Verifying Dereference Safety via Expanding-Scope Analysis

A. Loginov* E. Yahav S. Chandra S. Fink
IBM T. J. Watson Research Center

N. Rinetzky
Tel Aviv University

M. G. Nanda
IBM IRL

ABSTRACT

This paper addresses the challenging problem of verifying the safety of pointer dereferences in real Java programs. We provide an automatic approach to this problem based on a sound interprocedural analysis. We present a staged *expanding-scope* algorithm for interprocedural abstract interpretation, which invokes sound analysis with partial programs of increasing scope. This algorithm achieves many benefits typical of whole-program interprocedural analysis, but scales to large programs by limiting analysis to small program fragments. To address cases where the static analysis of program fragments fails to prove safety, the analysis also suggests possible annotations which, if a user accepts, ensure the desired properties. Experimental evaluation on a number of Java programs shows that we are able to verify 90% of all dereferences *soundly* and automatically, and further reduce the number of remaining dereferences using non-nullness annotations.

Categories and Subject Descriptors: D.2.4 [Program Verification]

General Terms: Algorithms, Reliability, Verification

Keywords: Static Analysis, Abstract Interpretation

1. INTRODUCTION

This paper addresses the challenging problem of verifying the safety of pointer dereferences in real Java programs. Our analysis operates on program *fragments*, and gradually expands the analysis scope in which a fragment is considered when additional context information is required.

We focus on sound analysis. In this setting, if the analysis issues no warnings, then all dereferences in the program are guaranteed to be safe. However, if the analysis issues a warning, it may be a *false positive*. The challenge in designing a sound analysis is therefore not only to produce one that scales in terms of program size, but does so while maintaining a low number of false positives.

Our goal is not to produce a tool specifically for bug finding; rather, our goal is to help a developer make her application more robust, while engaging her attention as few times as possible. Therefore, our metric of success is in getting the maximal number of dereferences that can be automatically established as safe with a minimal number of user annotations.

Verifying Dereference Safety. Null-pointer dereferences represent a non-negligible percentage of defects in Java applications. Furthermore, a null dereference is often a symptom of a higher-level problem; warning the programmer about a potential null dereference may help in exposing the more subtle problem.

We present the tool SALSA (**S**calable **A**nalysis via **L**azy **S**cope exp**A**nsion). Unlike most existing bug-finding tools for detecting null dereferences our analysis is *sound*. We are unaware of any reported *sound* approach for detecting null dereferences in real Java programs that produces a reasonable percentage of false positives.

Expanding-Scope Analysis. We present a *staged* analysis that adapts the *cost* of the analysis to the difficulty of the verification task. Our analysis breaks the verification problem into multiple subproblems and adapts the analysis of each subproblem along two dimensions: the *precision* dimension and the *analysis-scope* dimension. Our analysis adapts the *precision* (and thus the expected cost) of the abstract interpretation [4] to the difficulty of verifying the subproblem. In this aspect, it is similar to the staging in [11].

The novelty of our approach lies in its ability to adapt the *scope* of the analyzed program fragment to the difficulty of the verification task. Unlike existing staged techniques, which analyze whole programs (e.g., [11]), our analysis operates on program *fragments*. The basic idea, inspired by Rountev et. al. [18], is to break the program into fragments and analyze each fragment separately, making conservative assumptions about the parts of the program that lie outside the fragment. However, if the property cannot be verified under the conservative context assumptions, our approach provides for gradually *expanding the scope* of the analyzed fragment.

The premise of this work is that a large percentage of the potential points of failure in a program can be verified by (i) using a scalable imprecise analysis that conservatively approximates context information, and (ii) employing more precise analyses that consider a limited scope, which may be expanded as needed.

SALSA is based on the principle of expanding scopes; it applies this principle to the problem of dereference safety, which is particularly challenging due to its dependence on aliasing information.

Experimental Results. Our experimental results show that our analysis is effective in proving the safety of dereferences. Around 90% of the dereferences, averaged over a suite of 21 benchmarks, were proved safe *fully* automatically. The benchmarks, which are mostly open-source programs downloaded from sourceforge, ranged in size from 3K bytecodes to over 460K bytecodes (see Tab. 4). Given that each of these benchmarks had several thousand dereferences on average, we find the results encouraging.

*Author's current affiliation: GrammaTech, Inc.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISSTA'08, July 20–24, 2008, Seattle, Washington, USA.
Copyright 2008 ACM 978-1-59593-904-3/08/07 ...$5.00.

Related Approaches. To evaluate the results of our analysis, we compared it with the results of state-of-the-art best-effort bug-finding tools. We have implemented two such tools: Loco, a local belief-based bug finder, and Xylem, a local symbolic path simulator. We have also compared the results of our analysis to ones we obtained by running FindBugs [14] on our benchmark suite.

Our preliminary experimental evaluation suggests that SALSA can complement bug-finding tools by exposing problems that were missed by these tools, as well as by showing that some of their reports do not correspond to bugs when considered in a larger scope.

SALSA can benefit from the reports of bug-finding tools by using them as a basis for ranking.

Leveraging Annotations. For some safe dereferences, SALSA may not be precise enough to verify their safety fully automatically. In such cases, we allow the user to specify additional knowledge via lightweight annotations (described in §7). For example, if the user annotates a field as non-nullable, our analysis can use this knowledge to verify additional dereferences as safe.

SALSA uses simple heuristics to suggest effective candidate annotations to the user. The task of examining suggested annotations is far less onerous than the task of writing annotations from scratch. In our experiments, we show that using these suggested annotations can significantly increase the number of verified dereferences. More specifically, we annotated 8 out of our 21 benchmarks, and we show that adding a total number of 173 annotations reduced the number of remaining unverified dereferences by an average of 30% for these benchmarks.

Main Results. The contributions of this paper can be summarized as follows:

- We present a novel approach for a *staged analysis* based on the notion of expanding analysis scopes and apply it to the problem of dereference safety.
- We implemented our approach and evaluated it on a number of benchmarks. We show that our approach can automatically verify the safety of around 90% of dereferences.
- We show how simple user annotations enable the analysis to verify significantly more dereferences as safe.

```
1  public class EmpRec {
2    final String name;
3    Integer salary;

5    EmpRec(String empStr) {
6      int sepIdx = empStr⊡indexOf(':');
7      this.name = getName(empStr, sepIdx);
8      this.salary = getSalary(empStr, sepIdx);
9    }

11   String getName(String empStr, int sepIdx) {
12     return empStr⊡substring(0, sepIdx);
13   }

15   Integer getSalary(String empStr, int sepIdx) {
16     int len = empStr⊡length();
17     String salaryStr = empStr⊡substring(sepIdx+1, len);

19     if (salaryStr⊡matches("\\p{Digit}+"))
20       return Integer.valueOf(salaryStr);
21     else   // Unpaid
22       return new Integer(0);
23   }

25   public String toString() {
26     StringBuffer result = new StringBuffer();
27     result⊡append(this.name⊡toUpperCase());

29     result⊡append("\tsalary:␣");
30     result⊡append(this.salary⊡toString());
31     Currency currency = getCurrency();
32     result⊡append(currency⊡getSymbol());

34     return result⊡toString();
35   }

37   Currency getCurrency() {
38     Locale locale = Locale.getDefault();
39     Currency currency = Currency.getInstance(locale);
40     return (currency != null ?
41             currency : Currency.getInstance("USD"));
42   }

44   public static void test() {
45     EmpRec emp = new EmpRec("Jane␣Smith:50000");
46     System.out⊡println(emp⊡toString());
47   }
48 }
```

Figure 1: The running example.

