Null Dereference Verification via Over-approximated Weakest Pre-conditions Analysis

Ravichandhran Madhavan *
Microsoft Research India, Bangalore
t-rakand@microsoft.com

Raghavan Komondoor
Indian Institute of Science, Bangalore
raghavan@csa.iisc.ernet.in

Abstract

Null dereferences are a bane of programming in languages such as Java. In this paper we propose a sound, demand-driven, inter-procedurally context-sensitive dataflow analysis technique to verify a given dereference as safe or potentially unsafe. Our analysis uses an abstract lattice of formulas to find a pre-condition at the entry of the program such that a null-dereference can occur only if the initial state of the program satisfies this pre-condition. We use a simplified domain of formulas, abstracting out integer arithmetic, as well as unbounded access paths due to recursive data structures. For the sake of precision we model aliasing relationships explicitly in our abstract lattice, enable strong updates, and use a limited notion of path sensitivity. For the sake of scalability we prune formulas continually as they get propagated, reducing to $true$ conjuncts that are less likely to be useful in validating or invalidating the formula. We have implemented our approach, and present an evaluation of it on a set of ten real Java programs. Our results show that the set of design features we have incorporated enable the analysis to (a) explore long, inter-procedural paths to verify each dereference, with (b) reasonable accuracy, and (c) very quick response time per dereference, making it suitable for use in desktop development environments.

Categories and Subject Descriptors D [*2*]: 4—Assertion checkers; F [*3*]: 1—Assertions, Mechanical verification

General Terms Algorithms,Experimentation,Verification

* Part of this work was done while the author was affiliated with Indian Institute of Science, Bangalore.

OOPSLA'11, October 22–27, 2011, Portland, Oregon, USA.


```
1: foo(a,d) {
2:  b = null;
3:  if (a == null)
4:    b = d;
5:  c = a;
6:  if (c != null)
7:    b.g = 10;
8: }
```

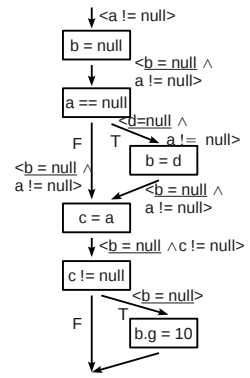


Figure 1. Computation of weakest at-least once pre-condition at line 1 for condition "b $= null$" at line 7

1. Introduction

We address the problem of verifying the non-nullness of dereferences in Java programs, via over-approximated weakest pre-conditions analysis. Given a program point p and a condition C, a *pre-condition* [5] for (p, C) is a constraint on the initial state of the program that guarantees that the program state will satisfy C every time control reaches the point p; control, however, will not necessarily reach p in every execution that begins by satisfying the pre-condition. A *weakest pre-condition* $wp(p, C)$ is the weakest such pre-condition for (p, C). Clearly, a property C at point p always holds iff $wp(p, C)$ is logically equivalent to $true$ (we assume that there are no external constraints on the possible initial states of a program). We define a notion of the *weakest at-least once pre-condition*, denoted $wp_1(p, C)$, as the weakest constraint on the initial state of the program that guarantees that execution will reach p at least once in a state that satisfies C; control, however, may reach p other times in states that do not satisfy C. Note: (a) for any (p, C), $wp_1(p, C) = \neg wp(p, \neg C)$, and (b) a property A always holds at a program point p iff $wp_1(p, \neg A)$ is logically equivalent to $false$.

Weakest at-least-once preconditions are not computable precisely in the presence of loops and recursion. Our approach is to use an *abstract interpretation* [4] to over-

approximate the weakest at-least-once pre-condition; i.e., for variable v dereferenced at a given program point p, we use a backwards, demand-driven dataflow analysis to compute a condition at the entry point to the program that is equal to or implied by $wp_1(p,\ v = null)$. We declare the dereference safe if this condition is equal to $false$. Our data-flow lattice is a lattice of predicates, whose literals are access paths (of the form $v.f_1.f_2 \ldots f_k$), and $null$. The lattice elements are ordered by implication, where weaker predicates dominate strong predicates, while our *join* operation is OR. We illustrate our analysis using a small example in Figure 1; as can be seen, the weakest at-least-once pre-condition at line 1 for `b` to be $null$ in line 7 is "`a` $\neq null$". In our implementation, in order to ensure termination, and for the sake of scalability, we ensure finite access path lengths, abstract out arithmetic, use summary tables [18] to cache the results of inter-procedural analysis, and use custom predicate-simplification rules rather than a full-blown satisfiability solver. For the sake of precision our analysis is inter-procedurally context-sensitive, keeps track of aliasing relationships precisely, and uses a limited form of path sensitivity.

1.1 Contributions

Our chief contributions are:

- A novel set of design features that together enable sound demand-driven verification of dereferences in real Java programs, with very quick response time per dereference, and reasonable precision. Our approach would be useful if integrated in a development environment, letting a programmer check the safety of the dereferences in their current scope of interest (e.g., the methods in the class they are editing) in near real time.
- An implementation of this approach.
- An evaluation of the approach on a set of real Java programs, with detailed analysis of the running time and precision. To the best of our knowledge ours is the first practical weakest pre-conditions-based verification approach to be demonstrated on large, real Java programs.

Over-approximating weakest at-least-once preconditions for Java is challenging for multiple reasons. Loops and recursion are difficult to handle soundly and with reasonable precision. Even for loop-free programs there is potential for exponential explosion due to the number of distinct paths in the program. Within straight-line code itself aliasing can increase the number of disjunctions in the preconditions. In addition to these issues, the computation should handle language features like virtual method dispatch, type checks, frequent accesses to heap, deeply nested chains of method calls, and call backs (from library methods to application methods). All of these, and also the sheer scale of real Java programs can make the analysis extremely expensive or very imprecise unless it is designed carefully. Therefore, we make a set of design decisions for our analysis (outlined above, and described in detail in the subsequent sections) that yield good precision in many (but not all) scenarios, without being impractical expense-wise.

Compared to previous related work on this topic, e.g., Xylem [15] and Salsa [13], our technical innovations are in terms of how we (a) perform *strong updates* (for precision) in the presence of aliasing, (b) handle recursion soundly in the backwards, demand-driven setting, and (c) enable more paths, and longer paths with deep call chains to be explored, by continually simplifying the formulas being propagated. Our formula simplification is based on dropping conjuncts that are less likely to eventually play a role in the validation or invalidation of the formula, and is with the intent of keeping formula sizes in check, which could otherwise explode with increasing path lengths. As a result of these innovations we are able to efficiently analyze large program scopes, thereby refuting a conjecture by previous researchers [13] that such an analysis would be infeasible. In fact, the primary strength of our approach is being able to verify a class of dereferences where the pointer value is not transmitted through recursive data structures or arrays (which are expensive to model precisely, and which we deal with conservatively), but are nonetheless difficult for other approaches to reason about due to the large number of paths, and long paths, that go from the program entry to the dereference.

Ours is a verification approach to efficiently find a superset of all potential null-dereference sites; this is in contrast with *bug-finding* approaches [11, 15], that may miss bugs (and also report false positives). Our approach complements precise, expensive, and incomplete approaches such as Snugglebug [3] that attempt to find a concrete input to a program that *disproves* a desired safety property, in that these approaches could be used to try to validate our bug reports.

