is organized as follows. Section 2 describes the problem of inferring parameter mutability and illustrates it on an example. Section 3 presents our staged mutability analysis. Sections 4 and 5 describe the dynamic and static mutability analyses that we developed as components in the staged analysis. Section 6 describes the experimental evaluation. Section 7 surveys related work, and Section 8 concludes.

2. Parameter Reference Immutability

The goal of parameter mutability analysis is the classification of each method parameter (including the receiver) as either reference-mutable or reference-immutable.

Informally, reference immutability guarantees that a given reference is not used to modify its referent. Parameter p of method m is reference-*mutable* if there exists an execution of m in which p is *used* to mutate the state of the object pointed to by p. Parameter p is said to be *used* in a mutation, if the left hand side of the mutating assignment was obtained during the given execution via a series of field accesses and copy operations from p. If no such execution exists, the parameter p is reference-*immutable*. For example, in the following method:

```
void f(C c) {
  D d = c.d;
  E e = d.e;
  e.f = null;
}
```

parameter `c` is used in the mutation in the last statement since the statement is equivalent to `c.d.e.f = null`.

The state of an object o consists of the values of o's primitive fields (e.g., `int`, `float`) and the states of all objects pointed to by o's non-primitive fields. The mutation may occur in m itself or in any method that m (transitively) calls. Array accesses are treated analogously to fields. Throughout this paper two objects are *aliased* if the intersection of their states contains at least one non-primitive object (the same object is reachable from both of them).

Previous work has mostly given informal definitions of immutability, like the above. A companion technical report [3] gives the first formal definition of reference immutability that accounts for parameter aliasing.

By contrast to reference immutability, *object immutability* indicates whether a specific object may be changed, via any reference. Among other uses, reference immutability is more useful for reasoning about method calls and for distinguishing between acceptable and unacceptable modifications. Reference immutability can be combined with aliasing information to compute object

```
1  class C {
2     public C next;
3  }
4
5  class Main {
6     void modifyParam1(C p1, boolean doIt) {
7        if (doIt) {
8           p1.next = null;
9        }
10    }
11
12    void modifyParam1Indirectly(C p2, boolean doIt) {
13       modifyParam1(p2, doIt);
14    }
15
16    void modifyAll(C p3, C p4, C p5, boolean doIt) {
17       p4.next = p3;
18       C c = p5.next;
19       c.next = null;
20       modifyParam1Indirectly(p5, doIt);
21    }
22
23    void doNotModifyAnyParam(C p6) {
24       if (p6.next == null)
25          System.out.println("p6.next is null");
26    }
27    void doNotModifyAnyParam2(C p7) {
28       doNotModifyAnyParam(p7);
29    }
30 }
```

Figure 1: Example code that illustrates our staged approach to parameter immutability. All non-primitive parameters other than `p6` and `p7` are *mutable*.

immutability [7, 34]. The rest of this paper uses *mutable* and *immutable* to refer to reference mutable and reference immutable, respectively.

2.1 Example

In the code in Figure 1, parameters `p6` and `p7` are reference-*immutable*, and parameters `p1` – `p5` are reference-*mutable*, since there exists an execution of their declaring method such that the object pointed to by the parameter reference is modified *via the reference*.

***immutable* parameters**:

- `p6` and `p7` are reference-*immutable*. No execution of either method `doNotModifyAnyParam` or `doNotModifyAnyParam2` can modify an object passed to `p6` or `p7`.

***mutable* parameters**:

- `p1` may be directly modified in `modifyParam1` (line 8).
- `p2` is passed to `modifyParam1`, in which it may be mutated.
- `p3` is *mutable* because the state of the object passed to `p3` can get modified on line 19 via `p3`. This can happen because `p4` and `p5` might be aliased; for example, in the call `modifyAll(x1, x2, x2, false)`. In this case, the reference to `p3` is copied into `c` and then used to perform a modification on line 19.
- `p4` is directly modified in `modifyAll` (line 17). Note that line 17 does *not* modify `p3` or `p5`, because the mutation occurs via reference `p4`. This paper is concerned with *reference*-(im)mutability rather than *object-(im)mutability* and thus the reference via which the modification happens is significant.
- `p5` is *mutable* because line 19 modifies `p5.next.next`.

Our dynamic and static analyses complement each other to classify parameters into *mutable* and *immutable*, in the following steps:

1. Initially, all parameters are *unknown*.
2. A flow-insensitive, intra-procedural static analysis classifies `p1`, `p4`, and `p5` as *mutable*. The analysis classifies `p6` as *immutable*—there is no direct mutation in the method and the parameter does not escape.
3. An inter-procedural static analysis propagates the current classification along the call-graph and classifies `p2` as *mutable* since it is passed to an already known mutable parameter, `p1`. It also classifies parameter `p7` as *immutable* since it can only be passed to *immutable* parameters.
4. A dynamic analysis classification of `p3` depends on the given example execution. The dynamic analysis classifies `p3` as *mutable* if a method (similar to the `main` method below)

   ```
   void main() {
     modifyAll(x1, x2, x2, false);
   }
   ```

   is supplied or generated (see Section 4.4). Otherwise the dynamic analysis classifies `p3` as *unknown*.

Our staged analysis correctly classifies all parameters in Figure 1. However, this example poses difficulties for purely static or purely dynamic techniques. On the one hand, static techniques have difficulties correctly classifying `p3`. This is because, to avoid over-conservatism, static analyses often assume that on entry to a method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. In our example, this assumption may lead such analyses to incorrectly classify `p3` as *immutable* (in fact, Sălcianu uses a similar example to illustrate the unsoundness of his analysis [33, p.78]). On the other hand, dynamic analyses are limited to a specific execution and only consider modifications that happen during that execution. In our example, a purely dynamic technique may incorrectly classify `p2` as *immutable* if during the execution, `p2` is not modified.

3. Staged Mutability Analysis

In our approach, mutability analyses are combined in stages, forming a "pipeline". The input to the first stage is the initial classification of all parameters (typically, all *unknown*, though parameters declared in the standard libraries may be pre-classified). Each stage of the pipeline refines the results computed by the previous stage by classifying some *unknown* parameters. Once a parameter is classified as *mutable* or *immutable*, further stages do not change the classification. The output of the last stage is the final classification, in which some parameters may remain *unknown*.