2. OVERVIEW

In this section, we introduce our running example and provide an overview of our approach at a semi-technical level. The technical details are fleshed out in later sections.

2.1 Running Example

Fig. 1 shows a Java class for manipulating an employee record. Method `test` creates an employee record for "Jane Smith", sets her salary to $50,000, and prints the record.

We emphasize points in the example program where dereferences have to be shown as safe by marking the dereference with a small square. Our analysis considers every such dereference as a separate verification problem.

The example program is safe. Our analysis is able to *verify* each of these subproblems automatically. It shows that every pointer dereference statement in the running example is safe.

Different reasons contribute to the successful verification of the various subproblems. For example, it can be established based on the following facts: (1) The result of a successful allocation is non-null (line 45), (2) A successful dereference ensures that subsequent dereferences of the same pointer will succeed, e.g., after successful dereference of `empStr` at line 16; (3) Specifications of library methods, e.g. `substring` (line 17) and `getInstance` (line 41) are guaranteed to return non-null objects; (4) Various path conditions (e.g. `x` **instanceof** `T`) guarantee non-nullness along the path.

An intraprocedural analysis can discover some of these facts, with a straightforward dataflow analysis similar to available expressions (§4). Such analysis can prove the following dereferences safe: (1) `emp` at line 46 due to prior allocation; (2) `empStr` at line 17 due to prior dereference; (3) all occurrences of `result` in `toString` due to prior allocation; (4) `salaryStr` at line 19 due to known non-null return value from preceding library call.

The remaining dereferences depend either on formal parameters, on values returned from called methods, or on temporal ordering of assignments and accesses to fields. Without information from callers and callees, sound intraprocedural analysis must make pessimistic assumptions. For example, dereferences of methods' parameters, such as the dereference of `empStr` in the constructor, must be considered possibly unsafe. This effect grows more pronounced with larger and more realistic programs.

When intraprocedural analysis fails, we turn to staged interprocedural analysis as illustrated in Fig. 2. The three columns of the figure show the callgraph of the program (we omit library methods for clarity). Fig. 2 (a) contains dashed boxes around every method that contains dereferences. The first stage, a 0-scope analysis is a (sound) intraprocedural analysis as just discussed.

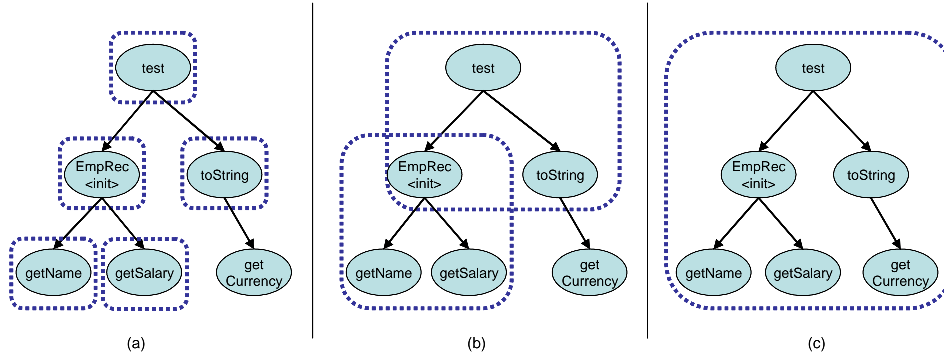

Figure 2: Expanding scopes for the running example: (a) 5 regions of scope depth 0, (b) 2 regions of scope depth 1, and (c) 1 region of scope depth 2

Expanding the scope of the analysis leads to the analysis of multiple methods together. Fig. 2 (b) shows the analysis scopes of the 1-scope analysis performed in the next stage. These scopes contain the analyzed method, one of its callers, and all the other methods that can be invoked directly by that caller. In our example, the analysis of `getName` and `getSalary` is performed in a scope containing `EmpRec`'s constructor and these two methods, and the analysis of the dereferences in `toString` and in `EmpRec`'s constructor is performed in a scope containing the `test` method and these two methods. The 1-scope analysis of `EmpRec`'s constructor can successfully propagate non-nullness information—a string constant being a non-null object—from the actual parameter in `test` to the formal parameter `empStr` of the constructor of `EmpRec`, and consequently show that the dereference on line 6 is safe.

Fig. 2 (c) shows the 2-scope analysis of `getName` and `getSalary`. This scope contain the grand-parent of the analyzed method in the call graph (i.e., method `test`) and all children and grandchildren of method `test`. This stage is able to show that the dereferences in methods `getName` (line 12) and `getSalary` (line 16) are safe by propagating the non-nullness of the `empStr` parameter from the `test` method. In a similar manner, the dereference `this.name` (line 27) are proven safe. The two remaining dereferences, `this.salary` (line 30), and `currency`(line 32) are proven safe by techniques described in §6.1 and §6.3, respectively.

In summary, all dereferences are proved safe for this example program. In this small program, a 2-scope analysis amounts to a whole-program analysis, but in general, we omit large portions of an application when reasoning about a certain dereference with a 2-scope analysis. Note also that the analysis performed with larger scopes deals with only those dereferences that could not be proved safe by preceding, less expensive scopes, and thus reduces the cost of the analysis significantly. §8.1 sketches one of a number SALSA's optimizations that are enabled by considering a small set of dereferences in larger scopes.

3. PRELIMINARY ANALYSIS

We assume the existence of a preliminary phase that constructs the program's call graph and obtains points-to information, together with aliasing information.

The program's call graph. The nodes of the program's call graph are the methods in the program. There is an edge between node m_1 and node m_2, if method m_1 may invoke method m_2. For example, Fig. 2 (a) shows the call graph of our running example, omitting nodes that represent library methods.

Access paths. Let *VarId* be the set of local variable identifiers, *SFieldId* be the set of static field identifiers, and *IFieldId* be the set of instance field identifiers of the program (all assumed to be of reference type). Let $\Gamma \in \mathit{IFieldId}^*$ be a (possibly empty) sequence of instance field identifiers. An *access path* $ap \in AP = (\mathit{VarId} \cup \mathit{SFieldId}) \times \Gamma$ represents the l-value of the reference-type expression that is the dereference of the first component by the second component.

Alias analysis. For an access path ap, may-alias information provides the set of access paths that may be aliased with ap (denoted by $\mathit{mayAlias}(ap) \subset AP$) at some point during program execution. For an access path ap at a given program point n, must-alias information provides the set of access paths that refer to the same object as ap every time program point n is reached. This set is denoted by $\mathit{mustAlias}(ap)$. For convenience in presentation, we assume that ap is included in $\mathit{mustAlias}(ap)$. For a method m, may-alias information provides the set of access paths that may change their value as a result of m's invocation. This set is denoted by $\mathit{mayKill}(m)$.

Our analysis is parametric in the preliminary phase. In our experiments we use a call-graph construction and may-alias analysis involving a flow-insensitive, context-insensitive subset-based Andersen-style may-points-to analysis, as described in [11]. For must-alias information, we use a simple demand-driven must-alias computation that is based on following def-use chains. Note that the analysis remains sound given any sound call-graph construction and alias analysis. In particular, we can utilize highly scalable class-hierarchy-based call-graph construction [5] and type-based alias analysis [6].

4. INTRAPROCEDURAL ANALYSIS

Our analysis attempts to infer for every pointer dereference statement whether the dereferenced access path is guaranteed to have a non-null value. In this section, we describe the abstract domain and the transformers that we use for the intraprocedural analysis.

