Computer Science and Artificial Intelligence Laboratory

Technical Report

MIT-CSAIL-TR-2006-065 September 17, 2006

Combined static and dynamic mutability analysis

Shay Artzi, Michael D. Ernst, David Glasser, and Adam Kiezun

massachusetts institute of technology, cambridge, ma 02139 usa — www.csail.mit.edu

Combined Static and Dynamic Mutability Analysis

Shay Artzi Michael D. Ernst David Glasser Adam Kieżun
MIT CSAIL
{artzi,mernst,glasser,akiezun}@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 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 demonstrate 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 modified, is useful in many software engineering tasks, such as modeling [4], verification [24], compiler optimizations [6, 21], program transformations [12], test input generation [1], regression oracle creation [14, 27], invariant detection [11], specification mining [7], and program comprehension [10].

Different approaches to the mutability problem have different strengths. For example, a static analysis uses static approximations regarding points-to information and which parameters are mutated, whereas a dynamic analysis can observe the actual run-time behavior. A dynamic analysis, however, is limited to the specific observed execution, which may not cover all of the code or exercise it in all possible ways. This paper presents an approach to the mutability problem that works by combining several different mutability analyses, and takes advantage of their relative strengths to create an accurate and scalable analysis.

Previous approaches to mutability analysis [22, 19] were developed with the assumption that a mutable parameter must never be misclassified as *immutable*, even when it leads to misclassifications in the other direction.[1] Because a dynamic analysis cannot prove the absence of a mutation, previous mutability analyses were typically static and could not draw on the many established techniques for dynamic program analysis. To reduce misclassifications, previous techniques used expensive pointer and call-graph construction analyses, which affected their scalability.

In certain contexts, such as many software engineering tasks, scalability and accuracy are key metrics of quality, so the assumptions made by previous work are unnecessarily restrictive. As one example, test suite generators should use methods with side effects in the bodies of test cases, and should use methods without side effects as "observers" in their assertions [14, 27, 1]. Misclassification is not fatal: if a method is mistakenly marked as side-effect-free, this merely reduces test suite coverage, and even if a supposed observer actually modifies some of its parameters, a test case can still be effective in revealing errors. Similar arguments apply to invariant detection and specification mining. Program understanding is yet another domain in which perfect classification is not essential, since humans easily handle some inaccuracy [16]. Human judgment is required in any event, because automatically generated results do not necessarily match programmer abstractions (e.g., automatic analyses do not ignore modifications to caches).

This paper makes the following contributions:

- A staged analysis approach for discovering parameter mutability. The idea of staged analyses is not new, but a staged approach has not been investigated in the context of mutability analysis. Our staged approach is unusual in that it combines static and dynamic stages and it explicitly represents analysis imprecision. The framework is sound, but an unsound analysis may be used as a component, and we examine the tradeoffs involved.
- Mutability analyses. The primary contribution is a novel, dynamic analysis that scales well, yields accurate results (it has a sound mode as well as optional heuristics), and complements existing analyses. We extend the dynamic analysis with random input generation, which improves the analysis results by increasing code coverage. Other contributions include an extension to a state-of-the-art static analysis [22] that improves accuracy while reducing scalability, and a simple static analysis that helps to reveal the costs and benefits of more sophisticated algorithms.

[1] This assumption is justified when the results of mutability analysis are used in a compiler, where such misclassifications may lead to incorrect optimizations.

- Evaluation. We have implemented our framework and analyses for Java, and we investigate the costs and benefits of various techniques, including both our own and that of Sălcianu and Rinard [22]. The results yield insight into the design and implementation of program analysis tools, with a particular emphasis on mutability analysis. For example, one finding is that a well-designed collection of fast, simple analyses can outperform a sophisticated analysis in both scalability and accuracy.

The remainder of the paper is organized as follows. Section 2 describes the problem of inferring parameter immutability. Section 3 presents our staged mutability analysis. Sections 4 and 5 describe the dynamic and static mutability analyses that we developed to be used in the staged analysis. Section 6 describes the experiments we performed to evaluate our work. Section 7 surveys related work, and Section 8 concludes.

2 Parameter and Method Mutability

The goal of mutability analysis is the classification of each method parameter as either *mutable* or *immutable*. Throughout this paper, "parameter" refers to a formal method parameter or receiver. Our implementation also discovers modifications of the program's global state, but in this paper we focus on parameter and method mutability.

Parameter p of method m is *mutable* if some execution of m can change the state of an object that at run-time corresponds to p. If no such execution exists, the parameter p is *immutable*. The change in the state may occur in m itself or in any method that m (transitively) calls. The state of an object o consists of the values of o's primitive (e.g., `int`, `float`) fields and the states of the objects pointed to by o's non-primitive fields. Arrays are treated analogously. A method is *side-effect free* if all its parameters are immutable.