We have evaluated our implementation on a set of ten medium-to-large real programs. The approach turns out to be highly efficient, taking less than 250 milli seconds per dereference on at least 93% of the dereferences in each of 9 (out of 10) programs, and on 85% of the dereferences in one program. The memory footprint of the analysis is very small. It is reasonably precise, reporting fewer than 16% of the dereferences as potentially unsafe in 7 out of the 10 programs, and between 21% and 29% of the dereferences as potentially unsafe in the remaining 3 programs. We have also identified various interesting characteristics of the dereferences that were verified as safe, such as lengths and context-depths of analyzed paths, as well characteristics of dereferences that were found to be potentially unsafe.

An advantage of our approach is that it is demand driven, in contrast to previous verification approaches that are based on a forward analysis, such as Salsa [13]. A demand-driven analysis means that an individual programmer working on a portion of a large application could try to verify dereferences in his or her code, by taking into account all paths in

AccessPath (AP)	$\rightarrow$	*Variable.Fields* | *Variable*
Fields	$\rightarrow$	*field.Fields* | ϵ
Atom	$\rightarrow$	*AP* | $null$
Predicate	$\rightarrow$	*Atom op Atom* | $true$ | $false$
op	$\rightarrow$	$= \mid \neq$
Disjunct	$\equiv$	$2^{Predicate}$
Formula	$\equiv$	$2^{Disjunct}$

Figure 2. Structure of formulas (lattice elements)

the program that lead to these dereferences, without paying the price of verifying *all* dereferences in the program. Our analysis is currently highly efficient for this purpose. In the future, we will look into trading off some of this efficiency to address more precisely certain complex features such as arrays and recursive heap data structures.

The rest of this paper is structured as follows. We discuss our analysis in the intra-procedural setting in Section 2, and its extension to the inter-procedural setting in Section 3. Section 4 contains a few details about our implementation, while Section 5 discusses our experiments in detail. We discuss related work in Section 6, and conclude the main paper in Section 7. Finally, we sketch a formulation of our analysis as an abstract interpretation in the Appendix.

2. The basic intra-procedural approach

As mentioned in the Introduction, our approach is a standard backwards data-flow analysis. In the rest of this section we discuss in the intra-procedural setting our abstract lattice, transfer functions, the analysis, as well as the key features and properties of this analysis.

2.1 Abstract lattice, and transfer functions

Our abstract lattice elements are formulas in disjunctive normal form, represented as shown in Figure 2. Note that a *Disjunct* is a conjunction of *Predicates*, and a *Formula* (i.e., a lattice element) is a disjunction of *Disjuncts*. The lattice elements are ordered intuitively by implication: in particular, given two *Formulas* f_1 and f_2, $f_1 \leq f_2$ if $f_1 \subseteq f_2$ (considering each Formula as a set of Disjuncts). Therefore, our join operation is set union (which implements logical OR), bottom element is the empty set of disjuncts (which represents logical falsehood), and the top element is the set of all Disjuncts (which represents logical truth). While the lattice as shown is of infinite size due to unbounded sequences of fields in access paths, in the analysis we effectively make the lattice finite by bounding the access paths (see details in Section 2.1.1).

We assume that the program is in an Intermediate Representation (IR) form like three-address code. We also assume that variables have been renamed in a way that the name of a variable fully identifies the scope it is declared in. The (backward) transfer functions for the individual statements in the IR are shown in Figure 3. Since the functions are all *distributive*, we express each one as taking a single disjunct in the statement's post-state as input, and returning a set (i.e., disjunction) of disjuncts ϕ' in the statement's pre-state.

Our notation and terminology is as follows. For a formula ϕ and variables v, w, $\phi[w/v]$ means ϕ in which v is replaced syntactically with w. $Vars(pred)$ is a set containing the (one or two) program variables (without the field names) referred to in $pred$. $SubAPs(\phi)$ is a set containing all prefixes (proper as well as improper) of all access paths that are operands of the predicates in ϕ. The symbol $=_s$ denotes syntactic identity of two terms (as opposed to value identity). Note that the PUTFIELD rule (in its second of three cases) is the only one to return a ϕ' containing two disjuncts. All other rules return a ϕ' containing a single disjunct; therefore, for brevity, we omit the curly braces around ϕ' in these rules. The *root dereference* is the given dereference that we wish to verify; e.g., in Figure 1 it is the dereference of b at line 7. Every disjunct in the analysis has zero or one *root predicates*, which are always of the form $AP = null$. At the program point preceding the root dereference of an access path ap the sole disjunct (i.e., the post-condition) contains a single predicate $ap = null$, which is the root predicate of this disjunct. Whenever a disjunct d with root predicate p_d gets transformed by a statement into a disjunct d', the root predicate in d' is that predicate that is equal to p_d or the result of the rewriting applied to p_d by the transfer function of the statement. In our notation we underline the root predicate in each disjunct; for instance, see Figure 1, where the disjunct at the start of the analysis (i.e., the post-condition) is "b = $null$" at the point before Line 7.

The objective of the transfer function of each statement is to accept a post-state ϕ, and return a pre-state ϕ' that's an over-approximation of the weakest at-least-once precondition of ϕ wrt the statement. The functions for COPY, NULLASGN, and RETURN are self-explanatory; we discuss below some of the more interesting ones. The EXPRASGN rule reduces all predicates in ϕ that depend on v to $true$, because we abstract out all arithmetic from our disjuncts. Note that reducing a predicate to $true$ is equivalent to removing (or *dropping*) the predicate from the disjunct. In fact, an empty disjunct (disjunct with no predicates) is equivalent to $\{true\}$. GETARRAY does a similar reduction, while PUTARRAY has an identity transfer function, because the incoming fact (i.e., Formula) can contain no array references. NEWASGN uses an approach that is standard in many previous approaches – of representing all objects allocated at an allocation-site i by a single variable t_i (that is not present in the original program).

2.1.1 Bounding access paths, and path sensitivity

We finitize our lattice by bounding access path lengths, as follows. In rule GETFIELD, whenever a predicate in the computed pre-condition ϕ' contains an access path in which some field f repeats more than once in the sequence of fields, we *drop* this predicate (i.e., reduce it to $true$). This