4.1 Abstract Domain

Our abstract domain is a product of three domains: (i) the abstract domain used for the may-alias analysis, (ii) the abstract domain used for the must-alias analysis, and (iii) a set AP_{nn} of access paths that are guaranteed to have a non-null value. We guarantee that the abstract domain is finite by placing a (parameterized) limit on the size of the AP_{nn} sets and on the maximal length of the ac-

Statement	Resulting abstract value
v = `null`	$AP_{nn} \setminus \{v.\gamma \mid \gamma \in \Gamma\}$
$v = w$	$AP_{nn} \cup \{v.\gamma \mid w.\gamma \in AP_{nn}\}$
$v = w.f$	$AP_{nn} \cup \{v.\gamma \mid w.f.\gamma \in AP_{nn}\} \cup mustAlias(w)$
v = `new` T()	$AP_{nn} \cup \{v\}$
$v.f$ = `null`	$AP_{nn} \setminus \{e'.f.\gamma \mid e' \in mayAlias(v) \wedge \gamma \in \Gamma\}$ $\cup\, mustAlias(v)$
$v.f = w$	$AP_{nn} \cup \{e''.f.\gamma \mid w.\gamma \in AP_{nn} \wedge e'' \in mustAlias(v)\}$ $\cup\, mustAlias(v)$
$v.foo()$ $u = v[i]$ $v[i] = u$ $u = v.length$	$AP_{nn} \cup mustAlias(v)$

Table 1: Transfer functions for statements indicating how an incoming non-null access-path set AP_{nn} is transformed, where $u, v, w \in$ *VarId* range over variable names and $f \in IFieldId$ ranges over instance-field identifiers

Condition	Resulting abstract value	
	on true branch	on false branch
v == `null`	$v \in AP_{nn} ? \bot : AP_{nn}$	AP_{nn} $\cup\, mustAlias(v)$
v `instanceof T`	$AP_{nn} \cup mustAlias(v)$	AP_{nn}
$v == w$	AP_{nn} $\cup\, (mustAlias(w)$ *if* $v \in AP_{nn})$ $\cup\, (mustAlias(v)$ *if* $w \in AP_{nn})$	AP_{nn}

Table 2: Transfer functions for conditions indicating how an incoming non-null access-path set AP_{nn} is transformed, where $v, w \in$ *VarId* range over variables names and $T \in TypeIds$ ranges over type names

cess paths that they may contain.[1] We refer to the size of the access path set AP_{nn} as the *width* of AP_{nn} and to the maximal length of an access path $ap \in AP_{nn}$ as the *length* of AP_{nn}.

4.2 Transformers

Our analysis collects at every program point the third component of the abstract domain, i.e., the set AP_{nn} of non-null-valued access paths. The first two components are determined by the results of the preliminary analysis; they are not modified thereafter.

We assume, without loss of generality, that instruction sequences are normalized in the following way: only local variables are dereferenced, every statement contains at most one field dereference, and that all assignments of the form $v = w$ and $v.f = w$ are preceded by assignments $v = null$ and $v.f = null$, respectively.

Tab. 1 shows the transfer functions associated with statements. Most transfer functions are quite straightforward. The only subtle point is that after a statement that dereferences a variable v, the analysis can add v, as well as access paths in v's must-alias set, to the set of non-null access paths. This is because the dereference of v can be executed successfully only if v is non-null. Tab. 2 shows the transfer functions associated with conditions. We explain the more subtle points. If v is known to be non-null, the true branch of the condition v == `null` indicates an infeasible path. Also, if v is known to be non-null, the true branch of the condition $v == w$ indicates that w, as well as its must-alias access paths, are non-null.

We can ensure that a statement at a given program point cannot perform a null dereference by verifying that the access path that the statement dereferences is in the set of non-null access paths AP_{nn} collected at that point.

[1]The length of access path $\langle v, \langle f_1, \ldots, f_k\rangle\rangle$ is defined to be $k + 1$.

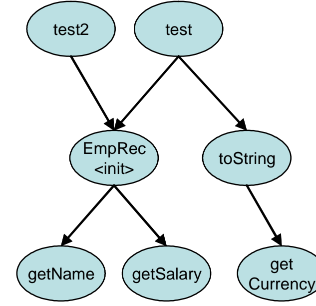

Figure 3: Call graph for an extension of the running example

5. EXPANDING-SCOPE ANALYSIS

The combinatorial nature of our abstract domain (that is based on sets of non-null access paths) makes a whole-program interprocedural analysis infeasible. We address the scalability problem by presenting the novel notion of the analysis of *limited program scopes*. A limited-program-scope analysis considers a small fragment of the program, namely a method that contains a dereference together with some of its neighbors in the call graph. (The effect of the rest of the program on the analysis is reflected conservatively.) The analysis attempts to verify that every dereference in the method is safe using the smallest possible scope. When the analysis cannot establish the safety of a dereference in a given scope, it extends the scope and performs the analysis again in the extended scope(s).

We found that the use of expanding scopes is key for the scalability of our analysis. The utilization of expanding scopes for scaling the analysis is one of the main contributions of this paper. We note that the notion of expanding-scope analysis is also relevant in other verification problems such as typestate verification [20].

This section describes the steps of expanding-scope interprocedural analysis for a set of dereferences that have not been shown to be safe (each paired with a calling context in which it should be analyzed next). These steps can be summarized as follows:

Compute analysis scopes Find all scopes that need to be analyzed to be able to verify the remaining dereferences.

Allocate dereferences to analysis scopes For each scope of analysis identified in the previous step, find all dereferences that should be analyzed in this scope.

Perform limited-scope interprocedural analysis Perform limited-scope interprocedural analysis on each analysis scope in an attempt to verify the safety of remaining dereferences allocated to the scope.

Extend dereference contexts for unverified dereferences Refine the contexts of those of the remaining dereferences that were not shown to be safe.

Consider a dereference statement S that occurs in a method m. To track the calling context in which S should be analyzed, we associate it with a *dereference context*. The dereference context is a chain in the call graph that starts with an ancestor of m (that we call *context root*) and ends with m, the method containing S. For instance, the sole dereference context associated with the 2-scope analysis of `getSalary` is ⟨`test`, `EmpRec<init>`, `getSalary`⟩. The analysis maintains a worklist of all dereferences whose safety has not yet been established, together with the dereference contexts in which they need to be analyzed (we refer to a pairing of a dereference with a context as an *unprocessed dereference*).

Compute analysis scopes. For a method m, we say that a subgraph of the program call graph, is a (u, d)-*scope* for m if, when considered without back edges,[2] the subgraph forms a dag rooted at method p, located at distance u above m in the call graph, and the subgraph contains all descendants of p up to depth d (we refer to parameter d as the *scope depth*). We refer to method p as the *scope root*. We refer to (u, d)-scope analysis where $u = d = k$ as k-*scope* analysis. Intuitively, parameters u and d in the analysis of method m control how much caller and callee context is considered when attempting to verify the dereferences in m. Parameter u controls how high up the chain of callers from m are the callers that a scope should include, and parameter d controls how deep below scope root are the nodes with a common ancestor with m (namely, scope root) that the scope should include.

Fig. 3 shows the call graph for our running example augmented with an additional test method, test2. (The code of test2 is irrelevant for the discussion.) A $(0, 1)$-scope analysis of method test attempts to verify the safety of dereferences in test in scope s = {test, EmpRec<init>, toString}, rooted at test. A $(1, 1)$-scope analysis of EmpRec<init> attempts to verify the safety of dereferences in the constructor in scope s, as well as in scope {test2, EmpRec<init>, toString}, rooted at test2.