Combining mutability analyses can yield an analysis that has better accuracy than any of the components. For example, a static analysis can analyze an entire program and can prove the absence of a mutation, while a dynamic analysis can avoid analysis approximations and can prove the presence of a mutation.

Combining analyses in a pipeline also has performance benefits—a component analysis in a pipeline may ignore previously classified parameters. This can permit the use of techniques that would be too computationally expensive if applied to an entire program.

The problem of mutability inference is undecidable, so no analysis can be both sound and complete. An analysis is *i-sound* if it never classifies a *mutable* parameter as *immutable*. An analysis is *m-sound* if it never classifies an *immutable* parameter as *mutable*. An analysis is *complete* if it classifies every parameter as either *mutable* or *immutable*.

In our staged approach, analyses may explicitly represent their incompleteness using the *unknown* classification. Thus, an analysis result classifies parameters into three groups: *mutable*, *immutable*, and *unknown*. Previous work that used only two output classifications [31, 29] loses information by conflating parameters/methods

Analysis	Name	Section	i-sound	m-sound
dynamic	D	4.2	✓*	✓
dynamic heuristic **A**	DA	4.3	-	✓
dynamic heuristic **B**	DB	4.3	✓*	-
dynamic heuristic **C**	DC	4.3	✓*	-
dynamic heuristics **A**,**B**,**C**	DH	4.3	-	-
static intraprocedural	S	5.2	✓	-
static intraprocedural heuristic	SH	5.2.1	-	-
static interproc. propagation	P	5.3	✓	✓†
JPPA [34]	J	6.2	-	✓*
JPPA + `main`	JM	6.2	-	✓*
JPPA + `main` + heuristic	JMH	6.2	-	-

Figure 2: The static and dynamic component analyses used in our experiments. "✓*" means the algorithm is trivially sound, by never outputting the given classification. "✓†" means the algorithm is sound but our implementation is not.

that are known to be mutable with those where analysis approximations prevent definitive classification.

Some tasks, such as many compiler optimizations [11, 33] require i-sound results (unless the results are treated as hints or are used online for only the current execution [41]). Therefore, we have i-sound versions of our static and our dynamic analyses. However, other tasks, such as test input generation [2], can benefit from more complete immutability classification while tolerating i-unsoundness. For this reason, we have devised several unsound approximations to increase the completeness (recall) of the analyses. Clients of the analysis can create an i-sound analysis by combining only i-sound components. Other clients, desiring more complete information, can use i-unsound components as well. Figure 2 summarizes the soundness characteristics of the analyses presented in this paper.

4. Dynamic Mutability Analysis

Our dynamic mutability analysis observes the program's execution and classifies as *mutable* those method parameters that are used to mutate objects. The algorithm is m-sound: it classifies a parameter as *mutable* only when the parameter is mutated. The algorithm is also i-sound: it classifies all remaining parameters as *unknown*. Section 4.1 gives the idea behind the algorithm, and Section 4.2 describes an optimized implementation.

To improve the analysis results, we developed several heuristics (Section 4.3). Each heuristic carries a different risk of unsoundness. However, most are shown to be accurate in our experiments. The analysis has an iterative variation with random input generation (Section 4.4) that improves analysis precision and run-time.

4.1 Conceptual Algorithm

During program execution, the dynamic analysis tags each reference in the running program with the set of all formal parameters (from any method invocation on the call stack) whose fields were directly or indirectly accessed to obtain the reference. When a reference `x` is side-effected (i.e., used in `x.f = y`), all formal parameters in the set of `x` are classified as mutable. The analysis tags references, not objects, because more than one reference can point to the same object. Primitives need not be tagged, as they are immutable.

The algorithm for detecting mutable parameters is given by a set of data-flow rules. The rules track mutations to each parameter. Next, we present those rules informally. The rules are formalized in the full version of this paper [3].

1. On method entry, the algorithm adds each formal parameter (that is classified as *unknown*) to the parameter set of the corresponding actual parameter reference.
2. On method exit, the algorithm removes all parameters for the current invocation from the parameter sets of all references in the program.
3. Assignments, including pseudo-assignments for parameter passing and return values, propagate the parameter sets unchanged.
4. Field accesses also propagate the sets unchanged: the set of parameters for `x.f` is the same as that of `x`.
5. For a field write `x.f = v`, the algorithm classifies as *mutable* all parameters in the parameter set of `x`.

The next section presents an alternative algorithm that we implemented.

4.2 Dynamic Analysis Algorithm

Maintaining reference tag sets for all references, a required by the algorithm of Section 4.1, is computationally expensive. To improve performance, we developed an alternative algorithm that does not maintain parameter reference tags. The alternative algorithm is i-sound and m-sound, but is less complete—it classifies fewer parameters. In the alternative algorithm, parameter p of method m is classified as *mutable* if: (i) the transitive state of the object that p points to changes during the execution of m, and (ii) p is not aliased to any other parameter of m. Without part (ii), the algorithm would not be m-sound—*immutable* parameters that are aliased to a *mutable* parameter during the execution may be wrongly classified as *mutable*.

The example code in Figure 1 illustrates the difference between the conceptual algorithm presented in Section 4.1 and the algorithm presented in this section. When method `main` executes, it calls `modifyAll`. The conceptual algorithm based on the definition (correctly) classifies parameters `p1`-`p5` as *mutable*. The alternative algorithm leaves `p4` and `p5` as *unknown* since those parameters are aliased (in fact in this example they are the same object)—when the modification to the referent object happens (line 17), the mutation is tracked to both parameters `p4` and `p5`. Note that the intra-procedural static analysis (Section 5.1) compensates for the incompleteness of the dynamic analysis in this case and correctly classifies `p4` and `p5` as *mutable*.

The algorithm permits an efficient implementation: when method m is called during the program's execution, the analysis computes the set $reach(m, p)$ of objects that are transitively reachable from each parameter p via field references. When the program writes to a field in object o, the analysis finds all parameters p of methods that are currently on the call stack. For each such parameter p, if $o \in reach(m, p)$ and p is not aliased to other parameters of m, then the analysis classifies p as *mutable*. The algorithm checks aliasing by verifying emptiness of intersection of reachable sub-heaps (ignoring immutable objects, such as boxed primitives, which may be shared).

The implementation instruments the analyzed code at load time. The analysis works online, i.e., in tandem with the target program, without creating a trace file. Our implementation includes the following three optimizations, which together improve the run time by over 30×: (a) the analysis determines object reachability by maintaining and traversing its own data structure that mirrors the heap, which is faster than using reflection; (b) the analysis computes the set of reachable objects lazily, when a modification occurs; and (c) the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set is modified.

