Transferring State-of-the-art Immutability Analyses: Experimentation Toolbox and Accuracy Benchmark

Benjamin Holland, Ganesh Ram Santhanam, and Suresh Kothari
Department of Electrical and Computer Engineering, Iowa State University, Ames, Iowa 50011
Email: {bholland, gsanthan, kothari}@iastate.edu

***Abstract*—Immutability analysis is important to software testing, verification and validation (V&V) because it can be used to identify independently testable functions without side-effects. Existing tools for immutability analysis are largely academic prototypes that have not been rigorously tested for accuracy or have not been maintained and are unable to analyze programs written in later versions of Java. In this paper, we re-implement two prominent approaches to inferring the immutability of an object: one that leverages a points-to analysis and another that uses a type-system. In addition to supporting Java 8 source programs, our re-implementations support the analysis of compiled Java bytecode. In order to evaluate the relative accuracy, we create a benchmark that rigorously tests the accuracy boundaries of the respective approaches. We report results of experiments on analyzing the benchmark with the two approaches and compare their scalability to real world applications. Our results from the benchmark reveal that points-to based approach is more accurate than the type inference based approach in certain cases. However, experiments with real world applications show that the points-to based approach does not scale well to very large applications and a type inference based approach may offer a scalable alternative.**

Toolbox: https://ensoftcorp.github.io/immutability-toolbox
Benchmark: https://kcsl.github.io/immutability-benchmark

I. Introduction

Many applications in software analysis, testing and verification call for tools that analyze the immutability of objects. For instance, in software testing it is useful to first identify independently testable functions that do not have side-effects (according to Sălcianu and Rinard [1], a function has a side-effect if it mutates an object that existed prior to its invocation), as it allows the tester to focus on more complex areas of the codebase. For security verification, immutability analysis is important because information leakage via space or time based side channels is primarily enabled through the presence of functions with observable side-effects [2], [3]. Furthermore, knowing that an object is immutable is particularly useful in verifying concurrent programs because an immutable object cannot be corrupted by thread interference.

An object is mutated when any of the object's fields are updated, i.e., a field assignment is made after the object is initialized. In Java, an object can only be accessed through a reference, so *immutability analysis* boils down to answering the question: *given code C and reference R, is the object O referenced by R mutated in the code C?*.

A. Immutability in Practice

An important question to ask is whether a dedicated immutability analysis is needed, given that Java already offers a mechanism for enforcing object immutability using the `final` keyword, which would require all fields (and the fields of fields) of an object to be marked `final`. First, marking an object immutable using the `final` keyword is impractical for general purpose programming, because it would lock the object's initialized state and prevent any further updates to the initialized object. Instead, it is more common for developers to define mutable object types, but treat particular instances of the object as immutable objects by simply not updating the object's fields. In fact, many IDEs encourage this design pattern by offering automatic generation of getter and setter functions for object fields. Second, our survey of 274,504 Java projects on Github using the Boa [4] framework showed that 38.45% of all types are potentially mutable having at least one non-final field [5]. Finally, there is no language mechanism for enforcing the immutability of array components. Hence, there is a need for reliable tools for recovering the immutability of objects, with applications to software testing and V&V.

B. State-of-the-art approaches

Tools published in academic research related to immutability analysis [1], [6], [7], [8], [9] use one of two prominent approaches to immutability analysis: one employs a points-to analysis to infer immutability and another uses a type based inference system. A points-to based immutability analysis takes a top-down approach by first resolving aliasing relationships and then using them to identify mutations to the aliased objects. The type inference based immutability analysis takes a bottom-up approach by identifying mutation at field assignments, and inferring the objects that may be affected by the mutation.

Our survey of the state-of-the-art for these two approaches to immutability analysis corroborates the findings of others [6], [7] that many tools only work small examples or cannot scale to the demands of realistic software [6] [7]. Moreover, the existing tools are largely academic prototypes that have not been rigorously tested for accuracy or have not been maintained and are unable to analyze programs written in later versions of Java. In particular, like many static analyses, a fully-precise immutability is intractable in general (e.g it may require a fully-precise pointer or alias analysis [10]), which forces tool implementers to make design choices or choose abstractions that necessarily affect the accuracy of their results.

*This material is based on research sponsored by DARPA under agreement numbers FA8750-15-2-0080 and FA8750-12-2-0126. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.55

The status of the current tooling prompts the need for (a) a robust implementation of the immutability analysis approaches that supports Java 8 and compiled Java bytecode, as well as (b) a benchmark that rigorously tests each approach for accuracy in the presence of various program features and analysis challenges.

C. Contributions

1) We build a robust immutability analysis toolbox that incorporates the transfer of two academic approaches for immutability analysis into an industry grade open source tool (the *immutability-toolbox*). The immutability-toolbox is built on top of a proven industry grade program analysis platform that lends itself to be composed with other software testing and verification tasks. The immutability-toolbox supports direct analysis of Java 8 source programs and compiled Java bytecode. Both implementations are capable of partial program analysis (analyzing libraries).
2) We develop a reusable, extensible benchmark for assessing accuracy boundaries of immutability analysis tools. The benchmark includes individual test cases that test specific program analysis challenges such as aliasing patterns, polymorphism, as well the minimal set of basic assignments that all tools must handle to correctly infer immutability. The novelty of the benchmark is that each testcase is: independently analyzable, human comprehensible, and contains an executable proof of correctness.
3) We use our benchmark to perform an in depth evaluation of the accuracy boundaries for each approach, and a set of real world applications to evaluate their scalability. In addition, we present the first detailed comparison of results of a points-to based immutability analysis and a type based immutability analysis.