As with previous work [22, 3, 18, 4], our immutability definition excludes aliasing—on top-level entry to a method, all parameters are considered to be fully un-aliased (i.e., point to disjoint parts of the heap). Accounting for all possible side effects under all possible aliasing conditions would yield unusably conservative results. For example, in the following code,

```
void mutateArg1(Cell c1, Cell c2) {
  c1.data = null;
}
```

only the `c1` parameter is mutable, even though a client could call the `mutateArg1` method using the same object for both arguments, as in `mutateArg1(c,c)`.

3 Staged Mutability Analysis

Combining mutability analyses can yield an analysis that has better accuracy than any of the components. In our approach, mutability analyses are combined in stages, forming a "pipeline". (A fixed-point analysis that iterates some stages is also possible.) Each pipeline stage refines the results computed by the previous analysis. An analysis can ignore code that has already been adequately analyzed; 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 sound if it never misclassifies a mutable parameter as *immutable* or vice versa. An analysis is precise if it classifies every parameter as either *mutable* or *immutable*. Our approach permits an analysis to explicitly represent its imprecision. An analysis result classifies parameters into three groups: *mutable*, *immutable*, and *unknown*.

Some previous work [22, 19, 18] has used the term "sound" to denote a weaker concept in which some misclassification is permitted, and provided proofs of this property [22]. For example, in the context of compiler optimizations, misclassifying a mutable parameter as *immutable* can lead to impermissible changes in behavior, but misclassification in the other direction does not lead to such errors. Such an analysis could be made sound by converting all *mutable* outputs to *unknown* (and sometimes that is the implied semantics even when the outputs are labeled *immutable* and *mutable*). Having only two output classifications loses information by conflating parameters that are known to be mutable with those where analysis approximations prevent definitive classification. As noted earlier, in many software engineering contexts, misclassification in either direction is undesirable but not fatal.

In a staged analysis, the input to the first stage in the staged analysis is an initial classification of all parameters (e.g., all *unknown*, with the possible exception of pre-analyzed standard libraries). The output of the last stage is the final classification, in which some parameters may remain *unknown*.

4 Dynamic Analysis

Our dynamic mutability analysis observes the program's execution and classifies as *mutable* those method parameters that correspond to actually mutated objects. It can soundly classify all other parameters as *unknown*, or it can optionally use heuristics that allow it to classify certain parameters as *immutable* or that improve its run time with some risk of unsoundness. The analysis has an iterative variation with random input generation for increased effectiveness and precision.

4.1 Dynamic Analysis Algorithm

Conceptually, the analysis tags each reference (*not* each object—more than one reference can point to the same object) in the running program with the set of all formal pa-

rameters (from any method invocation on the call stack) whose fields were directly or indirectly accessed to obtain the reference. Primitives need not be tagged, as they are all immutable.

For example, consider the method `mutateArg1` of Section 2. If a client calls `mutateArg1(a,a)`, then the algorithm observes the execution and classifies only parameter `c1` as *mutable*.

Due to space constraints, we give the flavor of the dataflow rules with a few examples.

1. On method entry, each formal parameter (that is classified as *unknown*) is added to the parameter set of the corresponding actual argument reference.
2. On method exit, all parameters for the current invocation are removed from all references in the program.
3. Assignments propagate the parameter set unchanged.
4. Field accesses also propagate the set unchanged: the set of parameters for `x.f` is the same as that of `x`.
5. For a field write `x.f = v`, the analysis classifies as *mutable* all the parameters in the set for `x`.

4.2 Dynamic Analysis Heuristics

The dynamic analysis algorithm described in Section 4.1 is sound—a parameter is classified as *mutable* only if it is modified during execution. We found the sound algorithm to be impractical for two reasons. First, the algorithm does not take advantage of the *absence* of parameter modifications. It never classifies any parameters as *immutable*, even if a method was executed a million times without modifying its parameters. Second, the run time of the algorithm suffers due to the need to maintain reference tags.

We addressed these problems by implementing heuristics. The heuristics potentially introduce unsoundness to the analysis, but in practice, they cause few misclassifications. The heuristics are also critical in achieving acceptable run time. We developed two kinds of heuristics.

The first kind classifies *unknown* parameters as *immutable*.

(**A**) **Classifying parameters as *immutable* at the end of the analysis.** This heuristic classifies as *immutable* all (*unknown*) parameters of methods that were executed during the dynamic analysis.

A related pipeline stage is A', which classifies as *immutable* all (*unknown*) parameters of methods that were executed during some dynamic analysis. This heuristic differs from **A** in that it is not applied during the dynamic analysis, but rather at any point in a pipeline—most sensibly, at the end, giving other analyses a chance to discover mutable parameters before A' classifies them as *mutable*. A' has no effect when heuristic **A** is enabled, because the set of parameters classified as *immutable* by **A** contains the all parameters that would be classified by A'.

(**B**) **Classifying parameters as *immutable* after a fixed number of calls.** This heuristic classifies as *immutable* all *unknown* parameters after a fixed number N of calls to their method (our experiments used $N = 10$). Alternatively, it could stochastically sample subsequent executions. The heuristic allows the analysis to ignore a parameter (i.e., not look for mutations via it, thus reducing computational load). This heuristic is especially useful for methods that are invoked millions of times without ever modifying (some of) their arguments.