4.3 Dynamic Analysis Heuristics

The dynamic analysis algorithm described in Sections 4.1 and 4.2 is m-sound—a parameter is classified as *mutable* only if it is modified during execution. Heuristics can improve the completeness, or recall (see Section 6), of the algorithm. The heuristics take advantage of the *absence* of parameter modifications and of the classification results computed by previous stages in the analysis pipeline. Using the heuristics may potentially introduce i-unsoundness or m-unsoundness to the analysis results, but in practice, they cause few misclassifications (see Section 6.3.5).

(**A**) **Classifying parameters as *immutable* at the end of the analysis.** This heuristic classifies as *immutable* all (*unknown*) parameters that satisfy conditions that are set by the client of the analysis. In our framework, the heuristic classifies as *immutable* a parameter p declared in method m if p was not modified, m was executed at least N times, and the executions achieved block coverage of at least t%. Higher values of the threshold N or t increase i-soundness but decrease completeness.

The intuition behind this heuristic is that, if a method executed multiple times, and the executions covered most of the method, and the parameter was not modified during any of those executions, then the parameter may be *immutable*. This heuristic is m-sound but i-unsound. In our experiments, this heuristic greatly improved recall and was not a significant source of mistakes (Section 6.3.5).

(**B**) **Using current mutability classification.** This heuristic classifies a parameter as *mutable* if the object to which the parameter points is passed in a method invocation to a formal parameter that is already classified as *mutable* (by a previous or the current analysis). That is, the heuristic does not wait for the actual modification of the object but assumes that the object will be modified if it is passed to a *mutable* position. The heuristic improves analysis performance by not tracking the object in the new method invocation.

The intuition behind this heuristic is that if an object is passed as an argument to a parameter that is known to be *mutable*, then it is likely that the object will be modified during the call. The heuristic is i-sound but m-unsound. In our experiments, this heuristic improved recall and run time of the analysis and caused few misclassifications (see Section 6.3.5).

(**C**) **Classifying aliased mutated parameters.** This heuristic classifies a parameter p as *mutable* if the object that p points to is modified, regardless of whether the modification happened through an alias to p or through the reference p itself. For example, if parameters a and b happen to point to the same object o, and o is modified, then this heuristic will classify both a and b as *mutable*, even if it the modification is only done using the formal parameter's reference to a.

The heuristic is i-sound but m-unsound. In our experiments, using this heuristic improved the results in terms of recall, without causing any misclassifications.

4.4 Using Randomly Generated Inputs

In this section we consider the use of randomly generated sequences of method calls as the required input for the dynamic analysis. Random generation can complement (or even replace) executions provided by a user. For instance, Pacheco et al [26], uses feedback directed random generation to detect previously-unknown errors in widely used (and tested) libraries.

Using randomly generated execution has benefits for a dynamic analysis. First, the user need not provide a sample execution. Second, random executions may explore parts of the program that the user-supplied executions do not reach. Third, each of the generated random inputs may be executed immediately—this allows the client of the analysis to stop generating inputs when the client is satisfied with the results of the analysis computed so far. Fourth, the client of the analysis may focus the input generator on methods with unclassified parameters.

Our generator gives a higher selection probability to methods with *unknown* parameters and methods that have not yet been executed by other dynamic analyses in the pipeline. Generation of random inputs is iterative. After the dynamic analysis has classified some parameters, it makes sense to propagate that information (see Section 5.3) and to re-focus random input generation on the remaining *unknown* parameters. Such re-focusing iterations continue as long as each iteration classifies at least 1% of the remaining *unknown* parameters (the threshold is user-settable).

By default, the number of generated method calls per iteration is *max*(*5000*, *#methodsInProgram*). The randomly generated inputs are executed in safe way [26], using a Java security manager.

5. Static Mutability Analysis

This section describes a simple, scalable static mutability analysis. It consists of two phases: S, an intraprocedural analysis that classifies as (*im*)*mutable* parameters (never) affected by field writes within the procedure itself (Section 5.2), and P, an interprocedural analysis that propagates mutability information between method parameters (Section 5.3). P may be executed at any point in an analysis pipeline after S has been run, and may be run multiple times (interleaving with other analyses). S and P both rely on an intraprocedural pointer analysis that calculates the parameters pointed to by each local variable (Section 5.1).

5.1 Intraprocedural Points-To Analysis

To determine which parameters can be pointed to by each expression, we use an intraprocedural, context-insensitive, flow-insensitive, 1-level field-sensitive, points-to analysis. As a special case, the analysis is flow-sensitive on the code from the beginning of a method through the first backwards jump target, which includes the entire body of methods without loops. We are not aware of previous work that has explored this point in the design space, which we found to be both scalable and sufficiently precise.

The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters whose state l can point to directly and a set $P(l)$ of parameters whose state l can point to directly or transitively. (Without loss of generality, we assume three-address SSA form and consider only local variables.) The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated (see below).

For each local variable l and parameter p, the analysis calculates a distance map $D(l, p)$ from the fields of object l to a non-negative integer or ∞. $D(l, p)(f)$ represents the number of dereferences that can be applied to l starting with a dereference of the field f to find an object pointed to (possibly indirectly) by p. Each map $D(l, p)$ is either strictly positive everywhere or is zero everywhere. As an example, suppose l directly references p or some object transitively pointed to by p; then $D(l, p)(f) = 0$ for all f. As another example, suppose $l.f.g.h = p.x$; then $D(l, p)(f) = 3$. The distance map D makes the analysis field-sensitive, but only at the first layer of dereferencing; we found this to be important in practice to provide satisfactory results.