II. Academic Transfer Challenges

In this section we present an overview of the existing tools for immutability analysis, and then motivate the need for an industry-grade experimentation toolbox for immutability analysis. Given multiple state-of-the-art approaches to immutability analysis, the toolbox is intended to enable industry practitioners to assess the relative trade-offs between these approaches applied to their domain specific analysis environment. The toolbox implements two prominent approaches, which we later use to critically evaluate the strengths and weaknesses of each approach in terms of accuracy, scalability and practical usability of the results. We describe the technical challenges we faced in transferring these approaches into an industry-grade open source *immutability toolbox* that can handle real world applications.

A. Overview of Existing Approaches and Tools

Existing approaches to immutability analysis can be classified into two categories: (1) those that that first perform a pointer analysis use the points-to results to infer immutability of objects, and (2) those that utilize a special formalism such as a type-system to define rules to infer immutability types of objects. ReImInfer (and its ancestor Javarifier) uses a declarative type-system based approach and applies predefined type inference rules to infer immutability types of references. In contrast, JPPA first performs a points-to analysis to derive the immutability of references using the computed points-to sets. A summary of the existing immutability analysis tools based on their support for analyzing Java source and Java bytecode, partial program analysis, and scalability to real world applications are presented in Table I.

TABLE I: Immutability Tool Tradoffs

Tool	Input	Supports Library Analysis	Notes
ReImInfer	Source	Yes	Only supports Java 6
JPPA	Source	No	Research prototype

Limitations of existing tools: JPPA is a research prototype implemented as compiler plugin that is no longer maintained. ReImInfer (and Javarifier) currently only supports programs compiled in Java 6 or earlier and does not report results for local references. Moreover, all tools expect Java source as input and do not natively support analysis directly on Java bytecode (at least not without some modifications). The lack of support for bytecode makes their applicability limited when source is not available. Given the above limitations, we decided to re-implement the type-inference based and points-to based approaches to immutability analysis for industry-grade use.

B. Analysis Platform for Immutability Toolbox

Immutability analysis is not used as an independent tool for software testing and V&V. Rather, immutability analysis is typically used to isolate parts of code without side-effects (e.g., by identifying pure methods), so that they can be independently tested. Hence, results of an industry-grade immutability analysis tool should be readily composable with other analyses. In addition, an industry-grade immutability analysis tool should support the analysis of partial programs as well as the analysis of bytecode when source is not available. Finally, the analysis should integrate seamlessly into the standard development environment and be easily inspected by a human. In summary, we expect any industry-grade implementation of immutability analysis to: provide results that are *readily composable* with other analysis, *accepts Java source and bytecode as input*, and supports easy *human inspection of results*.

Given the above considerations, we chose to implement our immutability analysis on top of the Atlas program analysis platform [11], which (a) provides native support for Java source and Java bytecode, (b) enables easy visual inspection of analysis results within the Eclipse IDE, and (c) supports on-demand composition of results from multiple analyses. Atlas is a mature and scalable program analysis platform for developing industry grade implementations of sophisticated algorithms for software verification, including safety-critical software verification tasks in automotive, avionics, and other industries requiring software V&V. In particular, Atlas has been used to build commercial products such as *Modelify* that recovers Simulink models from embedded C code [12]. Atlas has also been used by academic researchers to verify and discover bugs in the Linux Kernel [13], and as part of DARPA projects to detect novel and sophisticated malware in the Android ecosystem [14], and detect algorithmic complexity and side channel vulnerabilities in Java bytecode [15]. At its core, Atlas is a queryable, directed, graph database of attributed nodes and edges representing program artifacts and their relationships. Another specific advantage of using Atlas is its eXtensible Common Software Graph (XCSG) schema

that provides semantically precise definitions for all possible program artifacts.

We next describe the specific challenges we faced in transferring the type-inference based and points-to based academic approaches into an industry-grade *immutability-toolbox*. The *immutability-toolbox* which contains an type inference based implementation and points-to based implementation for immutability analysis is available online as open source project at https://ensoftcorp.github.io/immutability-toolbox.

C. Type Inference Based Approach

Approach Overview: The approach implemented by ReImInfer leverages a small auxiliary type system, a set of inference rules, and a type checker to infer the immutability of each object referenced in a program. The type system consists of three types with a hierarchy of `READONLY :> POLYREAD :> MUTABLE`, with `READONLY` as the most generic (super) type and `MUTABLE` as the most specific (sub) type. A type can only be assigned to the same type or a supertype. The `POLYREAD` type acts as a halfway point between the `READONLY` and `MUTABLE` types to account for mutations made in one calling context but not another. For instance a mutation to a field in one method does not imply that the field is mutated in every method that accesses the field. By default references are assigned a set of all three types, except for instance variables which are initialized with the set `READONLY` and `POLYREAD`. The inference rules are iteratively applied to each assignment statement in the program using a type checker to remove unsatisfiable types until a fixed point is reached. For instance in a basic assignment $x = y$, the `TASSIGN` inference rule asserts the constraint that the types on y should be less than or equal to the types on x. After fixed point is reached, the maximal type is selected from each set as the final type of the reference. This approach supports whole and partial program analysis with $O(3 * n)$ worst case number of iterations, where n is the number of assignments [6].