The context roots of all unprocessed dereferences define the roots of the analyses scopes that need to be analyzed in an attempt to verify those dereferences as safe. For an unprocessed dereference with context $\langle m_1, m_2, \ldots, m_u\rangle$, the analysis will compute a (u, d)-scope, where d is defined by the expanding mechanism in use. For instance, if the analysis expands scopes above the analyzed method, then $d = u$, and if the analysis expands scopes around the analyzed method, then $d = 2 * u$.

Allocate dereferences to scopes. When performing the analysis in a given context, we try to establish the safety of only those unprocessed dereferences whose context root is the same as the root method of the scope. This prevents unnecessary re-analysis. If a given unprocessed dereference is marked as safe during the analysis of a particular scope, it can be safely discarded (the dereference statement is safe in the corresponding calling context). A given scope rooted at method p is thus responsible for the verification of the safety of all unprocessed dereferences whose context starts with p.

Limited-scope analysis. We lift the intraprocedural analysis described in §4 into an interprocedural setting using the IFDS framework of Reps-Horwitz-Sagiv [17]. In this framework, every method is summarized by an input-output relation between sets of non-null access paths. (Recall that the other components of the abstract domain are not updated after by the preliminary analysis.)

Given an analysis scope, the analysis proceeds as if performing whole-program analysis. However, when it encounters a call to a method m that lies outside the analysis scope, it reflects the effect of the call conservatively by removing *mayKill*(m) (the set of access paths that may be modified by m) from the non-null access-path set collected prior to the call. In §6, we explain how to improve the handling of calls to methods outside the analysis scope.

Extend dereference contexts. When the analysis of a method in a given scope fails to establish the safety of an unprocessed dereference, we need to expand the scope of analysis.

To do this, we extend the context of the dereference by considering every possible caller of the current context root. For example, extending the scope of analysis for a dereference with context ⟨EmpRec<init>, getSalary⟩ results in two context dereferences: ⟨test, EmpRec<init>, getSalary⟩ and ⟨test2, EmpRec<init>, getSalary⟩.

Suppose method toString invoked method getSalary using the string "John Doe:40000" as a parameter (this is not the case in the example). In that case, 1-scope analysis of getSalary with dereference context ⟨toString, getSalary⟩ would succeed in verifying the safety of the dereference of empStr on line 16. However, as discussed above, 1-scope analysis will fail to do this with dereference context ⟨EmpRec<init>, getSalary⟩. Our analysis will extend only the latter context.

The analysis keeps expanding the analysis scope until it reaches a preset limit (or succeeds in marking all dereferences as safe).

Analysis of the running example. As stated in §2.1, 0-scope analysis of the scopes shown in Fig. 2 (a) succeeds in verifying that method test contains no unsafe dereferences. 1-scope analysis of the scope rooted at EmpRec's constructor—which also includes getName and getSalary—fails to verify the unprocessed dereferences of empStr on lines 12 and 16, as the value comes from test, which is not included in the scope. 1-scope analysis of the scope rooted at test—which also includes the constructor and toString—succeeds in verifying the safety of the remaining unprocessed dereference in the constructor on line 6. 2-scope analysis of the scope rooted at test—which includes all methods in this example program—succeeds in verifying the safety of the unprocessed dereferences of empStr in getName and getSalary. The analysis identifies these dereferences as safe because it finally includes test and the constructor, which witness the flow of the constant "Jane Smith:50000" from test to getName and getSalary.

The dereferences on lines 30 and 32 are not handled by a (u, d)-scope analysis, unless $d > u$ (and in k-scope analysis $d = u$). This is because k-scope analysis of method toString does not include methods getSalary and getCurrency that lie "below" toString in the call graph, and yet are needed for verifying the safety of toString. The obvious solution is to perform a (d, u)-scope analysis with $d > u$ but this incurs a significantly higher cost. In §6.1 and §6.3, we explain how to handle such dereferences while still using k-scope analysis.

6. STAGED ANALYSIS

These are SALSA's analysis stages, in order of execution:

- **Preliminary**: preliminary analyses, described in §3, construct the program flow graph and perform alias analysis.
- **Pruning**: pruning analysis, described in §6.1, verifies the safety of some dereferences, e.g., by identifying non-null final and stationary fields [22].
- **Caller**: caller-guarantee analysis, described in §6.2, uses a 0-scope analysis to compute for every method non-nullness information which holds whenever it is invoked.
- **Callee**: callee-guarantee analysis, described in §6.3, uses a 0-scope analysis to compute for every method non-nullness information which always holds when the method returns.
- **1-scope**: a 1-scope analysis, as described in §5.
- **2-scope**: a 2-scope analysis, as described in §5.

[2] We handle recursion soundly via a fixedpoint computation over strongly connected components, as described later.

6.1 Pruning Analysis

Pruning analysis prunes out dereferences that can be seen as safe given a simple dataflow computation. The primary kinds of dereferences that are pruned out are those of final and stationary fields that are assigned non-null values. For example, in method `toString` of the running example, pruning analysis identifies the dereference of final field `name` on line 27 as safe. Additionally, pruning analysis finds that field `salary` is stationary (in fact, it could have been declared final) and is assigned a non-null value, thus identifying the dereference on line 30 as safe. In this method, pruning analysis leaves only the dereferences of the local variables `result` and `currency` to subsequent stages.

6.2 Caller-Guarantee Analysis

Caller-guarantee analysis computes for every method m the set of access paths that all callers of m guarantee to be non-null at m's entry. Caller-guarantee analysis initializes a worklist to contain all methods in the call graph in topological order (which helps capture the flow of non-null values from callers to callees). For each method removed from the worklist, the algorithm performs local (0-scope) analysis. When the analysis encounters a call to method m, it updates the guarantee provided by callers of m by taking the intersection of the current guarantee set with the non-null access paths for this call. If the guarantee set shrinks, m is placed on the worklist for the sound handling of recursion.

The results of the caller-guarantee analysis help improve the precision of the following stages: during the analysis of a scope rooted at m, the set of non-null access paths AP_{nn} is initialized to the caller-guarantee non-null access-path set computed for method m. For example, during the analysis of method `test` in the running example, caller-guarantee analysis records the guarantee that the `empStr` parameter of the constructor is non-null. Then, during the analysis of the constructor, caller-guarantee analysis records the guarantee that the `empStr` parameters of `getName` and `getSalary` are non-null. The subsequent analyses of `getName` and `getSalary` identify the dereferences of `empStr` as safe without expanding scopes.

6.3 Callee-Guarantee Analysis

Callee-guarantee analysis computes for every method m the set of access paths that are guaranteed to be non-null upon m's return. Callee-guarantee analysis initializes a worklist to contain all methods in the call graph in reverse topological order (which helps capture the flow of non-null values from callees to callers). For each method removed from the worklist, the algorithm performs local (0-scope) analysis. When the analysis encounters a return statement, it updates the guarantee that m provides to its callers by taking the intersection of the current guarantee set with the non-null access-path set collected at the return statement. If the guarantee set shrinks, m's callers are placed on the worklist for the sound handling of recursion.

When a limited-scope analysis encounters a call to method m that lies outside the analysis scope, the analysis adds the callee-guarantee set (mapped to caller's variables) to the set of non-null access paths AP_{nn}.(The analysis first removes the members of the set $mayKill(m)$ from AP_{nn}, as before.) For example, during the analysis of `getCurrency` in the running example, callee-guarantee analysis records the guarantee that the method's return value is non-null. The subsequent analysis of `toString` identifies the dereference of `currency` on line 32 as safe without expanding scopes. During the analysis of `getSalary`, callee-guarantee analysis records the guarantee that this method's return value is also non-null.