The points-to analysis computes $D(l, p)$ via a fixpoint computation on each method. At the beginning of the computation, $D(p, p)(f) = 0$, and $D(l, p)(f) = \infty$ for all $l \neq p$. Due to space constraints, we give the flavor of the dataflow rules with a few examples:

- A field dereference $l_1 = l_2.f$ updates

$$\begin{aligned} \forall g : D(l_1, p)(g) &\leftarrow \min(D(l_1, p)(g), D(l_2, p)(f) - 1) \\ D(l_2, p)(f) &\leftarrow \min(D(l_2, p)(f), \min_g D(l_1, p)(g) + 1) \end{aligned}$$

- A field assignment $l_1.f = l_2$ updates

$$
\begin{aligned}
D(l_1, p)(f) &\leftarrow \min(D(l_1, p)(f), \min_g D(l_2, p)(g) + 1) \\
\forall g : D(l_2, p)(g) &\leftarrow \min(D(l_2, p)(g), D(l_1, p)(f) - 1)
\end{aligned}
$$

- Method calls are handled either by assuming they create no aliasing (creating an underestimate of the true points-to sets) or by assuming they might alias all of their parameters together (for an overestimate). If an underestimate is desired, no values of $D(l, p)(f)$ are updated. For an overestimate, let S be the set of all locals used in the statement (including receiver and return value); for each $l \in S$ and each parameter p, set $D(l, p)(f) \leftarrow \min_{l' \in S, g} D(l', p)(g)$.

After the computation reaches a fixpoint, it sets

$$
\begin{aligned}
P(l) &= \{p \mid \exists f : D(l, p)(f) \neq \infty\} \\
P_0(l) &= \{p \mid \forall f : D(l, p)(f) = 0\}
\end{aligned}
$$

5.2 Intraprocedural Phase: S

The static analysis S works in four steps. First, S performs the "overestimate" points-to analysis (Section 5.1). Second, the analysis marks as *mutable* some parameters that are currently marked as *unknown*: For each mutation $l_1.f = l_2$, the analysis marks all elements of $P_0(l_1)$ as *mutable*. Third, the analysis computes a "leaked set" L of locals, consisting of all arguments (including receivers) in all method invocations and any local assigned to a static field (in a statement of the form `Global.field = local`). Fourth, the analysis marks as *immutable* all *unknown* parameters that are not in the set $\cup_{l \in L} P(l)$ only if all method's parameters can be marked *immutable*.

S is i-sound and m-unsound. To avoid over-conservatism, S assumes that on the entry to the analyzed method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. This assumption may cause S to miss possible mutations due to aliased parameters; to maintain i-soundness, S never classifies a parameter as *immutable* unless all other parameters to the method can be classified as *immutable*. The m-unsoundness of S is due to infeasible paths (e.g., unreachable code), flow-insensitivity, and the overestimation of the points-to analysis.

For example, S does not detect any mutation to parameter `p3` of the method `modifyAll` in Figure 1. Since other parameters of `modifyAll` (i.e., `p4` and `p5`) are classified as *mutable*, S conservatively leaves `p3` as *unknown*. In contrast, Sălcianu's static analysis JPPA [34] incorrectly classifies `p3` as *immutable*. JPPA's result is incorrect because there may exist an execution in which `p4` and `p5` are aliased, in which case the object passed to `p3` is mutated in line 19 in Figure 1.

5.2.1 Intraprocedural Analysis Heuristic

We have also implemented a i-unsound heuristic SH that is like S, but it can classify parameters as *immutable* even when other parameters of the same method are not classified as *immutable*. In our experiments, this never caused a misclassification.

5.3 Interprocedural Propagation Phase: P

The interprocedural propagation phase P refines the current parameter classification by propagating both mutability and immutability information through the call graph. Given an i-sound input classification, propagation is i-sound and m-unsound.

Because propagation ignores the bodies of methods, the P phase is i-sound only if the method bodies have already been analyzed. It is intended to be run only after the S phase of Section 5.1 has already been run. However, it can be run multiple times (with other analyses in between).

Section 5.3.1 describes the binding multi-graph (BMG), and then Section 5.3.2 gives the propagation algorithm itself.

5.3.1 Binding Multi-Graph

The propagation uses a variant of the *binding multi-graph* (BMG) [12]; our extension accounts for pointer data structures. Each node is a method parameter `m.p`. An edge from `m1.p1` to `m2.p2` exists iff `m1` calls `m2`, passing as parameter `p2` part of `p1`'s state (either `p1` or an object that may be transitively pointed-to by `p1`).

A BMG is created by generating a call-graph and translating each method call edge into a set of parameter dependency edges, using the sets $P(l)$ described in Section 5.1 to tell which parameters correspond to which locals.

The BMG creation algorithm is parameterized by a call-graph construction algorithm. Our experiments used CHA [14]—the simplest and least precise call-graph construction algorithm offered by Soot. In the future, we want to investigate using more precise but still scalable algorithms, such as RTA [4] (available in Soot, but containing bugs that prevented us from using it), or those proposed by Tip and Palsberg [36] (not implemented in Soot).

The true BMG is not computable, because determining perfect aliasing and call information is undecidable. Our analysis uses an under-approximation (i.e., it contains a subset of edges of the ideal graph) and an over-approximation (i.e., it contains a superset of edges of the ideal graph) to the BMG as safe approximations for determining mutable and immutable parameters, respectively. One choice for the over-approximated BMG is the *fully-aliased* BMG, which is created with an overestimating points-to analysis which assumes that method calls introduce aliasings between *all* parameters. One choice for the under-approximated BMG is the *un-aliased* BMG, which is created with an underestimating points-to analysis which assumes that method calls introduce *no* aliasings between parameters. More precise approximations could be computed by a more complex points-to analysis.

To construct the under-approximation of the true BMG, propagation needs a call-graph that is an under-approximation of the real call-graph. However, most existing call-graph construction algorithms [14, 16, 4, 36] create an over-approximation. Therefore, our implementation uses the same call-graph for building the un- and fully-aliased BMGs. Due to this approximation, our implementation of P is m-unsound. Actually, P is m-unsound even on the under-approximation of the BMG. For example, assume that m1.p1 is unknown, m2.p2 is mutable, and there is an edge between m1.p1 and m2.p2. It is possible that there is an execution of m2.p2 in which p2 is mutated, but for every execution that goes through m1, m2.p2 is immutable. In this case, the algorithm would incorrectly classify m1.p1 as mutable. In our experiments, this approximation caused several misclassifications of *immutable* parameters as *mutable* (see Section 6.3.1).

5.3.2 Propagation Algorithm

Propagation refines the parameter classification in 2 phases.

The **mutability propagation** classifies as *mutable* all the *unknown* parameters that can reach in the under-approximated BMG (can flow to in the program) a parameter that is classified as *mutable*. Using an over-approximation to the BMG would be unsound because spurious edges may lead propagation to incorrectly classify parameters as mutable.