Transfer Challenges: First, the type inference based approach is declarative, and proposes a fixed point algorithm. The type inference rules describe "what" the types should be assigned, but does not provide details on "how" the algorithm could be implemented. Second, the authors present a core calculus and omits features not strictly necessary from the formalism for describing the inference rules. A complete detailing of the inference rules is not published, but a reference implementation supporting Java 6 program constructs is provided. For example, the published rules do not describe how a callsite without an assignment (e.g. `x.foo()`) would be handled (in practice, a dummy assignment `READONLY` reference is created to hold the result of the callsite). Third, when we consulted the implementation of ReImInfer to recover the undocumented type-inference rules, we found that in special cases the tool deviated from the approach stated in the paper. For example, there are cases when a `MUTABLE` type is added at runtime to an instance field, which contradicts statements in the [6], [7] papers that instance fields cannot be `MUTABLE` and that types are only ever removed. We verified this by reproducing the results with the original tool release (version 0.1.2) and observing that the tool reported `MUTABLE` instance fields. In addition, we observed several untyped references as well as unsatisfiable constraints for the reported benchmark applications. While a later paper [8] by the same authors does present a type system where instance variables can only be `{READONLY,MUTABLE}`, version 0.1.2 of the tool produced the same results presented in the OOPSLA 2012 [6] paper. Since, there is a series of papers [6], [7], [8], [9] which make improvements to the formalisms, but all papers reference the same code repository, it is difficult to recover and consult the implementation corresponding to a particular paper.

Addressing Challenges: Our primary source of clarifications to the implementation came from reaching out to the authors who kindly provided additional details. While the reference implementation leveraged a type checking framework, we extracted the constraints implied by the inference rules and built a dedicated constraint solver. We then leveraged the Atlas XCSG schema to enumerate the context in which mutations can happen and map each such mutation context to the corresponding constraints. The above enumeration and mapping were implemented using the native query language for XCSG. This effort included making several optimizations to our constraint solver, database queries, and result caching strategies. In particular, since the type system only consists of three types, it was possible to implementation our constraint solver as a lookup table that stores precomputed constraint results. Finally, in order to assert the correctness of the implementation for individual mutation contexts, we decided to develop a dedicated set of unit test cases which is described in detail in Section III. The final implementation represented over 8 weeks of full-time engineering effort by two software developers[1].

D. Points-to Based Approach

Approach Overview: Given the results of a points-to analysis it is a straight-forward task to implement an immutability analysis. There are many approaches to implementing points-to analyses, but for this work we implement a standard 0-CFA Andersen-style [16] pointer analysis (a context-insensitive subset constraint based approach). This choice is motivated by the fact that adding context sensitivity significantly increases cost without necessarily significantly improving the precision [17]. Given the pointer analysis, an immutability analysis only requires a single iteration through each new allocation in the program. For each new allocation the analysis determines if any of the references to the new allocation were used to update fields. If the a mutation occurred then all references to the new allocation are marked mutable. Updates to array components are treated as mutations to the array itself.

Transfer Challenges: There are two main challenges with respect to implementing a points-to analysis with the ultimate goal of computing immutability. First is the scalability of the analysis. It is know that the worst case complexity of an Andersen-style pointer analysis is $O(n^3)$ (where n is the number of nodes in the assignment graph) [16]. Second is the ability of the analysis to deal with partial programs (such as libraries). Most analyses, including JPPA, deal with both of these challenges by restricting the analysis to a main method or a root set of program entry points. The analysis is therefore restricted to a subset of the program, but cannot be performed for partial programs.

[1]The immutability-toolbox was implemented by the first two authors of this paper

Addressing Challenges: We address the above challenges in our points-to based immutability analysis implementation as follows. First, to increase the accuracy of the pointer analysis while maintaining scalability in practice, the implementation supports flow-sensitivity, on-the-fly resolution of dynamic dispatches, and a conservative tracking of flows through array components, as well as applying propagating subset constraints only based on type compatibility. Aside from the typical optimizations in points-to analysis (e.g. collapsing strongly connected components or using BDDs to store subsets) the implementation caches results of heavily used graph database queries such as subtype relationships. In addition, for immutability analysis we disable the tracking of primitives because they cannot be mutated.

To address the challenge of partial program analysis, the *points-to toolbox*) allows the analysis to assume every new allocation in the given code is feasible. This approach allows us to build a points-to mapping between references and new allocations even for partial programs at the cost of an increased computational burden. However, even with the burden of assuming all new allocations are feasible the *points-to toolbox* analysis completes in under a second for applications under 30K LOC, although the approach is not feasible for very large programs such as the full JDK. If the partial program analysis requirement could be relaxed such that the analysis has access to the whole program, but only needs to compute results for a subset of the program then it would be worth mentioning that recent work [18] in computing conservative points-to sets for libraries may make this approach feasible even for very large programs.