The second kind of heuristic classifies otherwise *unknown* parameters as *mutable*:

(**C**) **Using current mutability classification.** When an object is passed to a formal parameter that is already classified as *mutable*, the object is treated as if it were mutated immediately (i.e., without waiting for a field write), and is not tracked thereafter. A misclassification is possible if the object would have not actually have been changed by this execution of the method.

(**D**) **Classifying aliased mutated parameters.** This heuristic classifies a parameter p as *mutable* if the object that p points to changes state, 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.

This heuristic permits a more efficient dynamic analysis algorithm: when a method m is called during the program's execution, the analysis computes the set $r(m, p)$ 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 $o \in r(m, p)$, then the analysis classifies p as *mutable*.

Our implementation includes the following three optimizations, which improve run time by over 30×. First, the algorithm determines reachability by maintaining and traversing its own data structure that mirrors the heap; this is faster than using reflection. Second, the set of reachable objects is computed lazily, when a modification occurs. Third, the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set changes.

Our implementation performs load-time instrumentation and works online (in tandem with the target program, without creating a trace file).

4.3 Input to the Dynamic Analysis

The inputs to the dynamic mutability analysis are the current classification and an example execution. The example execution can be provided by the user or it can be randomly generated.

An execution provided by a user may exercise the program in ways in which a randomly generated code may not

be able to, such as by creating and mutating complex objects. However, dynamic mutability analysis is limited only to the part of the program that is covered by the example execution.

To analyze larger parts of the program, or even eliminate the need for a sample execution, the dynamic mutability analysis can generate random sequences of method calls [17]. The generator gives higher probability to to methods with *unknown* parameters and methods that have not yet been executed by other dynamic analyses in the pipeline. By default, the number of generated tests for each round is $max(500, \#methodsInProgram)$.

Once 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. Iterations continue as long as at least 1% of the *unknown* parameters are classified (the threshold is user-settable).

5 Static 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, and P, an interprocedural analysis that propagates mutability information along a graph of dependencies between method parameters. 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 a coarse intraprocedural pointer analysis that calculates the parameters pointed to by each local variable. We built a new analysis rather than re-using a previous implementation primarily to explore trade-offs in analysis complexity and accuracy.

5.1 Intraprocedural Points-To Analysis

The static analysis computes the mutability of each method parameter. Thus, the analysis must determine which parameters can be pointed to by which locals. Both phases of the static analysis use a coarse points-to analysis to determine this. For simplicity and scalability, we have made the design choices that the pointer analysis should be purely intraprocedural, flow-insensitive, and not incorporate an escape analysis.

The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters that l can point to directly and a set $P(l)$ of parameters that l can point to directly or transitively. (In either case, l can be pointing either to the parameter object itself or to an object that the parameter points to transitively through its fields.) The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated (see below).

The points-to analysis executes a fixpoint computation, calculating for each local l and parameter p the minimum number $D(l, p)$ of dereferences (possibly 0) that can be applied to l to find an object pointed to by p. At the beginning of the computation, $D(l, p)$ is ∞ for all l and p. After the computation reaches a fixpoint, it sets $P(l) = \{p \mid D(l, p) \neq \infty\}$ and $P_0(l) = \{p \mid D(l, p) = 0\}$. Due to space constraints, we give the flavor of the dataflow rules with a few examples:

- Direct assignments of a parameter p to a local l such as $l =$ `this` result in setting $D(l, p) = 0$.
- A field assignment $l_1.f = l_2$ results in setting
  $D(l_1, p) = \min(D(l_1, p), D(l_2, p) + 1)$
  $D(l_2, p) = \min(D(l_2, p), D(l_1, p) - 1)$
- 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 $D(l, p)$ are changed. 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 $D(l, p) = \min_{l' \in S} D(l', p)$.

5.2 Intraprocedural Phase: S

The intraprocedural phase first calculates the "overestimate" points-to analysis described in Section 5.1.

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*. (Array mutations are treated analogously.)

Next, S marks as *immutable* some parameters that are currently *unknown*. 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`). The analysis then marks as *immutable* all *unknown* parameters that are not in the set $\cup_{l \in L} P(l)$.

S never marks any parameter as *immutable* if the parameter can be referred to in a mutation or escape to another method body, so the analysis never mistakenly marks a parameter as *immutable*. However, because its pointer analysis is an overestimate, S can mark parameters as *mutable* that are actually immutable. For example, `l.f = this; l.g++` will lead to $P_0(l) = \{$`this`$\}$, and so the receiver will be marked as *mutable* after the second statement. (This is unavoidable with a pointer analysis that just returns sets of parameters for each local; a more nuanced pointer analysis could avoid this problem.)

5.3 Interprocedural Propagation Phase: P

The interprocedural propagation phase constructs a call-graph for the analyzed program and uses the graph to refine the parameter classification given as input, by propagating

both mutability and immutability information. This phase does not itself search in the program for mutations; it merely "distributes" the information that is already available. Propagation is sound in classifying parameters as *immutable*—given a correct input classification and a precise call-graph, propagation never misclassifies a mutable parameter as *immutable*.