The **immutability propagation** phase classifies additional parameters as *immutable*. This phase uses a fix-point computation: in each step, the analysis classifies as *immutable* all *unknown* pa-

program	size (LOC)	classes	parameters all	non-trivial	inspected
jolden	6,215	56	705	470	470
sat4j	15,081	122	1,499	1,136	118
tinysql	32,149	119	2,408	1,708	206
htmlparser	64,019	158	2,270	1,738	82
ejc	107,371	320	9,641	7,936	7,936
daikon	185,267	842	16,781	13,319	73
Total	410,102	1,617	33,304	26,307	8,885

Figure 3: Subject programs.

rameters that have no *mutable* or *unknown* successors (callees) in the over-approximated BMG. Using an under-approximation to the BMG would be unsound because if an edge is missing in the BMG, the analysis may classify a parameter as *immutable* even though the parameter is really mutable. This is because the parameter may be missing, in the BMG, a *mutable* successor.

6. Evaluation

We experimentally evaluated all sensible combinations (192 in all) of the mutability analyses described above, comparing the results with each other and with the correct classification of parameters. Our results indicate that staged mutability analysis can be accurate, scalable, and useful.

6.1 Methodology and Measurements

We computed mutability for 6 open-source subject programs (see Figure 3). When an example input was needed (e.g., for a dynamic analysis), we ran each subject program on a single input.

- **jolden** (`http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`) is a benchmark suite of 10 small programs. As the example input, we used the `main` method and arguments that were included with the benchmarks. We included these programs primarily to permit comparison with Sălcianu's evaluation [34].
- **sat4j** (`http://www.sat4j.org/`) is a SAT solver. We used a file with an unsatisfiable formula as the example input.
- **tinysql** (`http://sourceforge.net/projects/tinysql`) is a minimal SQL engine. We used the program's test suite as the example input.
- **htmlparser** (`http://htmlparser.sourceforge.net/`) is a real-time parser for HTML. We used our research group's webpage as the example input.
- **ejc** (`http://www.eclipse.org/`) is the Eclipse Java compiler. We used one Java file as the example input.
- **daikon** (`http://pag.csail.mit.edu/daikon/`) is an invariant detector. We used the StackAr test case from its distribution as the example input.

As the input to the first analysis in the pipeline, we used a pre-computed classification for all parameters in the Java standard libraries. Callbacks from the library code to the client code (e.g., `toString()`, `hashCode()`) were analyzed under the closed world assumption in which all of the subject programs were included. The pre-computed classification was created once, and reused many times in all the experiments. A benefit of using this classification is that it covers otherwise un-analyzable code, such as native calls.

We measured the results only for non-trivial parameters declared in the application. That is, we did not count parameters with a primitive, boxed primitive, or `String` type, nor parameters declared in external or JDK libraries.

To measure the accuracy of each mutability analysis, we determined the correct classification (*mutable* or *immutable*) for 8,885 parameters: all of jolden and ejc, and 5 randomly-selected classes from each of the other programs. To find the correct classification, we first ran every tool available to us (including our analysis pipelines, Sălcianu's tool, and the Javarifier [38] type inference tool for Javari). Then, we manually verified the correct classification for every parameter where any two tool results differed, or where only one tool completed successfully. In addition, we verified an additional 200 parameters, chosen at random, where all tools agreed. We found no instances where the tools agreed on the mutability result, but the result was incorrect.

Figure 4 and the tables in Section 6.3 present precision and recall results, computed as follows:

$$\text{i-precision} = \frac{ii}{ii+im} \qquad \text{m-precision} = \frac{mm}{mm+mi}$$

$$\text{i-recall} = \frac{ii}{ii+ui+mi} \qquad \text{m-recall} = \frac{mm}{mm+um+im}$$

where *ii* is the number of immutable parameters that are correctly classified, and *mi* is the number of immutable parameters incorrectly classified as *mutable* (similarly, *ui*). Similarly, for mutable parameters, we have *mm*, *im* and *um*. i-precision is measure of soundness: it counts how often the analysis is correct when it classifies a parameter as *immutable*. i-recall is measure of completeness: it counts how many immutable parameters are marked as such by the analysis. m-precision and m-recall are similarly defined. An i-sound analysis has i-precision of 1.0, and an m-sound analysis has m-precision of 1.0. Ideally, both precision and recall should be 1.0, but this is not feasible: there is always a trade-off between analysis precision and recall.

6.2 Evaluated Analyses

Our experiments evaluate pipeline analyses composed of analyses described in Section 3. X-Y-Z denotes a staged analysis in which component analysis X is followed by component analysis Y and then by component analysis Z.

Our experiments use the following component analyses:

- S is the sound intraprocedural static analysis (Section 5.2).
- SH is the intraprocedural static analysis heuristic (Section 5.2.1).
- P is the interprocedural static propagation (Section 5.3).
- D is the dynamic analysis (Section 4), using the inputs of Section 6.1.
- DH is D, augmented with all the heuristics described in Section 4.3. DA, DB, and DC are D, augmented with just one of the heuristics.
- DRH is DH enhanced with random input generation (Section 4.4); likewise for DRA, etc.
- J is Sălcianu and Rinard's state-of-the-art static analysis JPPA [34]. It never classifies parameters as *mutable*—only *immutable* and *unknown*.
- JM is J, augmented to use a `main` method that contains calls to all the public methods in the subject program [29]; J only analyzes methods that are reachable from `main`.
- JMH is JM plus an m-unsound heuristic to classify as *mutable* any parameter for which J provides an explanation of a potential modification.