	Instruction	Transfer Function: $\lambda\phi \in Disjunct.\phi'$, where $\phi' \in 2^{Disjunct}$, and is =
COPY	$v = w$	$\phi[w/v]$
NULLASGN	$v = null$	$\phi[null/v]$
NEWASGN	$v = new\ T$	$\phi[t_i/v]$, where t_i is a variable representing all objects allocated at this instruction i
GETFIELD	$v = r.f$	$\phi[r.f/v] \cup \{r \neq null\}$ (*See Section 2.1.1 regarding access-path bounding.*)
ASSUME	$assume(b)$	$\phi \cup \{b\}$, if $b =_s$ "$AP\ op\ null$" ϕ, otherwise
EXPRASGN	$v = v_1\ op\ v_2$	$\phi - S$, where $S = \{pred \in \phi \mid v \in Vars(pred)\}$
GETARRAY	$v = a[i]$	$\phi - S$, where $S = \{pred \in \phi \mid v \in Vars(pred)\}$
PUTARRAY	$a[i] = v$	ϕ
PUTFIELD	$r.f = v$	$\phi[r.f, v, ap_1.f][r.f, v, ap_2.f] \ldots [r.f, v, ap_n.f]$, where $\{ap_1.f, ap_2.f, \ldots, ap_n.f\}$ are the access paths in $SubAPs(\phi)$ that end with field f, and $\phi[r.f, v, ap_i.f]$ $= \phi[v/ap_i.f] \cup \{r \neq null\}$, if $MustAlias(r, ap_i)$ after $r.f = v$ $= \{\phi[v/ap_i.f] \cup \{r = ap_i, r \neq null\},\ \phi \cup \{r \neq ap_i, r \neq null\}\}$, if $MayAlias(r, ap_i)$ after $r.f = v$ $= \phi \cup \{r \neq null\}$, otherwise
RETURN	$return\ v$	$\phi[v/ret]$, where ret is a place-holder for the return value

Figure 3. Abstract transfer functions

effectively finitizes our lattice, because no access path can be longer than the total number of distinct fields (statically) in the program. This does render our analysis imprecise in the presence of recursive data structures; we postpone further discussion of this to Section 4.2.

We implement a limited notion of path-sensitivity as follows. The ASSUME rule conjuncts the condition b in $assume(b)$ with ϕ to yield ϕ' if b is an access path being compared to null; this predicate (i.e., b) could get validated or invalidated later during the propagation, based on other assignments or assumes that are encountered.

2.1.2 Strong updates in the presence of aliasing

Strong updates of fields are a requirement for precision in null-dereference analysis, but are difficult to perform in the presence of aliasing. Even precise pre-computed may-alias and must-alias information may not enable strong updates often enough, because at a given program point two variables may be aliased under some paths and not aliased under other paths. Since our approach is to propagate disjuncts selectively along paths (depending on the predicates in the disjuncts), we basically incorporate a flow- and context-sensitive and limited path-sensitive points-to analysis within our main analysis, and always perform strong updates at put-field statements. Given a put-field statement $r.f = v$, and for each access path ap_i that occurs in ϕ, we (a) hypothesize that ap_i and r are aliases, and generate a disjunct in ϕ' in which occurrences of $ap_i.f$ have been replaced with v, and also (b) hypothesize that ap_i and r are *not* aliases, and generate a disjunct in ϕ' that is identical to ϕ. To the first of the two disjuncts above we add an *aliasing predicate* $r = ap_i$, and to the second one we add the aliasing predicate $r \neq ap_i$. The aliasing predicate(s) in a disjunct encode the aliasing hypotheses under which the disjunct is valid, and are used by the analysis to subsequently validate (resp. invalidate) the disjunct as it gets propagated to statements that confirm (resp. contradict) the hypotheses. In contrast, previous approaches such as Xylem [15] and Salsa [13] are not able to take advantage of path-sensitive aliasing relationships as deeply as we do.

```
v = w;      <x.g = null>
w.f = x;    <x.g = null ∧ v = w>, <v.f.g = null ∧ v ≠ w>
v.f.g.h;    <v.f.g = null>
```

Figure 4. Illustration of our bottom-up analysis in the presence of PUTFIELD statement. Each entry on the right-hand side shows the formula at the program point that precedes the corresponding statement. Root predicates are underlined.

Note that in the PUTFIELD rule we do make use of pre-computed *MayAlias* and *MustAlias* information (if available). This is simply an optimization for efficiency; the precision of the pre-computed alias information does not influence the ultimate precision of our analysis (in the intra-procedural setting). For instance, if there is no pre-computed may-alias information available then every access path can be assumed to be may-aliased with every other access path (provided their static types are compatible); similarly, if there is no pre-computed must-alias information available (which is the case in our implementation), we can treat ap_i and ap_j to be must-aliased iff $ap_i =_s ap_j$.

We illustrate the above technique using Figure 4. The left hand side shows a toy program, wherein we wish to verify the dereference of the field "`v.f.g`". To the right of each statement we show the disjuncts at the point *above* the

statement. Propagating the disjunct "v.f.g = $null$" from the final point upwards, we encounter the put-field statement, which leads to the generation of two disjuncts. Both these disjuncts are propagated upwards, but the second one gets invalidated by the statement "v = w", meaning the first one alone reaches the top (and hence is the weakest at-least-once pre-condition).

Given a put-field statement $r.f = v$, if there are k prefixes of access paths of the form $ap.f$ in ϕ such that ap may be aliased with r, the resultant ϕ' will have 2^k disjuncts. While this blowup sounds excessive on paper, it is not a problem in practice, due to several reasons. One reason for the blow up, in which case the blowup gets compensated for very soon after it happens, is that many IRs, such as the one we use – Wala [21] – always copy program variables into temporaries before doing field accesses. Thus, the put-field statement above becomes something like "$t_k = r;\ t_k.f = v$". While propagating a disjunct that refers to r in the backwards direction the analysis would not know initially if t_k and r are aliased or not. Thus, it would blow up the disjunct. Then, upon encountering $t_k = r$ it would invalidate the disjunct that has the aliasing predicate $t_k \neq r$.

When programmers introduce aliasing in their code, then invalidation may not happen so quickly, and the number as well as sizes of disjuncts could increase inordinately when propagated through long inter-procedural paths (although, due to the finitized lattice, there is no danger of non-termination). To deal with this, we associate an *age* with every predicate, which is the number of statements it has been propagated through; we have a threshold k_1 (which we set to 1000 in our implementation), and drop (i.e., reduce to $true$) a predicate whenever its age increases beyond k_1. Additionally, we have another threshold k_2 (set to 3 in our implementation): whenever a disjunct has more than k_2 predicates, excluding aliasing predicates, we drop its oldest of its predicates. The idea behind dropping old predicates is our observation that branch correlations in paths typically occur between branches that are near each other in the code, than very far from each other. The root predicate alone is never dropped using these heuristics. Note that these heuristics are conservative; in fact, dropping any predicate at any point only results in a weaker pre-condition than the ideal one.

Xylem also bounds disjunct sizes, but in a more coarse-grained manner. They drop an entire disjunct when its size crosses a threshold. We, on the other hand, drop individual predicates from disjuncts, and that too "old" ones, which are least likely to get validated or invalidated going forward. We found that our approach serves the purpose of improving efficiency, but with greater precision.

2.2 Simplification rules

Inspired by the Snugglebug [3] approach, we apply a light-weight custom simplifier on each disjunct after it is produced by a propagation step, to validate, invalidate, or simplify the disjunct. Figure 5 shows a sampling of the rules used in our simplifier. In Figure 5, ap_i's represent access paths, and each t_i represents the special variable introduced corresponding to the allocation-site i by the analysis to represent all concrete objects allocated at this site (see the NEWASGN rule in Figure 3). Rules 1-3 are straightforward. Rules 4 and 5 are based on the fact that the sets of objects allocated at two different sites are disjoint. Rules 6 and 7 are based on the fact that a newly allocated object cannot be null (we do not model the potential failure of memory allocation). Rules 8 and 9 are based on the fact that a newly allocated object is not the same as any other existing object.