III. Benchmarking Immutability Analysis

The quality of software analysis tools can be evaluated along two dimensions, namely accuracy and scalability. Scalability is evaluated by testing the analysis tools on large real world applications. Ideally, accuracy of a tool is evaluated using a suite of test programs for which the ground truth (correct analysis result for each test) can be computed, which can then be compared with the results of the tool being tested. However, in the case of immutability analysis, there is a lack of a benchmark for which the ground truth is known. To cope with this, existing tools such as ReImInfer evaluate their accuracy using real world applications such as the DeCapo benchmark [19], and then compare their results with the results produced by previous tools and by manually inspecting the differences.

However, using real world software for evaluating the accuracy has several limitations. First, real world software applications were not developed with the intent to systematically test the *accuracy boundaries* (classes of inputs for which the tool cannot be guaranteed to report accurate results) of the tools and may miss important corner cases that are needed to guarantee the correctness of a tool. To make matters worse, the ground truth of the analysis result for real world software is not known *a priori*. Nor is there a way to compute the ground truth for reasonable sized real world applications. Second, even if real software were to be used as a benchmark to evaluate a tool, its inherent complexity makes it difficult to ascertain why the analysis tool failed, to resolve discrepancies in the results reported by different analysis tools, and to succinctly communicate the accuracy challenge that a tool can or cannot handle. Finally, since the accuracy can be independently tested exhaustively, it is best to design simple reproducible test cases with demonstrable ground truths that systematically test the accuracy boundaries of analysis tools.

A. Desirable Properties of Benchmark Testcases

In this section, we describe our approach to creating a benchmark that brings out the accuracy boundaries of the tools and embeds the ground truth within the test programs to evaluate the accuracy of the immutability analysis approaches. Specifically, we have designed our benchmarks for immutability analysis to contain test cases with the following desirable properties.

- **Independently Analyzable** Each test case should be individually testable, with all its dependencies encapsulated.
- **Human-comprehensible** The test cases should be simple enough that developers and users can understand the accuracy boundaries of a tool by browsing the tests cases in which the tool failed to produce correct analysis results.
- **Annotated** The test cases should be annotated to facilitate programmatic querying of the answer key. This makes it easy to write a test harness and evaluate whether a static analysis tool answered correctly for a given test case.
- **Executable** The test cases should also be executable, with its output serving as a proof of correctness of the test case's annotated answer key.
- **Reproducible** The test case should be deterministic and bundled with an environment that easily reproduces the results for a given implementation.
- **Individually Citable** To promote sharing of results, collaboration between researchers, and to support reproducibility, each test case should be individually referable. The DOI numbers for the benchmark (and its future versions) will be available at https://kcsl.github.io/immutability-benchmark. To cite a testcase, authors can specify the benchmark version by citing its DOI, along with the testcase category and number.

We next describe a benchmark containing the test cases with the above properties that we created for evaluating immutability analysis.

B. Basic Assignment Test Cases

We discuss a novel method to generate test cases for covering basic assignments in Java that any immutability analysis tool should be able to successfully analyze. These test cases do not require the tool to resolve aliases in order to pass. The novelty of our method of generating the basic assignment test cases that have the above desirable properties is twofold: (a) *systematic:* the test cases are generated by starting from the assignments allowed by the language model, and (b) *exhaustive:* every possible assignment within the Java language is tested. Since an object can be mutated only through field assignments, and aliasing is not considered, any immutability analysis tool should be expected to pass these test cases.

To ensure systematic, exhaustive coverage of basic assignment cases, it is important to note that all mutations occur as a result of an update (assignment after initialization) to a field. Let us first consider updates to class variables, instance variables, and array components of fields. Array fields

Listing 1: Basic Assignment Testcase

```
1  public class AGT117_This_Parameter {
2   @READONLY
3   public Test test = new Test();
4   public static void main(String[] args) {
5    new AGT117_This_Parameter().foo();
6   }
7   public void foo(){
8    System.out.println(test);
9    test.bar(test);
10   System.out.println(test);
11  }
12 }
13
14 class Test {
15  public Object f = new Object();
16  public void bar(Object p){
17   p = this; // no mutation
18  }
19  @Override
20  public String toString() {
21   return "Test [f=" + f + "]";
22  }
23 }
```

must be given special attention because there is no language mechanism to enforce the immutability of array components and updates to array components mutate the object containing the array field. Fields can be categorized into three groups, class variables, instance variables of another object instance, and instance variables of "this" object instance. Finally, we consider assignments involving stack variables, i.e. parameters passed and returned values. The high level assignment patterns can be represented succinctly as an *assignment graph* as shown in Figure 1. Each node in the assignment graph corresponds to a set of program artifacts that can be on the left hand side or right hand side of an assignment[2]. Each edge in the assignment graph can be instantiated multiple times, one for each pair of program artifacts in the source and destination node to generate a valid Java assignment. For instance, the downward edge to the right side of the assignment graph generates assignments from a final object, an `Enum`, a 'this' reference, a primitive (byte, char, short, int, long, float, double, boolean), 'null' or a String literal to a parameter. We enumerate all possible assignment pairs from the assignment graph to produce a total of 250 test cases.