6.3 Results

We experimented with six programs and 192 different analysis pipelines. Figure 4 compares the accuracy of a selected set of mutability analyses among those with which we experimented. S-P-DRBC-P is the best-performing i-sound staged analysis. For uses that do not require i-soundness, the pipeline with the highest overall precision and recall was SH-P-DRH-P. Compared to Sălcianu's [34] state-of-the-art analysis J, the staged mutability analysis achieves equal or slightly worse i-precision, better i-recall, and

Prog.	Analysis	i-recall	i-precision	m-recall	m-precision
ejc	S-P-DRBC-P	0.781	1.000	0.915	0.956
	SH-P	0.777	1.000	0.904	0.971
	SH-P-DRH-P	0.928	0.996	0.907	0.971
	J	0.593	0.999	0.000	0.000
	JMH	0.734	0.998	0.691	0.941
	JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951
jolden	S-P-DRBC-P	0.829	1.000	1.000	0.924
	SH-P	0.829	1.000	0.907	1.000
	SH-P-DRH-P	0.973	1.000	1.000	0.970
	J	0.894	1.000	0.000	0.000
	JMH	0.985	1.000	0.660	0.955
	JMH-SH-P-DRH-P	0.989	0.996	0.990	0.970
daikon	S-P-DRBC-P	0.705	1.000	0.931	0.844
	SH-P	0.636	1.000	0.931	0.844
	SH-P-DRH-P	0.750	1.000	0.931	0.844
	J	0.750	1.000	0.000	0.000
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
tiny+sat+html	S-P-DRBC-P	0.836	1.000	0.863	0.953
	SH-P	0.836	1.000	0.863	0.965
	SH-P-DRH-P	0.968	0.984	0.947	0.957
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-

Figure 4: Mutability analyses on subject programs. Subjects tinysql, sat4j and htmlparser are presented jointly as the last group, marked as tiny+sat+html. Empty cells mean that the analysis aborted with an error.

much better m-recall and m-precision. The staged analysis is also considerably more scalable.

This section discusses the important observations that stem from the results of our experiments. Each sub-section discusses one observation that is supported by a table listing representative pipelines illustrating the observation. The tables in this section present results for ejc. Results for other programs were similar. However, for smaller programs all analyses did better and the differences in results were not as pronounced.

6.3.1 Interprocedural Propagation

Running interprocedural propagation (P in the tables) is always beneficial, as the following table shows on representative pipelines.

Analysis	i-recall	i-precision	m-recall	m-precision
SH	0.563	1.000	0.299	0.998
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-P	0.940	0.776	0.663	0.988

Propagation may decrease m-precision but, in our experiments, the decrease was never larger than 0.03 (not shown in the above table). In the experiments, propagation always increased all other statistics (sometimes significantly). For example, the table shows that propagation increased i-recall from 0.563 in SH to 0.777 in SH-P and it increased m-recall from 0.299 in SH to 0.904 in SH-P. Moreover, since almost all of the run-time cost of propagation lies in the call-graph construction, only the first execution incurs notable run-time cost on the analysis pipeline; subsequent executions of propagation are fast. Therefore, most pipelines presented in the sequel have P stages executed after each other analysis stage.

6.3.2 Combining Static and Dynamic Analysis

Combining static and dynamic analysis in either order is helpful—the two types of analysis are complementary.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-SH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-SH-P	0.939	0.812	0.722	0.981
DRH-SH-P-DRH	0.943	0.813	0.722	0.981

For best results, the static stage should precede the dynamic stage. Pipeline SH-P-DRH, in which the static stage precedes the dynamic stage, achieved significantly better i-precision and m-recall than DRH-SH-P, with marginally lower (by .01-.02) i-recall and m-precision.

Repeating executions of static or dynamic analyses bring no substantial further improvement. For example, SH-P-DRH-SH-P (i.e., static-dynamic-static) achieves essentially the same results as SH-P-DRH (i.e., static-dynamic). Similarly, DRH-SH-P-DRH (i.e., dynamic-static-dynamic) only marginally improves i-recall over DRH-SH-P (i.e., dynamic-static).

6.3.3 Comparing Static Stages

In a staged mutability analysis, using a more complex static analysis brings little benefit. We experimented with replacing our lightweight interprocedural static analysis with J, Sălcianu's heavyweight static analysis.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DRH-P	0.928	0.996	0.907	0.971
J-DRH-P	0.973	0.787	0.664	0.998
JMH-DRH-P	0.939	0.922	0.878	0.949
JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951

SH-P-DRH-P outperforms JMH-DRH-P with respect to 3 of 4 statistics, including i-precision (see Section 6.3.6). Combining the two static analyses improves recall—JMH-SH-P-DRH-P has better i-recall than SH-P-DRH-P and better m-recall than JMH-DRH-P. This shows that the two kinds of static analysis are complementary.

6.3.4 Randomly Generated Inputs in Dynamic Analysis

Using randomly generated inputs to the dynamic analysis (DRH) achieves better results than using a user-supplied execution (DH). We also considered pipelines that use both types of executions.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DH	0.827	0.984	0.911	0.961
SH-P-DH-P-DRH	0.917	0.984	0.915	0.958
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P-DH	0.932	0.983	0.912	0.970

Pipeline SH-P-DRH achieves better results than SH-P-DH with respect to i-precision, i-recall and m-precision (with lower m-recall). Using both kinds of executions can have different effects. For instance, SH-P-DH-P-DRH has better results than SH-P-DH, but SH-P-DRH-P-DH has a lower i-precision (due to i-unsound-ness of heuristic **A**) with a small gain in i-recall and m-recall over SH-P-DRH.

The surprising finding that randomly generated code is as effective as using an example execution suggests that other dynamic analyses (e.g., race detection [35, 25], invariant detection [18], inference of abstract types [21], and heap type inference [27]) might also benefit from replacing example executions with random executions.

6.3.5 Dynamic Analysis Heuristics

By exhaustive evaluation, we determined that each of the heuristics is beneficial. A pipeline with DRH achieves notably higher i-recall and only slightly lower i-precision than a pipeline with DR (which uses no heuristics). This section indicates the unique contribution of each heuristic, by removing it from the full set (because some heuristics may have overlapping benefits).

Analysis	total time (s)	last component (s)
S	167	167
S-P	564	397
SH	167	167
SH-P	564	397
SH-P-DH	859	295
SH-P-DH-P	869	10
SH-P-DRH	1484	920
SH-P-DRH-P	1493	9
J	5586	5586
JM	-	-
JMH	-	-

Figure 5: The cumulative run time, and the time for the last component analysis in the pipeline, for the daikon subject program. Empty cells indicate that the analysis aborted with an error.

analysis	nodes	ratio	edges	ratio	time (s)	ratio
jolden + ejc + daikon						
no immutability	444,729	1.00	624,767	1.00	6,703	1.00
SH-P-DRH-P	124,601	3.57	201,327	3.10	4,271	1.56
J	131,425	3.83	210,354	2.97	4,626	1.44
htmlparser + tinysql + sat4j						
no immutability	48,529	1.00	68,402	1.00	215	1.00
SH-P-DRH-P	8,254	5.88	13,047	5.24	90	2.38
J	-	-	-	-	-	-

Figure 6: Palulu [2] model size and model generation time, when assisted by immutability classifications. The numbers are sums over indicated subject programs. Models with fewer nodes and edges are better. Also shown are improvement ratios over no immutability information (the "ratio" columns); larger ratios are better. Empty cells indicate that the analysis aborted with an error.