(1)	$(ap = ap)$	$\longrightarrow$	$true$
(2)	$\{ap_1 = ap_2, ap_1 \neq ap_2\}$	$\longrightarrow$	$\{false\}$
(3)	$\{ap_1 = null, ap_1 \neq null\}$	$\longrightarrow$	$\{false\}$
(4)	$(t_i = t_j)$	$\longrightarrow$	$false$
(5)	$(t_i \neq t_j)$	$\longrightarrow$	$true$
(6)	$(t_i = null)$	$\longrightarrow$	$false$
(7)	$(t_i \neq null)$	$\longrightarrow$	$true$
(8)	$(t_i = ap)$	$\longrightarrow$	$false$
(9)	$(t_i \neq ap)$	$\longrightarrow$	$true$

Figure 5. Rules for simplifying disjuncts

Note that for any statement *st*, its abstract transfer function f_{st} actually is the function shown in Figure 3, composed with repeated applications of the simplification rules until a fix-point is reached.

2.3 Putting it all together

Let ap be a given root dereference, i.e., a given access path that is dereferenced at a program point p that we wish to verify. We start the dataflow analysis with the fact (i.e., formula) at program point p set to $C \equiv ap = null$, and facts at all other program points set to the empty set. When the analysis terminates, we read off the fact at the program entry as an over-approximation of $wp_1(p, C)$. If this over-approximation is the empty set (i.e., logical falsehood) then the root dereference is safe.

Since the transfer functions are distributive, we mark and propagate individual disjuncts, not formulas (sets of disjuncts), using Kildall's [12] algorithm. This greatly reduces the time to reach a fix point. We have also implemented a few optimizations to the propagation: (a) Whenever any disjunct anywhere gets simplified to the empty set (i.e., logical truth) we terminate the analysis right away and call the root dereference unsafe, without waiting for a fix point to be reached, and (b) whenever a disjunct gets simplified to $\{false\}$ we unmark it, and do not propagate it any more. It would be possible to take a slice of the program starting from the root dereference, and do our analysis only on the slice. The result would be the same as performing the analysis on the whole program. However, we do not do this, because we found that

the time spent on computing the slice was not compensated by the time saved during the subsequent analysis.

Since we don't model array references in our abstract lattice, we don't analyze root dereferences that have array references in them (we call them unsafe by default). As with previous techniques [13, 15], our technique does not model concurrency soundly, nor dynamic features such as reflection and dynamic class loading. Our approach may miss null-dereference errors that manifest themselves due to these aspects.

See the right-side of Figure 1 for an illustration of our approach. The root dereference is b.g; hence we start the analysis with the singleton disjunct $b = null$ at the point above this dereference, and the empty set all other points. The disjuncts at each program point after the propagation is over are shown in the figure (we omit false disjuncts at the program points before Lines 1 and 2). Note the path sensitivity in this analysis: the disjunct $d = null \wedge a \neq null$ at the $true$ branch of the first conditional (in line 3) becomes false (and is not shown) after it propagates up through the conditional. The end result is that the dereference in line 7 may be unsafe if $a \neq null$ at the entrance to the program; in this example, this turns out to be the precise weakest at-least-once pre-condition, and not an over-approximation.

3. Inter-procedural analysis

Our inter-procedural analysis algorithm is based on the one used in Xylem. Their analysis, in turn, is based on Sharir and Pnueli's tabulation based approach to inter-procedural analysis [18], with an important modification: when a call-site is encountered, the analysis proceeds to analyze the callee (together with all its transitive callees) to completion before proceeding with the analysis of the caller. In other words, a depth-first propagation strategy is followed, rather than a chaotic strategy or a breadth-first strategy (we discuss later the advantages offered by this strategy). Xylem's algorithm is sound and fully context-sensitive in the absence of recursion, but is unsound in the presence of recursion (it skips recursive calls). We first discuss our basic approach (ignoring recursion) below, and then discuss in Section 3.1 how we handle recursion by iterating for a fix-point.

Figure 6 shows our inter-procedural analysis procedure. The arguments to this procedure are a method m, a statement *stmt* in m, and a post-condition ϕ_{stmt}. Let p be the program point after statement *stmt*. The return value from this procedure is a set of disjuncts, which represents an over-approximation of $wp_1(p, \phi_{stmt})$ at the entry of method m.

The procedure shown in Figure 6 uses two globally scoped data structures viz. a stack CS and a summary table Σ. The stack CS, as well as lines 8–9, 15–16, and 36–40 in the figure are about handling recursion, and are discussed in Section 3.1. We maintain for each method m a summary table $\Sigma[\![m]\!] : Disjunct \mapsto \mathcal{P}(Disjunct)$, which is a partial map from a disjunct at the exit of m to the set of disjuncts that would result at the entry of m after propagating the disjunct through the method (and its transitive callees). The algorithm uses a worklist W to propagate a disjunct

```
1: Procedure wpWrtMethod(m, stmt, φ_stmt)
2: worklist W = {(stmt, φ_stmt)}
3: result = ∅ {A set of disjuncts}
4: if stmt = exit(m) then
5:   if Σ[[m]](φ_stmt) is not defined then
6:     update Σ[[m]][φ_stmt ↦ emptyset]
7:   end if
8:   push(CS,(m, φ_stmt))
9:   Γ_m = Σ {Γ_m is a snapshot of summary table Σ at the
     entry of m}
10: end if
11: while W ≠ ∅ do
12:   Select (S, φ) ∈ W {φ is a post-cond. after stmt S}
13:   output = ∅ {A set of disjuncts}
14:   if S is a call instruction v_r = c(v_1, v_2, ..., v_n) then
15:     if (c, φ) ∈ CS then
16:       output = Σ[[c]](φ)
17:     else
18:       if Σ[[c]](φ) is defined then
19:         output = Σ[[c]](φ) {Summary hit}
20:       else {Summary miss}
21:         output = wpWrtMethod(c, exit(c), φ)
22:       end if
23:     end if
24:   else if S is Entry then
25:     Add φ to result
26:   else
27:     output = f_S(φ)
28:   end if
29:   for all Predecessor Statement S_P of S do
30:     for all φ' ∈ output do
31:       W = W ∪ {(S_P, φ')}
32:     end for
33:   end for
34: end while
35: if stmt = exit(m) then
36:   pop(CS)
37:   if (Σ[[m]](φ_stmt) ≠ result) then
38:     Σ = Γ_m {replace the summary table by the saved
       snapshot}
39:     update Σ[[m]][φ_stmt ↦ result]
40:     result = wpWrtMethod(m, stmt, φ_stmt)
41:   else
42:     update Σ[[m]][φ_stmt ↦ result]
43:   end if
44: end if
45: return result
```