Final analysis of the running example. The staged analysis described above is able to verify the safety of all dereferences via simple steps. First, pruning analysis identifies dereferences of final field `name` and stationary field `salary` as safe. Then, caller-guarantee analysis succeeds in verifying the dereferences of parameter `empStr` in the constructor, as well as in methods `getName` and `getSalary`, because the string constant `"Jane Smith:50000"` flows from `test`. Finally, callee-guarantee analysis succeeds in verifying the only remaining dereference—that of `currency` in `toString` —because a non-null return value flows from `getCurrency`.

7. USER ANNOTATIONS

When there are dereferences that cannot be established as safe by the analysis, SALSA can leverage information specified via user annotations. The basic mechanism for the user to declare an unverified dereference as safe is by adding an annotation for that single dereference. However, the user can also provide the tool with higher-level information as described below.

We note that our goal is to reduce the number of annotations that have to be introduced into the code. Fewer annotations mean fewer places that the programmer has to reason about, and fewer changes to apply when the code is modified. Additionally, we want to help the programmer express knowledge about her program that goes beyond the null-check of a local value. Ideally, we would like to enable the programmer to express the strongest knowledge she has about program behavior (with respect to nullness of values).

7.1 Annotations

Our annotations are essentially the same as the annotations proposed in JSR 305 [15]. We deliberately choose to keep the annotation language simple, even at the cost of limited expressiveness.

We support the three kinds of annotations described in Tab. 3. Annotations **@CheckForNull**(*ap*) and **@NonNull**(*ap*) operate on an access-path expression *ap* that can be a local variable, a static field, an expression of the form *pexp.fld* where *fld* is an instance field, or a designated variable *ret* that denotes a method's return value. Annotation **@NonNullable**(*fld*) operates on a field *fld*.

Annotation **@CheckForNull**(*ap*) corresponds to a required runtime check that ensures that the value of *ap* is non-null. Annotation **@NonNull**(*ap*) allows the user to express that a parameter to a method, or a method's return value can be assumed to be non-null. Annotation **@NonNullable**(*fld*) allows the user to express that an instance field maintains a non-null value in each containing object. The latter is inspired by the notion of non-null types proposed in previous work [1, 10], and requires that: (i) every assignment to the field is of a non-null value; (ii) all accesses to the field take place after an assignment.

These annotations are limited and do not allow the programmer to capture notions such as varying nullness of fields. However, we will see in §8 that even these simple annotations can go a long way.

7.2 Suggesting Annotations to the User

Our system provides user guidance in the form of suggested **@NonNull** and **@NonNullable** annotations. The system produces a list of these annotations ranked by their expected effect on the overall number of required annotations.

It is important to note that inferring **@NonNull** and **@NonNullable** annotations from **@CheckForNull** is a strengthening of the asserted information. In general, such strengthening may be based on information that is known to the programmer, but is very hard or even impossible to infer from the program itself. Our current strategies for suggesting annotations are based on simple heuristics.

Annotation	Meaning
@CheckForNull(*ap*)	Require the access path *ap* to have a non-null value at runtime.
@NonNull(*ap*)	Assume the access path *ap* to have a non-null value.
@NonNullable(*fld*)	Assume the field *fld* to have a non-null value.

Table 3: Non-nullness annotations, *ap* is an access path, *fld* is a field

8. EXPERIMENTAL RESULTS

In this section, we describe our prototype implementation, and evaluate our approach over a number of benchmarks. We start by evaluating the approach without any user-provided annotations, and then show how the system helps the programmer provide a small number of annotations that make a significant impact.

8.1 Implementation

We have implemented our approach in a tool called SALSA, based on the WALA framework [23]. SALSA handles the full Java language, excluding concurrency, subject to caveats regarding dynamic language features such as reflection. The analysis deals with reflection by tracking objects to casts, as in [11]. Subject to these standard caveats, our implementation is sound.

Focusing the attention of the analysis on a small set of dereference statements in a given scope enables a number of optimizations that we implemented in SALSA. For instance, we use a cheap and conservative slicer implementation to cull from the scope those methods that have no bearing on the value being dereferenced at any of the statements in question. The slicer analyzes the scope by assuming that it is a complete program call graph. However, it soundly over-approximates the effect of program methods that lie outside the scope by adding a *heap dependence* from a statement that modifies an object to all statements that may access that object according to may-alias information. Such *intra-scope* slicer invocations incur negligible execution cost but often dramatically reduce the sizes of analyzed scopes. In the example of Fig. 2, slicing removes methods `toString` and `getCurrency` from 2-scope analysis of the scope rooted at `test`. This leaves exactly the methods that are relevant to establishing the safety of dereferences in `getName` and `getSalary`. In the future, we plan to start using slicing for even finer control—by skipping individual statements' transformers we expect to achieve greater scalability.

8.2 Benchmarks

Tab. 4 lists the benchmarks used in this paper. Apache `Ant` is a Java-based build tool. `Antlr` is a parser generator. `ArtOfIllusion` (`aoi`) is an open source 3D modeling and rendering studio. Apache `Bcel` is a bytecode toolkit with a sample verifier. `flickrj` is a Flickr client. `Fluid` is a server daemon for streaming media. `freecol` is an open-source strategy game. `freemind` is a "brain mapping" content-management system. `ganymed` is a library that implements the SSH-2 protocol in pure Java. `Java_cup` is a parser generator. `Jbidwatcher` is an online auction tool. `jgnash` is a personal finance desktop application. `JLex` is a lexical analyzer. `jo` is a pure Java webserver. `kolm` is a desktop tool which interfaces with an online adventure game. `L2j` is Multi-User Dungeon game server. `ourtunes` is a popular Java-based itunes browser. `pjirc` is a Java-based IRC client. `toyWS` is a small web-server written in Java. `tvla` is a static analysis framework. `warrior` is an open-source web browser that is written completely in Java.

Num	Benchmark	Classes	Methods	Bytecodes	Derefs
1	ant	575	4311	353553	4486
2	antlr	109	1337	191492	14803
3	aoi	158	1232	104446	5853
4	bcel	297	1754	88543	10252
5	flickrj	32	152	6052	608
6	fluid	33	187	12648	1109
7	freecol	237	2083	157582	13729
8	freemind	360	2408	119294	10838
9	ganymed	91	443	70193	7755
10	javacup	32	279	28194	2856
11	jbidwatcher	216	1494	167184	16648
12	jgnash	1215	7340	465943	37225
13	Jlex	24	130	25534	2570
14	jo	10	41	3010	321
15	kolm	237	2726	293003	25174
16	l2j	280	1868	104872	10562
17	ourtunes	64	243	20548	1640
18	pjirc	37	109	6298	612
19	toyWS	13	48	3439	372
20	tvla	222	1284	105017	9587
21	warrior	333	1880	129742	7460
	Total	**4575**	**31349**	**2456587**	**179443**

Table 4: Benchmarks. The numbers exclude all library code used by the applications.

8.3 Methodology

In this section, we evaluate our analysis with respect to two major parameters that affect its precision: access-path length (see §3), and scope depth (see §5).

In order to evaluate our techniques, we consider the number of warnings generated for our benchmarks with various choices for the above parameters. In §8.4, we describe the results obtained for our benchmarks without any user-provided annotations. Then, in §8.5 we show how the system helps the user record her knowledge about program behavior with simple high-level annotations that reduce the number of warnings. The final results obtained for our benchmarks are summarized in Tab. 6. It is important to note that caller and callee guarantees play a key role in the effectiveness of analysis in smaller scopes described in the following results.

8.4 Initial Results

In this section we describe the number of warnings that are generated automatically without any user assistance.