Our algorithm is parametrized by a call-graph construction algorithm. Our experiments used CHA [8]—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 [2] (available in Soot, but containing bugs that prevented us from using it), or those proposed by Tip and Palsberg [23] (not implemented in Soot).

5.3.1 Parameter Dependency Graph

The propagation uses a *Parameter Dependency Graph* (PDG). It is a directed graph, similar to a call graph. In a PDG, each node is a method parameter `m.p`. An edge from `m1.p1` to `m2.p2` exists iff `m1` calls `m2`, passing as position `p2` either `p1` or an object that may be transitively pointed-to by `p1`.

We create PDGs 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 true PDG is not computable, because determining perfect aliasing and call information is undecidable. Our analysis uses an under-approximation and an over-approximation to the PDG as safe approximations for determining mutable and immutable parameters, respectively. Our over-approximation (i.e., it contains a superset of edges of the ideal graph) is called the *fully-aliased* PDG, which is created with an overestimating points-to analysis which assumes that method calls introduce aliasings between *all* parameters. Our under-approximation (i.e., it contains a subset of edges of the ideal graph) is the *un-aliased* PDG, which is created with an underestimating points-to analysis which assumes that method calls introduce *no* aliasings between parameters.

To construct the under-approximation of the true PDG, propagation needs a call-graph that is an under-approximation of the real call-graph. However, most existing call-graph construction algorithms [8, 9, 2, 23] create an over-approximation. Therefore, our implementation uses the same call-graph for building the un- and fully-aliased PDGs. In our experiments, this never caused misclassification of parameters as *immutable*, and only a minimal number of parameters misclassified as *mutable*.

5.3.2 Propagation Algorithm

Propagation starts with the initial classifications of parameters, and refines the classification in 2 phases.

In the **mutability propagation** phase, the analysis classifies additional parameters as *mutable*. It classifies as mutable all the *unknown* parameters in the PDG that can reach in the graph (flow to in the program) a parameter that is classified as *mutable*. To propagate mutability, the analysis uses the under-approximation to the PDG (the un-aliased PDG, introduced above). Using an over-approximation to the PDG would be unsound because spurious edges may lead propagation to incorrectly classify parameters as mutable.

In the **immutability propagation** phase, the analysis classifies additional parameters as *immutable*. This phase uses a fix point computation: in each step, the analysis classifies as *immutable* all *unknown* parameters that have no *mutable* or *unknown* successors (callees) in the PDG. To soundly propagate immutability, the analysis must use an over-approximation to the PDG (the fully-aliased PDG, introduced above). Otherwise, if an edge is missing in the PDG, 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 PDG, a *mutable* successor.

Because propagation ignores the bodies of methods, the `P` phase is 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 be run. However, it can be run multiple times (with other analyses in between).

6 Evaluation

We experimentally evaluated 168 combinations of mutability analyses, comparing the results with each other and with a manually computed (and inspected) optimal classification of parameters. Our results indicate that staged mutability analysis can be accurate, scalable, and useful.

6.1 Methodology

We performed our experiments on 6 open-source subject programs (see Figure 1). When an example input was needed (e.g., for a dynamic analysis), we ran each subject program on a single, small input.

- **jolden**[2] 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 [22].
- **tinysql**[3] is a minimal SQL engine. We used the program's test suite as the example input.

[2] `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`
[3] `http://sourceforge.net/projects/tinysql`

program	LOC	classes	parameters	
			all	non-trivial
jolden	6215	56	705	470
sat4j	15081	122	1499	1136
tinysql	32149	119	2408	1708
htmlparser	64019	158	2270	1738
eclipsec	107371	320	9641	7936
daikon	185267	842	16781	13319
Total	410102	1617	33304	26307

Figure 1. Subject programs.

- **htmlparser**[4] is a real-time parser for HTML. We used our research group's webpage as the example input.
- **sat4j**[5] is a SAT solver. We used a file with an unsatisfiable formula[6] as the example input.
- **eclipsec**[7] is the Java compiler supplied with the Eclipse project. The example input was JLex[8] (one file with 7841 LOC).
- **daikon**[9] is an invariant detector. We used a test case included in the distribution as the example input.

As the input to the first analysis in the pipeline, we used a pre-computed (manually created) classification for all parameters in the standard Java libraries. The use of a pre-computed classification is justified because it has to be created only once, and it can be reused many times. Additionally, the pre-computed classification covers otherwise unanalyzable code, such as native calls.

We measured the results only for non-trivial parameters declared in the application. We ignored parameters declared in external or JDK libraries, and ignored all parameters with a primitive, boxed primitive, or `String` type.

For jolden and eclipsec, we manually determined the optimal classification (*mutable* or *immutable*) for all parameters. For each other program, we used a random number generator to choose 5 classes, then manually determined the optimal classification for each parameter in those classes. In all, we manually classified over 8600 parameters, spot-checking many of these later. Knowing the optimal classification allowed us to measure the accuracy of each mutability analysis.