Figure 6. Computing $wp_1(stmt, \phi_{stmt})$ at the entry of method m for post-cond. ϕ_{stmt} at point after $stmt$ in m.

```
1: Procedure wpWrtPgm(m, stmt, φ_stmt)
2: propagated = propagated ∪ (stmt, φ_stmt)
3: result = wpWrtMethod(m, stmt, φ_stmt)
4: precond = ∅
5: if ∃ a predecessor for m then
6:    for all disjunct d' in result do
7:       for all callsite s' of m in predecessor p of m do
8:          if (s', d') ∉ propagated then
9:             precond = precond ∪ wpWrtPgm(p, s'', d'),
10:            where s'' is the statement that precedes s'
11:         end if
12:      end for
13:   end for
14: else
15:   precond = result
16: end if
17: return precond

18: Procedure analyzeDeref(m, stmt, ap)
19: set CS, propagated to empty
20: set all entries in Σ to undefined
21: result = wpWrtPgm(m, stmt, ap = null)
22: if result is the empty set then
23:   return "safe"
24: else
25:   return "potentially unsafe"
26: end if
```

Figure 7. $wpWrtPgm$: Computing $wp_1(stmt, \phi_{stmt})$ at the entry of the program for post-condition ϕ_{stmt} just after statement $stmt$ in m. $analyzeDeref$: Computing $wp_1(stmt, ap = null)$ at the entry of the program, where ap is an access path in $stmt$.

intra-procedurally through the statements in the method to the entry of the method. Each element in the worklist is a pair (S, ϕ), where ϕ is post-condition at the the program point after the statement S.

The lines 2–7 in Figure 6 performs the initialization of the worklist, summary table and a variable $result$ that is used to store the precondition computed at the entry of the method. As shown in lines 11–34, the algorithm iteratively processes the elements in the worklist. For each element (S, ϕ) in the worklist, where S is a statement and ϕ is a post-condition after S, the algorithm computes the pre-condition (as a set of disjuncts) at the point before the statement S and stores it in a variable $output$ (lines 13–28). For every statement S other than a call statement and a method entry statement, the intra-procedural transfer function f_S as defined in Section 2.2 is used to compute the pre-condition before the statement S (line 27). If S is a call statement of the form $v_r = c(v_1, \ldots, v_n)$, the procedure *wpWrtMethod* is recursively invoked (at line 21) to process the callee c with the post-condition ϕ only if there is no result in the summary table for (c, ϕ) (this check is done in line 18). Note that when we analyze a callee (in line 21) we need to map (i.e., replace) the occurrences of the actual parameters (at the call-site) in the access paths in ϕ with the corresponding formal parameters, and similarly unmap the access paths in the condition that was returned from the callee analysis (i.e., in $output$). This mapping and unmapping is straightforward, so we omit the details. The pre-condition $output$ computed before the statement S is the post-condition for each predecessor S_P of S. Hence, for each disjunct $\phi' \in output$ (S_P, ϕ') is added to the worklist W so that it would be processed in the subsequent iterations (this is done in lines 29–33). Finally, once the pre-condition $result$ is computed at the entry of the method, the summary table Σ is updated to map the input post-condition ϕ_{stmt} to the precondition $result$ (at line 42) provided the input post-condition was for the $exit$ statement of the method m (this check is done at line 35).

Procedure *wpWrtPgm* in Figure 7 has the same signature as procedure *wpWrtMethod*, but computes the weakest at-least-once pre-condition of ϕ_{stmt} (at the point after $stmt$ in method m) at the entry of the entire program, rather than at the entry of m. It does so by first computing the pre-condition at the entry of m (see call to *wpWrtMethod* in line 3), and then propagating the disjuncts in this pre-condition through all predecessors (i.e., transitive callers) of m until the disjuncts reach the entry of the program (see tail-recursive call in *wpWrtPgm* to itself in line 9). The variable $propagated$ is a global variable, and is used to remember the facts that were propagated up to the various call-sites, in order to ensure termination. Procedure *analyzeDeref* is the main routine to analyze the safety of the dereference of a given access path *ap* at a point after statement *stmt* of method m. It is basically a wrapper around procedure *wpWrtPgm*.

The depth-first approach described above (which was followed in Xylem, too) has the advantage that if a disjunct becomes *validated*, i.e., becomes true at some point or reaches the program's entry while being satisfiable, then we terminate the analysis (and call the root dereference unsafe). If we adopted a chaotic or breadth-first approach it would potentially take much more analysis time before any single disjunct is propagated deep enough to become validated. Another key advantage of our method is that it is space efficient. We construct a control-flow graph of a method only on demand, i.e., when we encounter a call to the method. Also, once we have analyzed this method (which is a callee), the memory allocated to it for storing the data-flow facts (or disjuncts) at every program point, the intra-procedural worklist, etc., can be freed up. (However, as in the tabulation based approach, we re-analyze a method when it is re-encountered with a different data-flow fact at the exit.) Similarly, in procedure *wpWrtPgm*, we can de-allocate memory used for analyzing any method m before we go onto analyzing its predecessors.

$\Sigma[\![\text{f}]\!](b = null)$ =				
	false	$b = null$	"$b = null \vee a = null$"	"$b = null \vee a = null$"
1:	f(b,a) {			
2:	if (*) {	$b = null$	"$b = null \vee a = null$"	"$b = null \vee a = null$"
3:	b = a;	false	$a = null$	$a = null$
4:	f(b,a);	false	$b = null$	"$b = null \vee a = null$"
5:	}	$b = null$	$b = null$	$b = null$
6:	}	$b = null$	$b = null$	$b = null$
	Iteration:	1	2	3

Figure 8. Bottom-up analysis of recursive method f with post-condition $b = null$, done repeatedly until fix-point. Each iteration i uses summary table computed in previous iteration (see top row in iteration $i - 1$) to analyze call at line 4. Each entry shows formula at the program point that precedes the corresponding statement.

3.1 Handling recursion by iterating for a fix-point

We identify recursive calls during the analysis using a context-stack CS, whose entries are pairs of the form (c, ϕ); the presence of such an entry in the stack means that an analysis of method c with post-condition ϕ (at the method's exit) is ongoing and not yet completed. Given this, if we encounter another call to c with the same post-condition ϕ, as checked for in line 15 in Figure 6, we have detected recursion; we then pick the (potentially intermediate non-fix-point) result $\Sigma[\![c]\!][\phi]$ from the summary table (instead of starting a re-analysis of c), complete the analysis of the caller (i.e., method m, with post-state ϕ_{stmt}). Eventually, we complete the analysis of the recursive method c using the intermediate non-fix-point result of c. At this point we restart the analysis of c with the same post-condition (see line 40). However, during the reanalysis we discard the intermediate non-fix-point summary entries created during the previous analysis. For this purpose, we use a temporary place holder variable Γ_m. Each time we analyse a method c, we store a snapshot of the summary table at the entry of c in Γ_m (line 9) and during the reanalysis, we replace the summary table by the cached snapshot thereby invalidating all the non-fix-point summary entries created during the previous iteration (see line 38). In essence, if during the analysis of a method m with post-condition ϕ_{stmt} we detect a recursive call, we iteratively analyze m with the same post-condition until $\Sigma[\![m]\!](\phi_{stmt})$ reaches a fix-point. During every iteration $i \geq 1$, we use $\Sigma[\![m]\!](\phi_{stmt})$ computed in the iteration $i - 1$, which we denote here as $\Sigma[\![m]\!]^{i-1}(\phi_{stmt})$, to compute $\Sigma[\![m]\!]^{i}(\phi_{stmt})$ ($\Sigma[\![m]\!]^{0}(\phi_{stmt})$ being false). We repeat this process until $\Sigma[\![m]\!]^{i-1}(\phi_{stmt}) = \Sigma[\![m]\!]^{i}(\phi_{stmt})$ for some $i \geq 1$.