Tab. 5 shows the effect of scope depth on the number of warnings generated when the access-path length is limited to 2. Fig. 4 presents the data of Tab. 5 pictorially; it shows the percentage of dereferences handled by each of the stages for each benchmark. The rightmost column shows the aggregate percentages over all benchmarks. Starting from 100%, the contribution of the different stages is shown in top-down order: pruning, caller, callee, 1-scope, and 2-scope. The percentage of remaining annotation is shown at the bottom of the chart.

As shown in the chart, there is a clear trend of diminishing returns from deeper stages, which is natural and expected. Pruning itself is sophisticated enough to dismiss around 8% of the unveri-

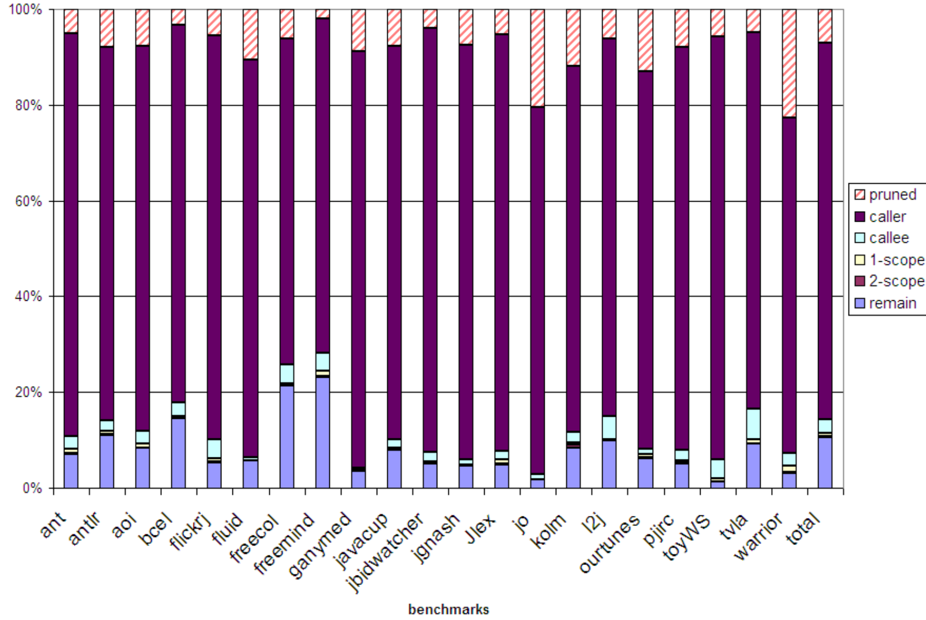

Figure 4: Percentage of dereferences handled by each stage, and percentage of dereferences that result in warnings. Results were obtained with access-path length limited to 2, and scope depth limited to 2.

fied dereferences on average. The top-down analysis using caller guarantees (see §6.2) is able to dismiss around 77% of the unverified dereferences on average, and the deeper analyses are targeting the remaining 15%. Note that this chart corresponds to stages up to scope depth 2 with access-path length limited to 2. These settings allow comparison across all benchmarks. The final results obtained in this paper (Tab. 6) are obtained using varying settings for different benchmarks, specified based on the expected cost of analysis.

The final results show that by using a larger scope depth (and sometimes a larger limit on access-path length), the number of remaining warnings can be further automatically reduced to an average of 10.3% of all potentially unsafe dereferences. The running time of the final analyses did not exceed 3 hours for any benchmark.

For most benchmarks, increasing the access-path length beyond 2 does not have an effect on the number of generated warnings. However, for a few benchmarks, using access paths of length greater than 2 does reduce the number of generated warnings. The benchmarks that benefit from long access paths are written in a C-like style in which fields are accessed directly rather than through accessor methods. For example, for the `JLex` benchmark with scope limited to 2, limiting the length to 2 yields 238 warnings, and extending it to 4 reduces the number of warnings to 127.

There are several conclusions that we draw from the evaluation:

- increasing the access-path length limit beyond 2 is not beneficial on most benchmarks.
- increasing the scope depth has diminishing returns in terms of the numbers of dereferences handled. This has a lot to do with the fact that caller-guarantee and callee-guarantee analyses have propagated a large portion of the information that requires wide scopes.
- one of the main sources of imprecision in our analysis is deep initialization sequences observed in some benchmarks (e.g., `freemind`). Such initialization sequences require a rather large scope depth, and are thus costly to observe.

While we can keep increasing the scope depth, covering larger and larger scopes, it is sometimes more efficient to get user assistance in the form of higher-level annotations, as we describe in the following section.

We expected some false alarms to be caused by the relative simplicity of our abstract domain. For instance, we expected branch correlation to be a significant cause of false warnings. Interestingly, this was not the case. Nonetheless, we implemented a heuristic symbolic path-validity checker that we used for ranking reports issued by SALSA. In our preliminary experiments, our ranking selected around 20% of the reports as high-priority warnings. In future work, we plan to use a more advanced interprocedural path validator in order to produce finer rankings of reports, and possibly generate concrete counterexamples.

8.5 Leveraging User Annotations

In some cases, a large number of warnings are generated for dereferences of values obtained from one field. The analysis may not be precise enough to establish that all accesses to the field obtain a non-null value, for a variety of reasons. For example, the scope depth may be insufficient to observe the temporal ordering of calls that set the field to a non-null value.

Tab. 6 shows the number of annotations added by the user, and their effect on the overall number of produced warnings. In this experiment, we do not use **@CheckForNull** annotations.

We only added user annotations to 8 of the benchmarks for which the automated phase reported warnings for a high percentage of dereferences. For these 8 benchmarks, we show that adding a total number of 173 annotations reduced the number of remaining warnings by an average of 30%.

In `antlr`, a large number of generated warnings are due to `grammar` fields that record the parent grammar of various entities. These are non-nullable fields that are not modified once set, and at least in some cases can be even made final.

Benchmark	initial	pruned	caller	callee	1-scope	2-scope
ant	4486	4263	482	370	328	317
antlr	14803	13642	2090	1754	1656	1634
aoi	5853	5402	691	545	494	486
bcel	10252	9912	1827	1543	1497	1485
flickrj	608	574	61	37	33	32
fluid	1109	992	70	64	63	63
freecol	13729	12871	3548	2992	2950	2933
freemind	10838	10629	3052	2641	2530	2510
ganymed	7755	7076	323	287	275	266
javacup	2856	2635	288	242	237	228
jbidwatcher	16648	15972	1254	921	862	832
jgnash	37225	1088	69	58	56	55
Jlex	2570	2434	304	264	245	238
jo	321	255	9	6	6	6
kolm	25174	22160	2931	2393	2278	2134
l2j	10562	9900	1578	1070	1045	1042
ourtunes	1640	1428	134	117	103	101
pjirc	612	564	49	35	34	31
toyWS	372	351	22	7	5	5
tvla	9587	9126	1582	972	891	881
warrior	7460	1888	178	115	83	73
total	**184460**	**171723**	**26044**	**20709**	**19610**	**19243**

Table 5: Number of warnings generated with access-path length limited to 2, and interprocedural scope limited to 2.

In bcel, around 800 of the generated warnings are due to accesses to the stack field in the class Frame, which can be made into a non-nullable field. Once this single annotation is added, the number of warnings is dramatically reduced.

In freecol, the implementation widely uses initialization of values by either initializing them in a constructor, or reading them from a serialized file. This widely used pattern prevents the designer from declaring the fields as final, although in most cases the values are indeed not modified. This pattern makes the non-nullness of a field harder to observe, as it becomes dependent on calling a method other than the constructor to guarantee that the field has a non-null value. Additionally, nearly 2200 of the generated warnings in freecol are due to dereferencing of return values from 137 getters and other simple methods that can in fact return null. Over 1100 of these potentially unsafe dereferences are due to 10 getter methods. A redesign of these methods can improve the quality of the code and lead to a substantially lower number of warnings.