6.2 Evaluated Analyses

Our experiments use the following component analyses:

- S is our intraprocedural static analysis (Section 5.2).
- P is our interprocedural static propagation (Section 5.3).
- D is our dynamic analysis (Section 4).
- DH is D, augmented with all the heuristics described in Section 4.2. DA, DB, and DC are D, augmented with just one of the heuristics.
- DRH is DH enhanced with random input generation (Section 4.3); likewise for DRA, etc.
- A' classifies executed parameters never observed to be mutated, as described in Section 4.2.
- JPPA is Sălcianu and Rinard's state-of-the-art static analysis [22]. It never classifies parameters as *mutable*—only *immutable* and *unknown*.
- JPPAM is the classification computed by JPPA using a `main` method that contains calls to all the public methods in the subject program (including the original `main` method); this increases coverage. Since JPPA is a whole program analysis, it only analyzes methods that are reachable from the `main`.
- JPPAH is JPPA augmented with a heuristic to classify as *mutable* any parameter in which JPPA's explanation of its *unknown* classification is a specific potential modification (as opposed to unanalyzed code or other reasons).
- JPPAMH is the same as JPPAM but uses the heuristics from JPPAH to classify parameters as *mutable*.

X-Y-Z denotes the staged analysis in which component analysis X is followed by component analysis Y and then by component analysis Z.

6.3 Experimental Observations

We experimented with 6 programs and 168 different analysis pipelines. This section discusses several observations that stem from the results of our experiments. We present the results in a way that helps us to determine the best combined static and dynamic analysis.

We compared the computed classifications to the manually-computed optimal one. The **correct** category collects cases where an analysis correctly classified a mutable parameter as *mutable*, or an immutable one as *immutable*. The **imprecise** are cases where an analysis classified a parameter as *unknown*. The **misclassified** are cases where an analysis incorrectly classified an immutable parameter as *mutable*, or vice versa.

The tables in this section present results for eclipsec. Results for other programs were similar. For smaller programs, all analyses did better, so the differences in the analyses were not as pronounced.

6.3.1 Effect of Interprocedural Propagation

The propagation step P of Section 5.3 requires computing a call graph; thereafter, it is fast (see Section 6.5). Although it is the major source of misclassification in our analyses, it increases correct classifications by substantially more.

[4] `http://htmlparser.sourceforge.net/`
[5] `http://www.sat4j.org/`
[6] `ftp://dimacs.rutgers.edu/`
[7] `http://www.eclipse.org/`
[8] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`
[9] `http://pag.csail.mit.edu/daikon/`

Analysis	correct %	imprecise %	misclassified %
S	42.0	57.6	0.4
S-P	83.7	13.7	2.6
S-P-DRH	89.9	7.3	2.8
S-P-DRH-P	90.2	7.0	2.8

In the sequel, we always run P after every stage that follows a S stage. For example, D-S-DRH is shorthand for D-S-P-DRH-P (we sometimes explicitly state the P stages for emphasis).

6.3.2 Combining Static and Dynamic Analysis

We evaluated the effect of combining static and dynamic analysis. The following table shows representative results. For brevity, we present the results only for DRH (the best dynamic analysis; see Sections 6.3.3 and 6.3.4) and not for all combinations of dynamic analyses.

Analysis	correct %	imprecise %	misclassified %
DRH	33.1	56.5	10.5
DRH-S	81.7	6.9	11.4
S	83.7	13.7	2.6
S-DRH	90.2	7.0	2.8

Combining static and dynamic analysis in either order is helpful—the two types of analysis are complementary. However, for best results, the static stage should precede the dynamic stage. (This has positive effects on run time, as well.) Therefore, in the sequel, we will always present pipelines that start with a static analysis.

A dynamic analysis can improve even a very accurate static analysis.

Analysis	correct %	imprecise %	misclassified %
JPPA	27.5	72.5	0.0
JPPA-DRH	42.1	47.9	10.0
JPPA-S	89.3	10.5	0.2
JPPA-S-DRH	93.6	6.0	0.4
JPPAM	34.1	65.9	0.0
JPPAM-DRH	46.2	43.2	10.6
JPPAM-S	91.2	8.8	0.0
JPPAM-S-DRH	94.5	5.1	0.4
S	83.7	13.7	2.6
S-DRH	90.2	7.0	2.8

Adding a dynamic stage to the sophisticated JPPA analysis reduces imprecision from 72.5% to 47.9%. Including the simple static analysis S helps even more, in part by classifying missed parameters and in part by enabling the propagation P to be run. Adding the dynamic analysis tended to level the playing field: it provided greater improvement to less accurate analyses.

6.3.3 Random Input Generation

We evaluated the use of random executions, instead of executions provided by the user, by the dynamic mutability analysis. We compared pipelines that use DH with those that use DRH, DH-DRH, or DRH-DR instead.

Analysis	correct %	imprecise %	misclassified %
S-DH	86.9	9.3	3.8
S-DH-DRH	90.6	5.5	3.9
S-DRH	90.2	7.0	2.8
S-DRH-DH	91.1	5.5	3.4

Using both a user-supplied execution and random executions fares best. However, those results are very little better than only using a random execution; since it also had the lowest misclassification rate, we consider it best. Using only a user-supplied execution comes in a distant fourth place.