Listing 1 shows the basic assignment test case corresponding to the assignment of a 'this' reference to a parameter (line 17). Clearly, it is independently analyzable because there are no third party dependencies, and a human readable can quickly comprehend the specific language feature being tested. The test also contains a main method that if executed prints the state of the `test` object before and after the assignment on line 17 is made in the `bar(Object p)` method invocation. Executing the test case produces reproducible output that indicates indicate the `test` field was not mutated. In this test, the assignment to a parameter does not mutate the parameter and so the `test` field is annotated with `READONLY`, which serves as an answer key that can be readily verified. When possible, we have designed the test cases to use only the necessary language features and program artifacts required to create each test case. In the above example, removal of any reference or method

[2]In the assignment graph, we intentionally do not consider inherited fields of an object, i.e., references through the "super" keyword to avoid introducing polymorphism challenges.

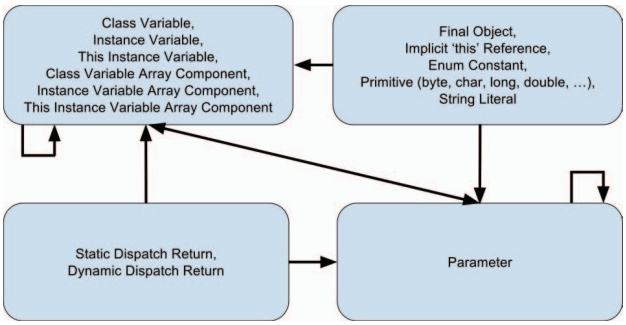

Fig. 1: Assignment Graph

would arguably compromise the desired properties of the test case such as executability or human comprehensibility.

C. Complex Program Analysis Challenge Test Cases

In addition to the basic assignment test cases, we created a set of manually curated test cases that involve complex program analysis challenges. These test cases test the accuracy of immutability analysis in the presence of aliasing, polymorphism and inheritance, open world assumptions, as well as issues that may arise from conservative assumptions to deal with array components and dynamic dispatch resolution. We next describe how we generated test cases that involve aliasing challenges. The other test cases involving polymorphism and inheritance, array index sensitivity and path sensitivity are not discussed here and can be found in the benchmark repository.

In the basic assignment test cases we defined a reference `test` that was explicitly mutated in the code and the analysis question was to correctly detect that `test` is mutated. In contrast, in the aliasing test cases, we create an alias to a reference `test`, and mutate the alias. The analysis question is to detect that `test` is indirectly mutated via the alias. In this category, there are four distinct aliasing patterns we test for, which can be explained using the concept of an aliasing chain. An aliasing chain is a sequence of explicit assignments of one reference to another, e.g., the statements `b = a; c = b;` creates the aliasing chain `c` $\rightarrow$ `b` $\rightarrow$ `a`. The four patterns of tests we perform are: (a) test case contains the aliasing chain `a` $\rightarrow$ `test`; (b) test case contains the aliasing chain `test` $\rightarrow$ `a`; (c) test case contains the aliasing chain `a` $\rightarrow$ `b` $\rightarrow$ `test`; and (d) test case contains the aliasing chain `test` $\rightarrow$ `b` $\rightarrow$ `a`. In all the above cases, the test case contains an explicit mutation to `a`, and analysis question asks whether `test` is mutable. The first two patterns (a) and (b) test whether mutations to an alias are propagated in either direction of the aliasing chain, while (c) and (d) test whether the analysis is also robust enough to detect mutations to aliases are propagated over levels of the aliasing chain. Listing 2 shows a test case corresponding to (b). Further these four basic aliasing patterns can be replicated by swapping the reference `a` with all the typable references such as a field, parameter, etc. Our test cases for aliasing are not exhaustive, and future work exists to expand these tests in a systematic way, as we have done for the basic assignment test cases.

Note that tests that address potential accuracy boundaries of points-to analysis implementations would also be applicable to testing immutability analysis implementations. In a feasibility

Listing 2: Aliasing Testcase

```
public class AT_002 {
 @MUTABLE
 public Test test;
 public static void main(String[] args) {
  new AT_002().foo();
 }
 public void foo() {
  // aliasing pattern: test -> a
  Test a = new Test();
  test = a;
  System.out.println(test);
  a.f = new Object(); // test is mutated
  System.out.println(test);
 }
}

class Test {
 public Object f = new Object();
 @Override
 public String toString() {
  return "Test [f=" + f.toString() + "]";
 }
}
```

study we performed to create systematic tests for points-to analysis, available at https://github.com/kcsl/JPATS, we discovered and reported an accepted bug in the Soot program analysis framework [20]. In addition to the aliasing patterns, it is also possible to generate test cases for various kinds of dynamic dispatch and inheritance patterns, which would specifically test an analysis for correctness on propagating mutations through callsites. Our benchmark includes a few test cases in this category. This also opens another area of future work to create systematic test cases that systematically model inheritance and dynamic dispatch patterns.