Heuristic **A** (evaluated by the DRBC line) has the greatest effect; removing this heuristic significantly lowers i-recall (as compared to SH-P-DRH-P, which includes all heuristics.) However, because the heuristic is i-unsound, removing it increases i-precision, albeit only by 0.004 (all measurements are for ejc). Heuristic **B** (the DRAC line) increases both i-recall and i-precision, and improves performance by 10%. Heuristic **C** (the DRAB line) is primarily a performance optimization. Including this heuristic results in a 30% performance improvement and a small increase to m-recall.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DR-P	0.777	1.000	0.905	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
SH-P-DRBC-P	0.777	1.000	0.906	0.971
SH-P-DRAC-P	0.927	0.995	0.905	0.971
SH-P-DRAB-P	0.928	0.996	0.906	0.971

6.3.6 *i-sound Analysis Pipelines*

An i-sound mutability analysis never incorrectly classifies a parameter as *immutable*. All our component analyses have i-sound variations, and composing i-sound analyses yields an i-sound staged analysis.

Analysis	i-recall	i-precision	m-recall	m-precision
S	0.454	1.000	0.299	0.998
S-P	0.777	1.000	0.904	0.971
S-P-DRBC-P	0.777	1.000	0.906	0.971
S-P-DBC-P	0.777	1.000	0.912	0.959

S is the i-sound intra-procedural static analysis. Not surprisingly, the i-sound pipelines achieve lower i-recall than i-unsound pipelines presented in Figure 4 (which presents the results for S-P-DRBC-P for all subjects). For clients for whom i-soundness is critical, this may be an acceptable trade-off. In contrast to our analyses, J is not i-sound [33], although it did achieve very high i-precision (see Figure 4).

6.4 Scalability

Figure 5 shows run times of analyses on daikon (185 kLOC, which is considerably larger than subject programs used in previous evaluations of mutability analyses [31, 29, 34]). The experiments were run using a quad-core AMD Opteron 64-bit 4×1.8GHz machine with 4GB of RAM, running Debian Linux and Sun HotSpot 64-bit Server VM 1.5.0_09-b01. Staged mutability analysis scales to large code-bases and runs in about a quarter the time of Sălcianu's analysis (i.e., J in Figure 5).

Figure 5 shows that S-P (Section 6.3.6) runs, on daikon, an order of magnitude faster than J (or even better, if differences in call graph construction are discounted). Moreover, S-P is i-sound, while J is i-unsound. Finally, S-P has high m-recall and m-precision, while J has 0 m-recall and m-precision.

An optimized implementation of the P and DRH stages could run even faster. First, the major cost of propagation (P) is computing the call graph, which can be reused later in the same pipeline. J's RTA [4] call graph construction algorithm takes seconds, but our tool uses Soot, which takes two orders of magnitude longer to perform CHA [14] (a less precise algorithm). Use of a more optimized implementation could greatly reduce the cost of propagation. Second, the DRH step iterates many times, each time performing load-time instrumentation and other tasks that could be cached; without this repeated work, DRH can be much faster than DH. These optimizations would save between 50% and 70% of the total SH-P-DRH-P time.

There is a respect in which our implementation is more optimized than J. J is a whole-program analysis that cannot take advantage of pre-computed mutability information for a library such as the JDK. By contrast, our analysis does so by default, and Figure 5's numbers measure executions that use this pre-computed library mutability information. The number of annotated library methods is less than 10% of the number of methods in daikon.

6.5 Application: Test Input Generation

In addition to evaluating the accuracy of mutability analysis, we evaluated how much the computed immutability information helps a client analysis. We experimented with Palulu [2], a system that generates models for model-based testing. The model is a directed graph that describes sequences of method calls. The model can be pruned (without changing the state space it describes) by removing calls that do not mutate specific parameters, because non-mutating calls are not useful in constructing new test inputs. A smaller model permits a systematic test generator to explore the state space more quickly, or a random test generator to explore more of the state space. Per-parameter mutability information permits more model reduction than method-level purity information.

We ran Palulu on our subject programs using no immutability information, and using immutability information computed by J and by SH-P-DRH-P. Mutability information permitted Palulu to run faster and to generate smaller models. Figure 6 shows the number of nodes and edges in the generated model graph, and the time Palulu took to generate the model (not counting the immutability analysis). This experiment uses J rather than JM, in part because JM runs on more of the programs (3 out of 6), but primarily because JM's results would be no better than J. Palulu models include only methods called during execution, and J starts from the same main method. JM adds more analysis contexts, but doing so never changes a mutable parameter to immutable, which is the only way to improve Palulu model size.

7. Related Work

For reasons of space, we discuss only the most closely related work, which discovers immutability.

Early work [5, 12] on analyzing programs to determine what mutations may occur considered only pointer-free languages, such as Fortran. In such a language, aliases are induced only by reference parameter passing, and aliases persist until the procedure returns. MOD analysis determines which of the reference parameters, and which global variables, are assigned by the body of a procedure. Our static analysis extends this work to handle pointers and object-oriented programs, and incorporates field-sensitivity.

Subsequent research, often called side-effect analysis, addressed aliasing in languages containing pointers. An alias analysis can determine the possible referents of pointers and thus the possible side effects. Ours is the first analysis to address reference immutability, i.e., what references might be used to perform a mutation.

New alias/class analyses lead to improved side effect analyses [32, 29]. Landi et al. [22] improve the precision of previous work by using program-point-specific aliasing information. Ryder et al. [32] compare the flow-sensitive algorithm [22] with a flow-insensitive one that yields a single alias result that is valid throughout the program. The flow-sensitive version is more precise but slower and unscalable, and the flow-insensitive version provides adequate precision for certain applications. Milanova et al. [24] provide a yet more precise algorithm via an object-sensitive, flow-insensitive points-to analysis that analyzes a method separately for each of the objects on which the method is invoked. Object sensitivity outperforms Andersen's context-insensitive analysis [30]. Rountev [29] compares RTA to a context-sensitive points-to analysis for call graph construction, with the goal of improving side-effect analysis. Rountev's experimental results suggest that sophisticated pointer analysis may not be necessary to achieve good results. We, too, compared a sophisticated analysis (Sălcianu's) to a simpler one (ours) and found the simpler one competitive.