Random input generation is able to explore parts of the program that the user-supplied execution may not have. Furthermore, the tool requires only the program's source code—the user is not forced to select a representative execution. Depending on the size of the sample execution, the analysis may run faster as well.

The surprising finding that randomly generated code is as effective as using an example execution suggests that other dynamic analyses might also benefit from replacing example executions with random executions.

6.3.4 Dynamic Analysis Heuristics

By exhaustive evaluation, we determined that each of the heuristics except for A' is beneficial: DRH is the best dynamic analysis. This section indicates the unique contribution of each heuristic, by removing it from the optimal combination (because some heuristics, e.g., **A** and **B**, have overlapping benefits). As noted in Section 4.2, heuristic **D** is always enabled in our current implementation, and heuristic **A** is mutually exclusive with A'.

Analysis	correct %	imprecise %	misclass'd %	time
S-DR	83.7	13.6	2.7	140.7
S-DRBC	89.3	7.7	3.0	128.4
S-DRBC-A'	90.3	6.7	3.0	128.4
S-DRAC	90.1	7.0	2.9	143.7
S-DRAB	90.2	7.0	2.8	135.4
S-DRH	90.2	7.0	2.8	132.9

Heuristic **A** (evaluated by the DRBC line) has the greatest effect. Replacing it by A' decreases accuracy. Heuristics **B** and **C** are primarily performance optimizations. The table shows that they have no effect on the eclipsec results, and that they reduce time (the time shown is for the first iteration of DR, and **B** helps most). The benefits of the heuristics are greater on sample executions (D) than on random executions (DR, which is shown). For example, **B** alone improves the run time of the dynamic analysis by an order of magnitude on the eclipsec example execution.

Prog.	Analysis	correct %			imprecise %			misclass'd %		
		i/i	m/m	sum	u/i	u/m	sum	m/i	i/m	sum
eclipsec	Optimal	45.9	54.1	100.0	0.0	0.0	0.0	0.0	0.0	0.0
	JPPA	27.5	0.0	27.5	18.4	54.1	72.5	0.0	0.0	0.0
	JPPAMH	34.1	54.0	88.1	7.9	0.1	8.0	3.9	0.0	3.9
	JPPAMH-S-DRH	42.0	54.0	96.0	0.0	0.0	0.0	4.0	0.0	4.0
	S-P	35.0	48.6	83.7	8.3	5.5	13.7	2.6	0.0	2.6
	S-P-DRH-P	41.4	48.8	90.2	1.9	5.1	7.0	2.7	0.1	2.8
jolden	Optimal	73.1	26.9	100.0	0.0	0.0	0.0	0.0	0.0	0.0
	JPPA	65.0	0.0	65.0	8.1	26.9	35.0	0.0	0.0	0.0
	JPPAMH	72.2	22.5	94.7	0.3	4.4	4.7	0.6	0.0	0.6
	JPPAMH-S-DRH	70.7	28.8	99.5	0.0	0.0	0.0	0.5	0.0	0.5
	S-P	60.3	24.4	84.7	11.9	2.5	14.4	0.8	0.0	0.8
	S-P-DRH-P	70.0	26.9	96.9	1.1	0.0	1.1	1.9	0.0	1.9
daikon	Optimal	60.3	39.7	100.0	0.0	0.0	0.0	0.0	0.0	0.0
	JPPA	42.5	0.0	42.5	17.8	39.7	57.5	0.0	0.0	0.0
	JPPAMH	-	-	-	-	-	-	-	-	-
	JPPAMH-S-DRH	-	-	-	-	-	-	-	-	-
	S-P	38.4	37.0	75.3	15.1	2.7	17.8	6.8	0.0	6.8
	S-P-DRH-P	42.5	37.0	79.5	11.0	2.7	13.7	6.8	0.0	6.8
tinysql+sat4j+htmlparser	Optimal	76.6	23.4	100.0	0.0	0.0	0.0	0.0	0.0	0.0
	JPPA	-	-	-	-	-	-	-	-	-
	JPPAMH	-	-	-	-	-	-	-	-	-
	JPPAMH-S-DRH	-	-	-	-	-	-	-	-	-
	S-P	63.3	20.4	83.7	10.8	3.0	13.8	2.5	0.0	2.5
	S-P-DRH-P	72.7	22.4	95.1	1.2	0.0	1.2	2.7	1.0	3.7

Figure 2. Mutability analyses on subject programs. The columns illustrate six outcomes of comparing the computed classification with the optimal one. Case **m/m** (**i/i**) is when the analysis correctly classifies an (im)mutable parameter. Case **u/m** (**u/i**) is when the analysis classifies an (im)mutable parameter as *unknown*. Case **m/i** is when the analysis incorrectly classifies an immutable parameter as *mutable* (**i/m** is the opposite). Empty cells mean that the analysis finished with an error. As noted in Section 6.3.1, in this table JPPAMH-S-DRHstands for JPPAMH-S-P-DRH-P.