IV. Experiments

To assess the industry-readiness of research prototypes developed in academia, it is necessary to understand their relative tradeoffs with respect to accuracy and scalability. These tradeoffs are non-trivial because research publications typically do not include a precise definition of the accuracy boundaries of the tool (i.e., cases where the tool is not guaranteed to be accurate) and the practical limits of its scalability. We perform experiments to help industry practitioners identify these tradeoffs. Specifically, we answer the following research questions.

- RQ1 How accurate are the approaches based on their results of analyzing the benchmark testcases? In the case of failures, what insights can we derive about the relative strengths and/or weaknesses of the approaches?
- RQ2 How do the results compare in their reported mutations on real world applications? Does one approach consistently detect more mutations across all the applications tested?
- RQ3 How does the performance of each approach scale on the real world applications tested?

A. RQ1: Accuracy with respect to benchmark

We discuss the results of ReImInfer and our points-to implementation on the basic assignment test cases and those involving complex program analysis challenges.

Results of basic assignment test cases: Recall that we generated a total of 250 basic assignment test cases, which cover all possible Java language assignment patterns. These tests ensure that an immutability analysis can correctly detect mutations on direct references to an object instance. As expected, both approaches (type-system based ReImInfer implementation and our points-to based immutability-toolbox implementation) correctly analyzed all of the 250 test cases.

Results of test cases involving complex program analysis challenges: Test cases in this category required the analysis to correctly detect mutations when the reference being tested is involved in various aliasing patterns, in the presence of polymorphism and inheritance, etc. (see Section III-C).

Aliasing: Our points-to based analysis passed test cases corresponding to all the four aliasing patterns. ReImInfer passed only two of the test cases, and failed on the two other aliasing patterns: `test` → `a` (as shown in Listing 2) and `test` → `b` → `a` (in both cases, ReImInfer could not detect that `test` is mutated when `a` is mutated). From this result, we derived the insight that one of the fundamental constraints for the assignment `x=y`, the constraint `x :> y` does not propagate mutations on `y` to `x`. With the understanding that longer aliasing chains offer more opportunities for such aliasing patterns to occur, we leveraged our points-to based implementation for calculating the frequency of aliasing chains of different lengths in a real world application TinySQL. Figure 2 plots a histogram of this statistic, showing that such cases are quite prevalent in practice. Specifically, for TinySQL we estimated that cases of aliasing chains of length 2 or more (47.12% of all aliasing chains detected by the points-to analysis) present opportunities for incorrect detection of mutations by ReImInfer.

Polymorphism and Inheritance: Another test case that ReImInfer fails that the point-to implementation in the immutability-toolbox passes, involves method overriding and is shown in Listing 3. This test creates two subtypes `A` and `B` that both override the base method `foo` in `Test`. `Test.foo` and `A.foo` do not mutate the inherited field `f`, while `B.foo` does mutate the field `f`. While the points-to based implementation resolves dynamic dispatches on the fly, in ReImInfer a mutation in an overridden method always observed as a mutation in the overridden method and the base method (regardless of the base method's behavior).

Other challenges where both approaches failed: Finally, there are a set of testcases in the benchmark that both approaches fail on, which involve challenges such tracking mutations to a specific array index or detecting mutations along conditional paths (a mutation in one path but not in another is always conservatively detected as a mutation).

B. RQ2: Comparison of results on real world applications

Experimental setup: We ran experiments on nine real world applications, which is a subset of the original set of applications on which ReImInfer reported results [6]. We omitted the `java.lang` and `java.util` JDK packages because the authors did not specify the build version used in their experiments. We omit the SPECjbb benchmark because it is a non-free proprietary benchmark. We also omit the JOlden benchmark because it primarily consisted of small toy examples, which are not representative of typical real world applications. Finally, we restrict our comparison to results reported by our points-to based implementation and the original results reported by ReImInfer (rather than our re-implementation) because it

Listing 3: Overrides Testcase

```
public class ST_005 {
 public static Test test = new B();
 public static void main(String[] args){
  System.out.println(test);
  test.foo(); // B.foo mutates test
  System.out.println(test);
 }
}

class Test {
 protected Object f = new Object();
 public void foo(){} // no mutations
 @Override
 public String toString() {
  return "Test [f=" + f + "]";
 }
}

class A extends Test {
 @Override
 public void foo(){} // no mutations
}

class B extends Test {
 @Override
 public void foo(){
  this.f = new Object(); // mutates 'this'
 }
}
```

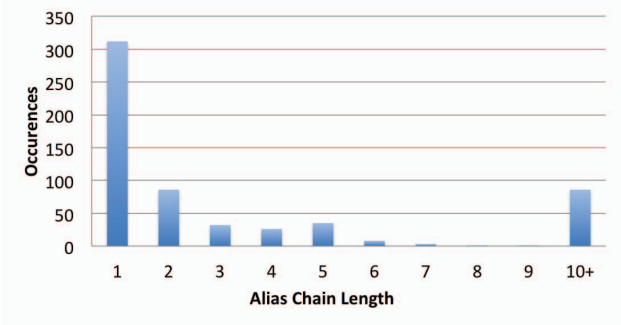

Fig. 2: Alias Chain Lengths in TinySQL

differs in how it applies types to fields as we discuss in Section II-C.