In freemind, highly customizable controller model and other parts of the framework use null to denote options that are not used, empty collections, etc. The system model uses heavy delegation even for the simplest operations, making it hard to verify the safety of dereferences within small scopes.

In JLex, minor changes to field initialization ordering in some constructors can make a few more fields unconditionally initialized to a non-null value, reducing the final number of warnings to 40.

In tvla, a class hierarchy implementing logical formulae consists mostly of final non-nullable fields. The system suggests some of these annotations, which are the majority of the total of 65 user annotations added to this benchmark. Since some authors of this paper are familiar with this code base, adding annotations was even easier than adding them in other benchmarks.

In pjirc, we do not add user annotations, but it is interesting to observe that the source of 6 of the checks is due to calls such as getArray("style:smiley"). Establishing that such calls return a non-null value requires reasoning about actual key values, which is a very challenging task.

Bench	AP	SD	Initial	Auto		User	Final	
ant	2	2	4486	315	7.0%	0	315	7.0%
antlr	2	4	14803	1628	11.0%	10	1194	8.1%
aoi	2	2	5853	486	8.3%	0	486	8.3%
bcel	2	4	10525	1481	14.1%	1	668	6.3%
flickrj	2	2	608	32	5.3%	0	32	5.3%
fluid	2	2	1109	63	5.7%	0	60	5.4%
freecol	2	4	13729	2889	21.0%	45	1384	10.1%
freemind	2	4	10838	2491	23.0%	21	2280	21.0%
ganymed	2	4	7755	258	3.3%	0	258	3.3%
javacup	4	5	2856	223	7.8%	10	190	6.7%
jgnash	2	4	37225	3453	9.3%	0	3453	9.3%
jbidwatch	2	4	16648	813	4.9%	14	716	4.3%
Jlex	4	3	2570	127	4.9%	7	78	3.0%
jo	2	2	321	6	1.9%	0	6	1.9%
kolm	2	2	25174	2134	8.5%	0	2134	8.5%
l2j	4	4	10562	1042	9.9%	0	1015	9.6%
ourtunes	3	4	1640	129	7.9%	0	129	7.9%
pjirc	2	2	612	31	5.1%	0	31	5.1%
toyWS	2	3	372	4	1.1%	0	4	1.1%
tvla	2	2	9587	890	9.3%	65	517	5.4%
warrior	2	2	7460	550	7.4%	0	550	7.4%
Total			**184733**	**19045**	**10.3%**	**173**	**15500**	**8.4%**

Table 6: Number of warnings generated, number of annotations accepted by the user, and the final number of warnings. The columns AP and SD show the access-path length and scope depth used, respectively.

For some benchmarks, we have done a preliminary study of what constitutes the remaining warnings. Our preliminary exploration suggests that some of the interesting causes are:

- higher-level correlations known to the programmer (e.g., list of argument names is not null implies that list of argument types is not null in Procedure representation of tvla.)
- implicit assumptions on the temporal ordering of method calls. In these cases, observing that a dereference is safe can be reduced to verifying a typestate property [20].
- access to indexed data-structures (for example, getArray("style:smiley") in pjirc.)

8.6 Comparison with Local Bug-Finding

In order to better understand the usability of SALSA in comparison to tools aimed at bug finding, we implemented two bug-finding tools, Loco and Xylem. We applied them, together with FindBugs [14], to our suite of benchmarks. All three tools essentially perform local analysis. In the absence of information about the callers and callees of a method, the tools rely on the programmer's *beliefs* about the nullness of values: whether or not a pointer dereference is guarded by a test against null constitutes a belief about whether or not the pointer may be null. FindBugs and similar bug detectors find likely bugs by identifying contradictory beliefs. A dereference of a pointer that occurs outside a test of that pointer against null presents a contradiction: the presence of the test indicates that the programmer believes that the pointer may be null, while the dereference of that pointer outside the scope of the test constitutes a belief that the pointer may not be null. It must be the case that either the test is superfluous or that a null dereference may occur.

Benchmark	FindBugs	Loco	Xylem
ant	2	9	24
antlr	1	17	14
aoi	8	27	44
bcel	1	4	7
flickr	3	4	4
fluid	0	0	4
freecol	6	17	0
freemind	1	4	7
ganymed	0	0	11
javacup	0	0	0
jbidwatcher	0	0	22
jgnash	7	3	55
JLex	1	9	13
jo	2	5	14
kolm	5	0	0
l2j	2	4	100
ourtunes	0	0	0
pjirc	0	0	0
toyWS	0	0	0
tvla	4	9	23
warrior	2	2	4

Table 7: Local tool results

The focus on belief contradictions can result in missed errors, as well as false warnings. The absence of a test against null prevents such tools from issuing any warnings. As discussed earlier, the majority of the warnings issued by SALSA for `freecol` are due to the dereferences of return values of methods that may return null. For that application, the hundreds of legitimate concerns raised by SALSA result in no warnings on the part of the contradiction-based tools at our disposal. For instance, those tools did not issue a warning for any dereference of the return value of the following method:

```
public Tile getTile(int x, int y) {
  if ((x >= 0) && (x < getWidth()) &&
      (y >= 0) && (y < getHeight())) {
    return columns.get(x).get(y);
  } else {
    return null;
  }
}
```

While much less common, false positives can also arise. They occur when a value that can never be null is compared with null. The following snippet is inspired by IBM software:

```
1  Value lhsVal = getValue(lhs).copy();
2  for (int j = 0; j < uses; j++) {
3    int currUse = instr.getUse(j);
4    if (currUse != -1) {
5      Value currVal = getValue(currUse);
6      if (currVal != null && lhsVal != null) {
7        lhsVal = join(lhsVal, currVal);
8      }
9    }
10 }
11 if (!lhsVal.equals(getValue(lhs))) { ...
```

Method `copy` invoked on line 1 and method `join` invoked on line 7 always return non-null values. The redundant test of variable `lhsVal` results in a report of a possible null dereference of `lhsVal` on line 11 by the three contradiction-based tools.

Tab. 7 shows the number of reports collected by FindBugs, Loco, and Xylem. The number of reports produced by Loco and Xylem is generally higher than the number of reports produced by FindBugs. FindBugs appears more aggressive in suppressing warnings that arise on few paths through a method. For instance, the dereference of `target` on the last line of the following snippet of `freecol` results in a warning from Loco and Xylem, but not from FindBugs:

```
Colony target = null;
for (int i = 0; i < nearbyColonies.size(); i++) {
  Tile t = getGame().getMap().getTile(it.next());
  ...
  if (tension > targetTension) {
    targetTension = tension;
    target = t;
  }
}
Iterator<Unit> it = settlement.getOwnedUnitsIterator();
AIUnit chosenOne = null;
while (it.hasNext()) {
  chosenOne = (AIUnit) getAIMain().getAIObject(it.next());
  ...
}
if (chosenOne != null) {
  PathNode pn = chosenOne.getUnit().findPath(
    settlement.getTile(), target.getTile());
```

If the first `for` loop never executes, the execution of the last line will necessarily result in a null dereference. However, FindBugs chooses to suppress such a warning to minimize false alarms. There are also cases when FindBugs issues a warning, while Loco and Xylem do not. This is not surprising, as the three tools use different heuristics to suppress undesirable warnings. The results of the three tools are often incomparable.