We illustrate the analysis of the recursive method f in Figure 8 with post-condition $b = null$ at the exit of the method. In each iteration of the analysis, to the right of each statement of code, we show the formula computed at the point just above the statement. Consider Iteration 1, where we propagate $b = null$ upwards from the exit. When we encounter the recursive call in line 4, we pick up the value of $\Sigma[\![\text{f}]\!](b = null)$ from the initial summary table, which is false (see the top of the figure, above the program). Therefore, the disjunct that reaches the top of the method is false, via the *true* branch of the "if", and $b = null$ (the post-condition at the end of the method), via the *false* branch. The disjunction of these two facts is $b = null$. Therefore, we update $\Sigma[\![\text{f}]\!](b = null)$ to $b = null$ (as shown at the top of the figure along Iteration 1), and start Iteration 2 from the bottom. This time, at line 4, since the post-state is the same ($b = null$), we pick up the value $b = null$ from $\Sigma[\![\text{f}]\!](b = null)$ (as updated in the previous iteration). This gets transformed to $a = null$ after propagation through "b = a;". Therefore, the fact at the beginning of the method becomes $b = null \ \vee \ a = null$, where $b = null$ came, as explained before, via the *false* branch. This fact is updated into the summary table (as shown at the top of the figure, along Iteration 2), and is looked up and used at line 4 in Iteration 3. When this fact flows through "b = a" it becomes $a = null \ \vee \ a = null$, which simplifies to $a = null$. Therefore, a fix point is reached. The finally computed weakest at-once pre-condition at the beginning of method f for the post-condition $b = null$ is $b = null \ \vee \ a = null$, which, in this case, turns out be the precise solution.

3.2 Optimizations

Using a precomputed side-effects analysis. During the analysis of a method m if we encounter a method call c with a disjunct ϕ as the post-state, we propagate the disjunct ϕ to the entry of c (if the summary for ϕ is not available). However, if the information about the set of all access-paths in ϕ that may be modified by c (after mapping) is available then we can partition the disjunct ϕ into $\phi_1 \wedge \phi_2$ such that none of the access-paths in ϕ_2 (or their aliases) are modified by c. By the frame rule of the separation logic [16], $wp_1(m, \phi) = wp_1(m, \phi_1) \wedge \phi_2$. Hence, an over-approximation of $wp_1(m, \phi)$ could be computed by over-approximating $wp_1(m, \phi_1)$ and conjuncting each resulting disjunct it with ϕ_2. We consider all the access-paths that uses the return value of c as modified by c. This approach has two advantages, (a) it reduces the amount of work that has to be done inside the method m (and all its transitive callees) (b) it increases the summary hits.

In our implementation, we used an inexpensive *ModRef* (or side-effect) analysis that was available in our program analysis framework, Wala.

Increasing summary hits by reusing summaries for weaker disjuncts. In cases where we do not have a mapping for a disjunct ϕ_1 in the summary table, but have a mapping for a disjunct ϕ_2 such that $\phi_1 \supseteq \phi_2$ (i.e., ϕ_1 im-

plies ϕ_2), instead of analyzing the method m with ϕ_1 as post-condition we simply pick up and use $\Sigma[\![m]\!](\phi_2)$. Since $wp_1(m, \phi_1) \Rightarrow wp_1(m, \phi_2))$, if ϕ_1 implies ϕ_2 then any over-approximation of $wp_1(m, \phi_2)$ is a correct over-approximation of $wp_1(m, \phi_1)$. However, in order to minimize the loss of precision that this may entail, we use this heuristic only if the root predicate in ϕ_1 is the same as the root predicate in ϕ_2. We found this approach pragmatically effective.

4. Implementation details

4.1 Analysis framework

We have implemented our approach using the Wala [21] program analysis framework. Wala provides us control-flow graphs of methods on demand, as well as points-to (i.e., may-alias) information (which it pre-computes). From among the various points-to analysis methods Wala offers, we selected a flow-insensitive, partially context-sensitive method (namely, `ReceiverTypeContextSelector`). Our approach uses Wala's points-to analysis results for three purposes: (a) to construct a call-graph (particularly, to resolve virtual method calls), (b) to compute Mod-Ref sets (i.e., side-effect information) for methods, (c) in the PUT-FIELD rule, to reduce the number of aliasing combinations that have to be considered (see Figure 3). Imprecision in Wala's points-to analysis affects the precision of our approach due to points (a) and (b) above, and affects the scalability of our approach due to all three points above. Note that point (c) does not affect our precision, due to our precise modeling of aliasing relations (see Section 2.1.2). Therefore, our approach would significantly benefit from a more precise points-to analysis. However, due to scalability limitations of Wala's points-to analysis implementations, we chose a points-to analysis method that is reasonably precise but highly scalable.

4.2 Balancing scalability and precision

There are several idioms in real-world Java programs that pose severe challenges to any technique that wishes to perform verification scalably and precisely. We explore some of these idioms, and discuss the engineering decisions we have taken wrt giving up precision in certain situations wherein a precise analysis would have turned out to be impractically expensive.

Issue 1: Call-backs from the library. In Java, several methods like *equals*, *hashCode*, *toString* defined in the application classes are invoked by library methods; such calls are generally referred to as *call backs*. For e.g., *HashSet::Contains* invokes the *equals* method on the elements of a hash set, which could be objects of application classes. Given a root dereference inside a called-back method, such as *equals*, it is often very expensive to propagate the disjuncts that reach the entry of this method back through all its caller chains (via library code) to the entry of the program, as there may be numerous (transitive) call sites for this methods inside the library and in the application. In fact, going through all paths back from called-back methods would result in many spurious paths; e.g., a path back from method *A::Equals* through *HashSet::Contains* can only reach points in the application where *HashSet::Contains* is being called on a hash-set that contains objects of type *A*. Rather than analyze all these paths, which increases running time without giving much precision gain, we have chosen to always drop (i.e., reduce to $true$) a disjunct that needs to be propagated back from the entry of a called-back method to a call-site to the same method that is inside a library method[1]. By "called-back method", we mean a method that is not on the context stack when the analysis leaves the method, which implies that the root dereference is contained in the method or in one of its transitive callees.