Comparison of reported mutations: Table II shows a comparison of the analysis results based on the fraction of references reported as `READONLY` vs. all other potential mutations (`POLYREAD` and `MUTABLE`) respectively by both approaches. Interestingly, there was no single approach that consistently detected more potential mutations than the other across all the applications. Specifically, the ReImInfer reported more potential mutations for htmlparser, xalan, commons-pool, jdbm, jdbf, jtds, whereas our points-to based immutability-toolbox reported more potential mutations for javad, ejc, and TinySQL. This mixed trend can be explained by our previous observation that ReImInfer fails to correctly detect mutations in cases involving certain aliasing patterns and method overriding patterns (as discussed in Section IV-A). In addition, this discrepancy may also be attributed to ReImInfer's conservative assumptions that were not made by our points-to based analysis (ReImInfer assumes that library methods outside of a default set of pure methods always mutate objects).

To derive additional insights into where differences were reported in the tools we investigated the detailed break down of mutations on fields, method parameters, method identities, and method returns for one application, namely TinySQL (shown in Table III). Notice that the number of identifiable references do not total to the same counts as reported by ReImInfer. This is because, as the authors did when they ported the applications to Java 6, we also added empty method implementations to satisfy new interface contracts for Java 8 to analyze the applications with the immutability-toolbox. As a result we have more parameters (including the implicit 'this' reference) and returns than were originally reported by ReImInfer. Finally, we observed a discrepancy between the results of the approaches in the number of `POLYREAD` and `MUTABLE` fields reported across all applications. This is likely because the implementation of ReImInfer allows `MUTABLE` types on instance fields (contrary to the authors' proposal in the paper [6]), as we noted in Section II-C. In our points-to based analysis, we follow the proposed type system of ReImInfer, which does not allow `MUTABLE` types on instance fields.

C. RQ3: Scalability in practice

Given that no single approach dominates another in terms of accuracy, scalability may become the deciding criteria for the choice of an immutability analysis approach in an industrial deployment. Table II shows the time in seconds taken by ReImInfer and the cumulative time taken be the immutability-toolbox's points-to based immutability analysis and client immutability analysis implementations. For the analysis of small applications such as TinySQL, htmlparser, javad, commons-pool, jdbm, jdbf, and jtds the points-to based analysis performed significantly better. However, as the size of the application grew past 38k lines of code, as was the case for ejc and xalan, the type inference based approach performed significantly better. This result indicates that the type inference based approach is a clear winner for the analysis of very large applications since a points-to based analysis understandably becomes prohibitively expensive. It is important to note however, that this comparison only applies to immutability analysis using an underlying 0-CFA Andersen style points-to analysis. The accuracy and scalability trade-offs between several different points-to analyses have been well researched [17], [21]. One direction for future work is to study how the choice of points-to analysis impacts a points-to based immutability analysis in terms of accuracy and scalability.

V. Conclusion

In this paper we re-implemented two prominent approaches to immutability analysis, namely a points-to based and a type-inference based approach. We presented the challenges we faced in the academic transfer and how we addressed them. In particular, our re-implementation of the points-to based immutability approach involved modifying a well known static analysis algorithm (0-CFA Andersen style pointer analysis), our reimplementation of ReImInfer was more challenging because it required running several experiments and directly conversing with the original authors for clarifications to unstated assumptions made in their papers. Our re-implementations are built on top of a robust and well maintained program analysis platform that provides native support for handling the necessary language features for Java 8 source and Java bytecode. We developed a benchmark to validate the basic

TABLE II: Immutability Analysis Results on Real World Applications

Application	Lines of Code	ReImInfer READONLY	ReImInfer Time	Points-to READONLY	Points-to Time	Immutability Analysis Time	Total Time
TinySQL	31980	62%	15.1s	60.2%	1.7s	3.1s	4.8s
htmlparser	62627	56%	16.9s	59.4%	1.7s	3.5s	5.2s
ejc	110822	40%	66.2s	20.9%	371.6s	146s	517.6s
xalan	348229	61%	81s	73.9%	127.5s	504.7s	632.2s
javad	4207	69%	3.2s	34.2%	0.3s	0.6s	0.9s
commons-pool	4755	44%	3.8s	68.2%	0.2s	0.5s	0.7s
jdbm	11610	40%	5.9s	52.0%	0.3s	1s	1.3s
jdbf	15961	66%	9.6s	78.9%	1.0s	1.5s	2.5s
jtds	38064	56%	17.2s	59.5%	4.7s	6.7s	11.4s

TABLE III: ReImInfer vs. Points-to based Immutability Analysis on TinySQL *ReImInfer results are shown on top in gray while our points-to based analysis results are shown below

TinySQL	READONLY	POLYREAD	MUTABLE	Total
Field	124	1	132	257
	138	109	10	257
Parameter	703	2	264	969
	772	0	221	993
Identity	970	62	529	1561
	937	0	743	1680
Return	404	198	0	602
	448	162	0	610

correctness of the approaches implemented, and to test the accuracy boundaries inherent to each approach.