All tools that ignore interprocedural flow make assumptions about the missing context. The number of false alarms that would result from a *sound* local analysis makes such an approach of little practical utility. Thus, local tools generally make optimistic assumptions about the context, and report only highly suspicious code. Such tools provide a complementary approach to that of SALSA. For instance, contradiction-based bug detectors are particularly well-suited for finding obvious problems early in the development cycle. However, as a software product matures, more complete coverage, such as that offered by SALSA, may become a worthy investment. Note that it is possible to combine contradiction-based analysis with expanding-scope analysis. For instance, searching for contradictions between nullness assumptions made in callers and callees may yield better coverage.

9. RELATED WORK

Analyses for null dereferences and similar safety properties can be roughly classified as *verification*, or *bug finding*. Verification aims to prove the absence of problems in any possible execution, and bug-finding analyses aim to find defects without promising to be either sound or complete. From the vast literature covering this space, we briefly review some of the most relevant related work.

PREfix [2] can detect possible null-dereference errors in C and C++ programs with only a few false positives. It employs a path-sensitive interprocedural analysis. The analysis, however, is not conservative because only a few paths are considered (usually 100 paths in every function). PREfast [16] is a more lightweight tool which attempts to find (in an unsound way) idioms associated with programming mistakes in C and C++ programs.

Tomb et. al. [21], introduce the notion of variably interprocedural analysis that roughly corresponds to a subgraph of the call-graph, similar to the program fragments we consider. They paper uses such program fragments to perform symbolic path simulation that is later combined with a dynamic analysis. In contrast, our approach uses a sound abstract interpretation and the notion of expanding analysis scopes.

The Spec# [1] system checks specifications both statically via an automatic program verifier (Boogie), and dynamically via assertions. The language includes non-null types, as well as a subset of C# for specifying preconditions, postconditions, frame conditions, and object invariants. The Boogie verifier employs interprocedural

abstract interpretation to learn loop invariants, and then employs a theorem prover to check feasibility of contract violations. The system is incorporated into the Visual Studio IDE, and provides instantaneous feedback via decorations in the program editor.

ESC/Java [13] uses a theorem prover to find defects related to a class of errors which includes null dereferences as well as array-bounds errors, type errors, and synchronization problems. In contrast to our approach, ESC/Java does not attempt to find all errors. ESC/Java relies on annotations to summarize external behavior, without any interprocedural analysis, and so introduces an interactive methodology whereby the user adds annotations to suppress tool findings. Later work on ESC/Java2 [3] adapted the ESC checker to the JML specification language.

To reduce the annotation burden for ESC/Java, the Houdini [12] tool infers annotations by searching a range of plausible annotations, and then employing the ESC checker to refute invalid candidates. As in our expanding-scope analysis, Houdini must make either optimistic or pessimistic assumptions regarding unanalyzed libraries; Houdini chooses optimistic assumptions, which reduce the number of false alarms compared to the pessimistic alternative.

Saturn [24] checks C programs for a number of safety properites. Saturn translates the program and safety properties into a boolean constraint, and invokes a SAT solver to check feasibility of error states. Saturn employs no abstraction, and is underapproximate in the presence of loops. Notably, Saturn computes modular summaries to enable bottom-up interprocedural summary-based analysis, enabling the tool to scale to large code bases.

Annotations of program behavior correspond to *invariants* that must hold at a program point. Many works have targeted static and/or dynamic approaches to learn likely invariants. Most notably, the Daikon [9] tool employs profiling and dynamic analysis to infer likely invariants from program executions.

Approches based on shape analysis [19], e.g., [7, 8], enable the conservative verification of the safety of pointer dereferences, including those that occur during traversals of complicated date structures. These approaches use complicated invariants to capture the shape of the program's heap. It is very challenging to scale these approaches to realistic Java programs due to their use of expensive abstract domains.

Fähndrich and Leino check the null-dereference property using non-null types [10]. Their approach focuses on the important problem of verifying accesses to partially constructed objects. Having the user annotate methods with non-null types allows for modular checking. Our works handles this problem conservatively by propagating non-nullness information interprocedurally.

10. CONCLUSIONS AND FUTURE WORK

We present a novel approach for a *staged analysis* based on the notion of expanding scopes. We applied our approach to the challenging problem of verifying dereference safety in real Java programs. In our experiments, we found that in many cases individual dereference statements can be verified as safe using precise analyses that consider limited scopes. We believe that our approach for staged analysis can also be effective for typestate [20] verification; we plan to investigate it in the future.

11. REFERENCES

[1] BARNETT, M., LEINO, K., AND SHULTE, W. The Spec# programming system: An overview. In *CASSIS* (2004).

[2] BUSH, W. R., PINCUS, J. D., AND SIELAFF, D. J. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper. 30*, 7 (2000).

[3] COK, D. R., AND KINIRY, J. ESC/Java2: Uniting ESC/Java and JML. In *CASSIS* (2004).

[4] COUSOT, P., AND COUSOT, R. Systematic design of program analysis frameworks. In *POPL* (1979).

[5] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP* (1995).

[6] DIWAN, A., MCKINLEY, K., AND MOSS, J. Type-based alias analysis. In *PLDI* (1998).

[7] DOR, N., RODEH, M., AND SAGIV, M. Detecting memory errors via static pointer analysis (preliminary experience). In *PASTE* (1998).

[8] DOR, N., RODEH, M., AND SAGIV, M. Checking cleanness in linked lists. In *SAS* (July 2000).

[9] ERNST, M., COCKRELL, J., GRISWOLD, W., AND NOTKIN, D. Dynamically discovering likely program invariants to support program evolution. In *ICSE* (1999).

[10] FÄHNDRICH, M., AND LEINO, K. Declaring and checking non-null types in an object-oriented language. In *OOPSLA* (2003).

[11] FINK, S., YAHAV, E., DOR, N., RAMALINGAM, G., AND GEAY, E. Effective typestate verification in the presence of aliasing. In *ISSTA* (2006).

[12] FLANAGAN, C., AND LEINO, K. Houdini, an annotation assistant for ESC/Java. In *Proc. FME 2001.* (2001).

[13] FLANAGAN, C., LEINO, K., LILLIBRIDGE, M., NELSON, G., SAXE, J., AND STATA, R. Extended static checking for Java. In *PLDI* (2002).

[14] HOVEMEYER, D., AND PUGH, W. Finding bugs is easy. *SIGPLAN Not.* (2004).

[15] JSR 305: Annotations for software defect detection. http://jcp.org/en/jsr/detail?id=305.

[16] LARUS, J., BALL, T., DAS, M., DELINE, R., FAHNDRICH, M., PINCUS, J., RAJAMANI, S., AND VENKATAPATHY, R. Righting software. *IEEE Software* (2004).

[17] REPS, T., HORWITZ, S., AND SAGIV, M. Precise interprocedural dataflow analysis via graph reachability. In *POPL* (1995).

[18] ROUNTEV, A., RYDER, B., AND LANDI, W. Data-flow analysis of program fragments. In *FSE* (1999).

[19] SAGIV, M., REPS, T., AND WILHELM, R. Parametric shape analysis via 3-valued logic. *TOPLAS* (2002).

[20] STROM, R., AND YEMINI, S. Typestate: A programming language concept for enhancing software reliability. *TSE* (1986).

[21] TOMB, A., BRAT, G., AND VISSER, W. Variably interprocedural program analysis for runtime error detection. In *ISSTA* (2007).

[22] UNKEL, C., AND LAM, M. Automatic inference of stationary fields: a generalization of java's final fields. In *POPL* (2008).

[23] WALA: The T. J. Watson Libraries for Analysis. http://wala.sourceforge.net.

[24] XIE, Y., AND AIKEN, A. Saturn: A scalable framework for error detection using boolean satisfiability. *TOPLAS* (2007).