Issue 2: Unbounded access paths, and arrays. In the presence of recursive data-structures the length of an access path (used in a predicate) can become unbounded, leading to non-termination. In our analysis, we ensure that the fields in an access path do not repeat (see Section 2.1.1). Whenever an access-path in a predicate violates this property, we drop the predicate (i.e., reduce it to $true$). A commonly mentioned alternative is to forcibly bound the lengths of access paths using *k-limiting* [13, 14]. The problem with this approach is that two syntactically identical k-limited access-paths ap_1 and ap_2 (at a programs point) need not necessarily point to the same runtime object, and hence are not must-aliased. Therefore, strong updates become difficult to perform, impacting precision. For an illustration of this, consider a put-field statement $ap_1.f = v$, such that the post-condition ϕ after this statement involves $ap_2.f$. Even though $ap_1 =_s ap_2$, we cannot simply replace $ap_2.f$ in the ϕ with v. Rather, we would need to blow up ϕ, and produce two disjuncts, $\phi[v/ap_2.f] \wedge ap_1 = ap_2$, and $\phi \wedge ap_1 \neq ap_2$. The problem is that since ap_1 and ap_2 are k-limited, neither $ap_1 = ap_2$ nor $ap_1 \neq ap_2$ will be invalidated during further propagation. This means there is no chance of ϕ getting invalidated, implying imprecision. Inferring must-alias relationships in the presence of recursively defined data structures needs sophisticated *shape analysis* [17], which in its current state of evolution is unlikely to scale to programs of sizes that we are interested in. Hence our decision to drop predicates involving access paths with repeated fields. Note that an advantage of our approach over k-limiting is that we do allow access path lengths to grow without any apriori constant length bound, as long as fields do not repeat. Therefore, for low values of k, our approach is likely to be actually more precise than k-limiting.

Similar to predicates with unbounded access-paths, we also drop predicates containing array accesses. Since array elements are implicitly initialized to $null$ on creation, un-

[1] with one exception – when the library method is `Thread.start`

less strong updates are performed on array elements, a predicate of the form $\langle arrayap = null \rangle$, where $arrayap$ contains an array access, will eventually become $true$. There do exist techniques, e.g., that of Dillig et al [7], which precisely model integer arithmetic, and perform strong updates on array operations. However, it is not clear how such techniques can be adapted in a demand-driven setting like ours for analysing real world Java programs.

Issue 3: Too many "AP op $null$" predicates in disjuncts Note that the ASSUME, GETFIELD, and PUTFIELD rules conjunct predicates of the form "*AP op* $null$" to disjuncts, where *op* is "$=$" or "$\neq$". Along long paths the average number of these predicates per disjunct increases a lot, without contributing proportionally to improved precision. Our observation was that among the numerous conditional statements that are typically encountered along a path that a disjunct propagates through, the ones that refer to the same object as the object referred to by the root predicate of the disjunct are the ones that usually correlate with whether the disjunct gets validated or invalidated. Therefore, we limit the addition of "*AP op* $null$" predicates to the following situations: In a statement "ASSUME b" we conjunct b to the disjunct ϕ' only if the access path in b is may-aliased with the access path in the root predicate in ϕ at the point after the assume statement. In GETFIELD and PUTFIELD statements we conjunct $r \neq null$ into ϕ' only if r is may-aliased with the access path in the root predicate in ϕ at the point after the statement, where r is the variable being dereferenced in the statement.

Issue 4: Virtual method calls. As mentioned earlier, when encountering a virtual method call we query Wala's moderate-precision points-to analysis to find its possible targets. In the programs that we analyzed there do exist virtual method call sites with tens or hundreds of targets, analysing all of which is practically infeasible. Hence, when we encounter a virtual call having more targets than a predefined bound (which is 10, in our implementation), we do not analyze any of the targets. Instead, we drop all predicates that make use of the return value from the method, and also all predicates that contain access paths that are aliased with any of the access paths mutated by any of the targets of the call (as indicated by Wala's Mod-Ref information), and continue the analysis above the call. Using a more precise points-to analysis would mitigate the problem with virtual calls, but it would not scale well. The idea of *directed call-graph construction* [3] would also yield benefit in this situation, but its scalability in our context is not yet clear.

Issue 5: Missing call targets. It is possible that our analysis encounters virtual method calls that do not have any targets. This may happen if the target of a method call is defined in a class that is not available to the analysis. In most cases, these method calls are calls to the GUI and JDBC libraries, which we do not link, as it affects the scalability of Wala's points-to analysis. We treat such method calls conservatively. Predicates that use the return value of method calls that do not have a target are dropped. We also over-approximate the side-effects of such a method call by assuming that it may modify every object reachable from the arguments passed to it. We use this over-approximate side-effects information to drop predicates that may be modified by the method call.

Issue 6: Library calls. In our experiments, we found that analyzing *every* library method call adversely affects the scalability. Hence, we analyze a library method only if the value returned by the library method is used in an access-path. This is the case wherein we feel the effort of the analysis is balanced by significant improvement in precision. If a library method mutates (writes to) an access-path (or its alias) used in a predicate in the post-state (as per Wala's Mod-Ref analysis), we drop the predicate.

This said, we add two features to our approach to add back some of the precision that was given up by the above-mentioned heuristic. We manually created a list (which we refer to as the *skip-list*) of library methods that have no externally visible side effect as per their documentation, but that are nonetheless declared by Wala's imprecise Mod-Ref analysis as potentially having side effects; during the analysis, for methods in this list, we treat their Mod-Ref sets as empty. Conversely, we found that there are several library methods that are called primarily for their side-effects, as opposed to their return values. We therefore created an *analyze-list* containing such methods, and always propagate to them predicates that might be affected by them (as per the Mod-Ref information). We use these two lists (which are disjoint) not only with direct calls to library methods, but also when library methods are potential targets of virtual calls. Currently our skip-list and analyze-list contain 136 and 84 library methods, respectively. It is noteworthy that manually creating these lists, although time consuming, is worthwhile because the lists can be reused during analysis of any program. To give maximum benefit, however, it is likely that a lot more methods could be added to both these lists.

5. Experimental Results

Figure 9 shows the programs used in our empirical study. Several of these programs are commonly used to evaluate the efficiency of the program analysis tools for Java. In fact, 8 out of the 10 benchmarks used in the study were picked from the Salsa paper [13], which also proposes a null-dereference verification technique for Java programs. In Wala, to construct a call-graph for a program, one or more entry methods for the program (called *entry-points*) have to be specified. All the benchmarks reported had at least one *main* method which was used by our implementation as an entry-point for constructing the call-graph. For programs that had more than one main method we included all of them. However, we ignored the main methods in the test-suites that

Benchmark	Description
jlex 1.2.6	Lexical analyzer generator
javacup 0.1	Parser generator
ourtunes 1.3.3	ITunes browser
jbidwatcher 2.1.2	Online Auctioning system
bcel 5.2	Libraries for bytecode manipulation
antlr 3.3	Compiler & translator generator
sablecc 4.2	OO Frameworks generator
proguard 4.5	Code optimizer & obfuscator
freecol 0.9.5	multi-player game
l2j 3.7	Multi-player game server

Figure 9. Benchmarks used in the study

are distributed along with these benchmarks. We analyzed the entire portion of each program that is reachable from the entry points, including library methods called from these portions (see the discussion in Section 4.2). We carried out all our experiments using Open JDK 1.6 libraries, on a server machine having 2.27 GHz, 8 core Intel Xeon processor, 16 GB RAM, running CentOS linux operating system. Our implementation is single threaded.