6.4 Accuracy of Staged Analyses

Figure 2 compares the accuracy of several mutability analyses. The two programs on which JPPA and JPPAM work are given separately; other programs are grouped by whether those analyses work. We make several observations.

(1) The results indicate that a staged mutability analysis, combined of static and dynamic phases, can achieve better accuracy than a complex static analysis. The results achieved by S-P-DRH-P are better than those achieved by JPPA (90.2% vs 27.5% correctly classified parameters).

(2) 45.9% of the non-trivial parameters in eclipsec are immutable, but JPPA detects less than half of these. By contrast, Rountev [18] detected 97.5% or 100% of pure methods (not parameters) in a set of data structures. On the smaller jolden programs, JPPA finds 89% of the immutable parameters.

(3) Our very simple static analysis, S-P, outperforms JPPA on eclipsec: S-P finds over 3/4 of the immutable parameters instead of under 1/2. (Both analyses are sound with respect to *immutable* classifications.) S-P is slightly worse than JPPA on jolden and daikon, but is simpler and more scalable.

Analysis	total	last component
JPPA	5586	5586
JPPAM	error	n/a
DH	318	318
S	167	167
S-P	564	397
S-P-DH	859	295
S-P-DH-P	869	10
S-P-DRH	1484	920
S-P-DRH-P	1493	9

Figure 3. Run time, in seconds, of analyses on daikon: both the cumulative time and the time for the last analysis in the pipeline. All P stages are explicit in this figure. The experiments were run using a 3GHz machine with 3GB of RAM, running Debian Linux and Sun JVM 1.5.0-b64.

(4) Our preferred staged analysis, S-P-DRH-P, always finds at least as many *immutable* parameters as JPPA.

(5) **i/m** misclassifications are generally considered worse than **m/i** ones. S-P-DRH-P has none of the bad misclassifications on jolden or daikon, and only 0.1% (10 parameters) on eclipsec. These parameters are all modifiable on execution paths not taken by any execution.

(6) Staged analysis misclassifications in the **m/i** direction are modest—typically only a few percent. JPPA avoids these misclassifications by never issuing any *mutable* classification, but this dramatically reduces overall accuracy.

6.5 Scalability

Figure 3 shows run times of analyses on daikon (185 kLOC, which is larger than previous evaluations [19, 18, 22]). Staged mutability analysis scales to large code-bases and runs in about a quarter the time of JPPA; augmented versions of JPPA are even slower.

The figure overstates the cost of both the P and DRH stages, due to limitations of our implementation. First, the major cost of propagation (P) is computing the PDG, which can be reused later in the same pipeline. Sălcianu says that JPPA's RTA call graph construction algorithm takes seconds, and our tool takes two orders of magnitude longer to to perform CHA (a less precise algorithm) using Soot. 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; DRH can be much faster than DH. These algorithmic fixes would save between $1/2$ and $2/3$ of the total S-P-DRH-P time.

JPPA focuses on *immutable* classifications and is sound with respect to them. In our experiments, S-P is as sound as JPPA and classifies about as many parameters as *immutable*

analysis	nodes	edges	time (s)
jolden + eclipsec + daikon			
no immutability	444729	624767	6703
JPPA	131425	210354	4626
S-DRH	124601	201327	4271
htmlparser + tinysql + sat4j			
no immutability	48529	68402	215
JPPA	-	-	-
S-DRH	8254	13047	90

Figure 4. Palulu model size and run time, when assisted by immutability classifications. Smaller models are better.

(more for eclipsec, fewer for daikon)—yet it runs an order of magnitude faster (or even better, if differences in call graph construction are discounted).

6.6 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 [1], a system that generates tests based on a model. The model is a directed graph that describes permitted sequences of method calls. The model can be pruned (without changing the space it describes) by removing calls that do not mutate specific parameters; non-mutating calls are not useful in constructing complex 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.

We ran Palulu on our subject programs using no immutability information, and immutability information computed by S-P-DRH-P and JPPA. Figure 4 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). Because JPPA did not run for all programs, Figure 4 has two parts.

Purity information permitted Palulu to run faster and to generate smaller models. On the programs which JPPA was able to analyze, both evaluated analyses helped Palulu to create significantly smaller models than running without purity information; the two analyses created models of very similar sizes. Our staged analysis was able to run on several programs which JPPA could not analyze, and similarly helped Palulu on those programs.

7 Related Work

Previous side effect analyses [5, 19, 15, 18, 22, 21] have largely originated in the compiler community, and so their focus has been quite different than ours. This different focus led to different tradeoffs in their design and implementation, most notably acceptance of imprecision in return for soundness of *immutable* classifications. Our work investigates other tradeoffs and other uses for the information. For instance, our work includes a novel dynamic analysis, it combines dynamic and static stages, it aims to compute both *mutable* and *immutable* classifications, it focuses on overall accuracy and permits an analysis to explicitly represent its precision, and it is scalable to substantial programs.

Our static analysis stage is similar in flavor to, but simpler than, other static analyses for Java. This makes it more scalable, but potentially less accurate. Our comparison focuses on the most recent projects.