Side-effect analysis [10, 31, 24, 29, 34, 33] originated in the compiler community and has focused on i-sound analyses. Our work investigates other tradeoffs and other uses for the immutability information. Specifically, differently from previous research, our work (1) computes both *mutable* and *immutable* classifications, (2) trades off soundness and precision to improve overall accuracy, (3) combines dynamic and static stages, (4) includes a novel dynamic mutability analysis, and (5) permits an analysis to explicitly represent its incompleteness.

Rountev [29] and Sălcianu [34, 33] developed static analyses for determining side-effect-free methods. Like ours, they combine a pointer analysis, an intra-procedural analysis to determine "immediate" side effects, and inter-procedural propagation to determine transitive side effects.

Sălcianu defines a side-effect-free method as one that does not modify any heap cell that existed when the method was called. Rountev's definition is more restricted and prohibits a side-effect-free method from creating and returning a new object, or creating and using a temporary object. Sălcianu's analysis can compute per-parameter mutability information in addition to per-method side effect information. Rountev's coarser analysis results are one reason that we cannot compare directly to his implementation. Rountev applies his analysis to program fragments by creating an artificial `main` routine that calls all methods of interest; we adopted this approach in augmenting J (see Section 6).

Porat et al. [28, 6] infer class immutability for global (static) variables in Java's `rt.jar`, thus indicating the extent to which immutability can be found in practice; the work also addresses sealing/encapsulation. Foster et al. [19] developed an inference algorithm for `const` annotations using Cqual, a tool for adding type qualifiers to C programs. Their algorithm does not handle aliasing. Foster et al. present also a polymorphic version of `const` inference, in which a single reference may have zero or more annotations, depending on the context.

Since we first reported our technique [1], other researchers have also explored the idea of dynamic side-effect analysis. Dallmeier and Zeller (`http://www.st.cs.uni-sb.de/models/jdynpur`, February 2007) developed a tool for offline dynamic side-effect analysis (not parameter mutability) but provide no description of the algorithm or experimental results. Xu et al. [41] developed dynamic analyses for detecting side-effect-free methods. Their work differs significantly from ours. Xu et al. consider only the method's receiver, while our analyses are more fine-grained and produce results for all parameters in the method, including the receiver. Xu et al. examine only one analysis at a time. In contrast, our framework combines the strengths of static and dynamic analyses. Xu et al. do not present an evaluation of the effectiveness of their analyses in terms of precision and recall and only report the percentage of methods identified as pure by their analyses. In contrast, we established the immutability of more than 8800 method parameters by manual inspection and report the results of our 192 analysis combinations with respect to the established ground truth. Finally, Xu et al.'s dynamic analysis is unsound. In contrast, our analysis framework is sound and we provide sound analyses, both static and dynamic, to use in the framework.

8. Conclusion

We have described a staged mutability analysis framework for Java, along with a set of component analyses that can be plugged into the analysis. The framework permits combinations of mutability analyses, including static and dynamic techniques. The framework explicitly represents analysis incompleteness and reports both immutable and mutable parameters. Our component analyses take advantage of this feature of the framework.

Our dynamic analysis is novel, to the best of our knowledge; at run time, it marks parameters as mutable based on mutations of objects. We presented a series of heuristics, optimizations, and enhancements that make it practical. For example, iterative random test input generation appears competitive with user-supplied sample executions. Our static analysis reports both *immutable* and *mutable* parameters, and it demonstrates that a simple, scalable analysis can perform at a par with much more heavyweight and sophisticated static analyses. Combining the lightweight static and dynamic analyses yields a combined analysis with many of the positive features of both, including both scalability and accuracy.

Our evaluation includes many different combinations of staged analysis, in both sound and unsound varieties. This evaluation sheds insight into both the complexity of the problem and the sorts of analyses that can be effectively applied to it. We also show how the results of the mutability analysis can improve a client analysis.

References

[1] Shay Artzi, Michael D. Ernst, David Glasser, and Adam Kieżun. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2006-065, MIT CSAIL, September 18, 2006.

[2] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack:

Generating legal test inputs for object-oriented programs. In *M-TOOS*, October 2006.

[3] Shay Artzi, Adam Kieżun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. Technical Report MIT-CSAIL-TR-2007-020, MIT CSAIL, March 23, 2007.

[4] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, October 1996.

[5] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *POPL*, pages 29–41, January 1979.

[6] Marina Biberstein, Joseph Gil, and Sara Porat. Sealing, encapsulation, and mutability. In *ECOOP*, pages 28–52, June 2001.

[7] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, October 2004.

[8] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[9] Néstor Cataño and Marieke Huisman. Chase: a static checker for JML's *assignable* clause. In *VMCAI*, pages 26–40, January 2003.

[10] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, January 1993.

[11] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[12] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI*, pages 57–66, June 1988.

[13] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[14] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, August 1995.

[15] Brian Demsky and Martin Rinard. Role-based exploration of object-oriented programs. In *ICSE*, pages 313–324, May 2002.

[16] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA*, pages 292–305, October 1996.

[17] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, July 2003.

[18] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, February 2001.

[19] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI*, pages 192–203, June 1999.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[21] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Dept. of EECS, May 5, 2006.

[22] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *PLDI*, pages 56–67, June 1993.

[23] Leonardo Mariani and Mauro Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301, June 2005.

[24] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA*, pages 1–11, July 2002.

[25] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *PPOPP*, pages 167–178, July 2003.

[26] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE*, May 2007.

[27] Marina Polishchuk, Ben Liblit, and Chloë Schulze. Dynamic heap type inference for program understanding and debugging. In *POPL*, January 2007.

[28] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, November 2000.

[29] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, September 2004.

[30] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java based on annotated constraints. In *OOPSLA*, pages 43–55, October 2001.

[31] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, pages 20–36, April 2001.

[32] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM TOPLAS*, 23(2):105–186, March 2001.

[33] Alexandru Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Dept. of EECS, September 2006.

[34] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI*, pages 199–215, January 2005.

[35] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *SOSP*, pages 27–37, December 1997.

[36] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, October 2000.

[37] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, September 2003.

[38] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, August 2006.

[39] Mark Weiser. Program slicing. *IEEE TSE*, SE-10(4):352–357, July 1984.

[40] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP*, pages 380–403, July 2006.

[41] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *PASTE*, June 2007.