We learned that the amount of engineering work that is involved in transferring an academic approach is surprisingly large. Specifically, in the case of transferring academic work that utilizes special formalisms for describing an analysis approach, seemingly innocuous assumptions not explicitly stated in the paper, which may be obvious to the authors, may have profound impacts on the accuracy of the approach. In addition to posing challenges to implementers transferring the approach, it is a problem if the consumers of an approach are not aware of the practical accuracy vs. scalability tradeoffs made by the approach.

Our experiments found that there was no clear winner between the approaches. The points-to based analysis was a winner in terms of accuracy, but the type based approach scaled better to large applications at the cost of potential false negatives. Our exercise of creating a benchmark brought out the accuracy boundaries inherent in each approach and provided precise examples of analysis challenges not handled by an approach. The approach agnostic benchmark created in this work can be used by industry practitioners to objectively assess current and future immutability analysis approaches.

REFERENCES

[1] A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *Proc. of the International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2005, pp. 199–215.

[2] DARPA, "Space / Time Analysis for Cybersecurity (STAC)," Information Innovation Office, Arlington, VA, Tech. Rep., 2014.

[3] M. Finifter, A. Mettler, N. Sastry, and D. Wagner, "Verifiable functional purity in java," in *Proc. of the Conference on Computer and Communications Security*. ACM, 2008, pp. 161–174.

[4] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen, "Boa: A language and infrastructure for analyzing ultra-large-scale software repositories," in *Proc. of the International Conference on Software Engineering*, 2013, pp. 422–431.

[5] B. Holland, "Immutable objects in github projects," http://boa.cs.iastate.edu/boa/?q=boa/job/public/43284, July 2016.

[6] W. Huang, A. Milanova, W. Dietl, and M. D. Ernst, "Reim & reiminfer: Checking and inference of reference immutability and method purity," in *ACM SIGPLAN Notices*, vol. 47, no. 10. ACM, 2012, pp. 879–896.

[7] W. Huang and A. Milanova, "Reiminfer: method purity inference for java," in *Proc. of the International Symposium on the Foundations of Software Engineering*, 2012, p. 38.

[8] A. Milanova and W. Huang, "Dataflow and type-based formulations for reference immutability," in *International Workshop on Foundations of Object-Oriented Languages*, 2012, p. 89.

[9] A. Milanova and Y. Dong, "Inference and checking of object immutability," in *Proc. of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*. ACM, 2016, pp. 6:1–6:12.

[10] G. Ramalingam, "The undecidability of aliasing," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 5, pp. 1467–1471, 1994.

[11] T. Deering, S. Kothari, J. Sauceda, and J. Mathews, "Atlas: a new way to explore software, build analysis tools," in *Proc. of the International Conference on Software Engineering*. ACM, 2014, pp. 588–591.

[12] E. Woestman and J. Sauceda, "Modelify: Semi-automatic conversion of control systems c code to simulink models," in *SAE Technical Paper*. SAE International, 04 2016. [Online]. Available: http://dx.doi.org/10.4271/2016-01-0020

[13] S. Kothari, A. Tamrawi, J. Sauceda, and J. Mathews, "Let's verify linux: Accelerated learning of analytical reasoning through automation and collaboration," in *Proc. of the International Conference on Software Engineering Companion*. ACM, 2016, pp. 394–403.

[14] B. Holland, T. Deering, S. Kothari, J. Mathews, and N. Ranade, "Security toolbox for detecting novel and sophisticated android malware," in *Proc. of the International Conference on Software Engineering - Volume 2*. IEEE Press, 2015, pp. 733–736.

[15] B. Holland, G. R. Santhanam, P. Awadhutkar, and S. Kothari, "Statically-informed dynamic analysis tools to detect algorithmic complexity vulnerabilities," in *16th IEEE International Conference on Source Code Analysis and Manipulation*, 2016.

[16] L. O. Andersen, "Program analysis and specialization for the c programming language," Ph.D. dissertation, University of Cophenhagen, 1994.

[17] O. Lhoták and L. Hendren, "Context-sensitive points-to analysis: is it worth it?" in *International Conference on Compiler Construction*. Springer, 2006, pp. 47–64.

[18] K. Ali and O. Lhoták, "Averroes: Whole-program analysis without the whole program," in *ECOOP*, 2013, pp. 378–400.

[19] S. M. Blackburn, R. Garner, C. Hoffman, A. M. Khan, K. S. McKinley, R. Bentzur, A. Diwan, D. Feinberg, D. Frampton, S. Z. Guyer, M. Hirzel, A. Hosking, M. Jump, H. Lee, J. E. B. Moss, A. Phansalkar, D. Stefanović, T. VanDrunen, D. von Dincklage, and B. Wiedermann, "The DaCapo benchmarks: Java benchmarking development and analysis," in *Proc. of the Conference on Object-Oriented Programing, Systems, Languages, and Applications*. ACM Press, 2006, pp. 169–190.

[20] "java.lang.runtimeexception when processing class files with array accesses on null arrays #527." [Online]. Available: https://github.com/Sable/soot/issues/527

[21] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proc. of Object Oriented Programming Systems Languages and Applications*. ACM, 2009, pp. 243–262.