Our analysis is similar to Rountev's [18] in that it combines pointer analysis (it permits an arbitrary pointer analysis to be plugged in), intraprocedural analysis to determine "immediate" side effects, and propagation to determine transitive side effects. Rountev applies his analysis to program fragments by creating an artificial `main` routine that calls all methods of interest; we adopted this approach in enhancing JPPA. One difference is that Rountev's analysis computes only one side-effect bit per method rather than determining per-parameter mutability. In experiments on 7 components with 9–35 public methods, it found all 40 side-effect free methods using a context-sensitive pointer analysis and 39 methods using the less precise RTA algorithm. This suggests that very sophisticated pointer analysis may not be necessary to achieve good results. Inspired by this result, we have extended the comparison to an even less precise static analysis. Our focus is on the evaluation, not on the novelty of the static analysis. (This mirrors other work questioning the usefulness of very sophisticated pointer analysis, e.g., [20, 13].) We have been unable to run Rountev's analysis on our subject programs, but hope to perform such a comparison in the future.

Sălcianu's analysis uses a more sophisticated pointer analysis, so it represents a different tradeoff. It is also based on an intra- and an inter-procedural stage. Its flow-insensitive method summary specially represents objects allocated by the current method invocation, so (unlike previous analyses) a pure method may to perform side effects on a newly-allocated but non-captured objects. It is compositional, analyzing methods separately and without knowing their calling context, then combining their summaries. (This feature is intended for program fragments rather than for arbitrary code that could be placed in a context in which callbacks (e.g., `toString`) can have arbitrary effects.) It handles unanalyzable calls such as native methods. Sălcianu gives a proof of correctness of his analysis. We evaluated against Sălcianu's implementation, which, like ours, computes parameter immutability.

Tschantz [25] presents an inference algorithm for reference immutability (that is, a `readonly` type qualifier), along with experiments regarding its accuracy. Reference immutability [26] is different than our definition of parameter mutability (which is shared with previous work), and neither one subsumes the other. Tschantz's type-based analysis forces all objects referenced by a variable to have the same type (even if the variable is reassigned). However, it

permits (and his tool infers) `assignable` and `mutable` annotations indicating when a field is not part of the abstract state of an object. This permits code that uses caches to be properly annotated as free of side effects.

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 imprecision, and this makes it possible to compute 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 is primitive, and permits us to investigate tradeoffs regarding analysis complexity and precision. To our surprise, it performs 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 analysis results can improve models created by a client analysis.

References

[1] S. Artzi, M. D. Ernst, A. Kieżun, C. Pacheco, and J. H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. Technical Report MIT-CSAIL-TR-2006-056, MIT CSAIL, Sept. 5, 2006.

[2] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *OOPSLA*, pages 324–341, Oct. 1996.

[3] A. Birka and M. D. Ernst. A practical type system and language for reference immutability. In *OOPSLA*, pages 35–49, Oct. 2004.

[4] L. Burdy, Y. Cheon, D. Cok, M. D. Ernst, J. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, June 2005.

[5] J.-D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *POPL*, pages 232–245, Jan. 1993.

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

[7] V. Dallmeier, C. Lindig, A. Wasylkowski, and A. Zeller. Mining object behavior with ADABU. In *WODA*, pages 17–24, May 2006.

[8] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, Aug. 1995.

[9] A. Diwan, J. E. B. Moss, and K. S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *OOPSLA*, pages 292–305, Oct. 1996.

[10] J. J. Dolado, M. Harman, M. C. Otero, and L. Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE TSE*, 29(7):665–670, 2003.

[11] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE TSE*, 27(2):99–123, Feb. 2001.

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

[13] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE*, pages 54–61, June 2001.

[14] L. Mariani and M. Pezzè. Behavior capture and test: Automated analysis of component integration. In *ICECCS*, pages 292–301, 2005.

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

[16] G. C. Murphy, D. Notkin, and E. S.-C. Lan. An empirical study of static call graph extractors. In *ICSE*, pages 90–99, Mar. 1996.

[17] C. Pacheco, S. Lahiri, M. Ernst, and T. Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, Sept. 2006.

[18] A. Rountev. Precise identification of side-effect-free methods in Java. In *ICSM*, pages 82–91, Sept. 2004.

[19] A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *CC*, pages 20–36, Apr. 2001.

[20] E. Ruf. Context-insensitive alias analysis reconsidered. In *PLDI*, pages 13–22, June 1995.

[21] A. Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Dept. of EECS, Sept. 2006.

[22] A. Sălcianu and M. C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI*, pages 199–215, Jan. 2005.

[23] F. Tip and J. Palsberg. Scalable propagation-based call graph construction algorithms. In *OOPSLA*, pages 281–293, Oct. 2000.

[24] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE*, pages 188–197, Sept. 2003.

[25] M. S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Dept. of EECS, Aug. 2006.

[26] M. S. Tschantz and M. D. Ernst. Javari: Adding reference immutability to Java. In *OOPSLA*, pages 211–230, Oct. 2005.

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