Figure 10 gives the overview of our experiments. We run our backward analysis separately on each dereference in the portion of the application that was analyzed (excluding dereferences inside library methods). The second and third columns, show the total number of bytecodes in library methods and application methods, respectively, that were ever entered during the analysis of any dereference. The preprocessing time is the time taken by Wala's points-to analysis, call-graph construction and Mod-Ref analysis phases, which we perform once before we analyze any of the dereferences. The analysis time reported is the total time taken for analysing all the dereferences in the program. The next two columns show the total number of dereferences that were analyzed, and the number among them that were found potentially unsafe. The last column is the percentage of dereferences that were found safe. It is to be noted that other than the *skip list* and *analyze list* (for standard Java library methods) that we create manually, there are no other manual inputs to our tool. For the benchmarks used in the study the peak memory utilized by the analysis was in the range 371MB (for *jlex*) to 1967 MB (for *antlr*).

The results show that on an average the tool classifies about 84% of the dereferences as safe and the remaining 16% of dereferences as unsafe. For seven out of ten programs, less than 16% of the dereferences were reported unsafe, while for the remaining three programs (viz. *javacup*, *freecol*, *antlr*) between 20-30% of the dereferences were declared unsafe.

In order to put our numbers in context, we did a quick search on the bugzillas of the 10 programs we analyzed; this revealed a total of 412 null dereference reports for the program l2j, and a total of 67 reports for the other programs. These numbers indicate that our approach probably has a high false positive rate (as do previous approaches). It is possible, however, to mitigate the problem of false positives by filtering and prioritizing bug reports in various ways, e.g., as in Xylem [15] and FindBugs. Also, as discussed in detail later, we propose that a dereference be marked as high priority if there is a static, inter-procedurally valid path that connects a null assignment statement to dereference. In fact, we find that fewer than 20% of our bug reports fall in this category. Our search of the bugzillas yields a separate observation, that null-dereference errors are a real problem commonly encountered during program usage, and worthy of the consideration of researchers in verification.

5.1 Categorization of error reports

To understand the reasons for large number of error reports on some benchmarks, we categorized the unsafe dereferences based on the reasons for the *root predicate* becoming $true$ during the analysis (see Section 2.1 for the definition of root predicate). Figure 11 shows the 6 categories used in the classification. A single dereference can go into multiple categories as several disjuncts may be introduced at various program points during the analysis of a dereference, with each of them becoming true for a different reason. The categories cover the most important reasons for imprecision but not every possible reason. In other words, some dereferences may not fall into any of the categories. For these reasons, the sum of the percentages across any row may be less than or more than 100%.

We first focus on the category *unbounded access-paths* shown in Figure 11. An unsafe dereference falls in this category if during its analysis some disjunct happened to contain a root predicate that was dropped because it had an access path with a repeating field or a reference to an element of an array. The figure shows that on an average 40% (max. 76% and min. 11%) of dereferences reported as unsafe by our analysis fall under this category, implying that recursive data structures and array accesses are ubiquitous in Java programs. However, handling these in a precise and scalable way is quite difficult.

The *Null-assignment* category includes all the dereferences, in whose analysis, a disjunct containing a root predicate was propagated through a null assignment statement that replaced the access-path in the root predicate by null. For instance, say during the analysis of dereference r at a program point p, a disjunct $<$v.f = null$>$ is propagated thorough the statement $S : v.f = null$. In this case, $v.f$ would be replaced by a $null$ in the resulting formulae and hence the dereference of r at p would be included in the null-assignment category. For the dereferences that fall under this category (which is around 19%) there exist at least one static path along which a null value flows to the root dereference. Hence, they are more likely to be true positives (they could still include false-positives because of infeasible paths not

Benchmarks	Lib. bytecode	App. bytecode	Preprocessing time (s)	analysis time (s)	# of derefs (excluding derefs of *this*)	Unsafe derefs	% deref verified
jlex	408	25056	6	9	2510	93	96.3%
javacup	509	29180	7	14	2851	607	78.7%
bcel	3596	86483	11	47	10143	1184	88.3%
jbidwatcher	16505	105043	17	530	9643	1490	84.5%
sablecc	4055	157169	20	189	14017	2116	84.9%
ourtunes	7206	127167	12	160	16449	1495	90.9%
proguard	2766	185594	19	149	17736	2778	84.3%
antlr	7224	251010	18	43892	17409	4042	76.8%
freecol	8889	260785	32	5815	24077	6994	71%
l2j	14310	373661	23	384	36899	5727	84.5%
							Avg = 84.02%

Figure 10. Results of analysing the benchmarks shown in Figure 9

Benchmarks	call-backs	missing-targets	virtual-calls	library calls	unbounded-aps	null-assignment
jlex	10%	1%	0%	0%	50%	27%
javacup	3%	0%	0%	0%	76%	2%
bcel	9%	1%	17%	0%	56%	25%
jbidwatcher	36%	19%	27%	1%	17%	21%
sablecc	6%	5%	6%	12%	22%	14%
ourtunes	3%	1%	0%	5%	68%	8%
proguard	7%	0%	38%	0%	57%	36%
antlr	9%	14%	51%	52%	20%	23%
freecol	4%	16%	20%	44%	13%	17%
l2j	17%	36%	0%	3%	11%	21%
Average	10.4%	9.3%	15.9%	11.7%	40%	19.4%

Figure 11. Percentage of unsafe dereferences in the various categories

caught by our limited path-sensitivity). It is an encouraging sign that this percentage is high (second highest among the categories) as it implies that for 19% of unsafe dereferences the analysis is able find one static inter-procedurally valid path (which is also to some extent path-sensitive) along which a null value flows to the point of dereference.

We now consider the categories *virtual calls* and *missing targets* that arise mainly due to the imprecision in the call-graph. A dereference falls into the *virtual calls* category if during its analysis some disjunct containing a root predicate had to be propagated though a virtual method call with more than 10 targets, at least one of which modifies the access path in the root predicate. Similarly, a dereference falls into the *missing targets* category if during its analysis some disjunct containing the root predicate had to be propagated though a method call with no targets (which are mostly calls to GUI libraries and JDBC libraries). In these situations we stop the analysis of the current dereference and call it unsafe (see Section 4.2). Figure 11 show that, on an average, a significant percentage of unsafe dereferences, namely around 16% and 9%, fall under the two categories, respectively. It is to be noted that in programs *freecol* and *antlr* for which our analysis generates many error reports, the number of dereferences that fall under one of the two categories is quite high (around 52% in *antlr* and 20% in *freecol*) implying that the large number of error reports are more likely the result of an imprecise call-graph.

We now discuss the categories *call-backs* and *library calls* that arise due to the complex interactions between the application and the library code, as explained in Section 4.2. A dereference falls into one of these categories, respectively, if during its analysis some disjunct containing the root predicate (a) reached the entry of a method that is *called-back* by a library method or (b) had to be propagated though a library method that has a *side-effect* on the access-path used in the root predicate. Figure 11 indicates that a significant number of error reports fall into these two categories in all benchmarks (particularly, in *antlr*, *freecol* and *jbidwatcher*). Later, in Section 5.4.2, we present an empirical evaluation

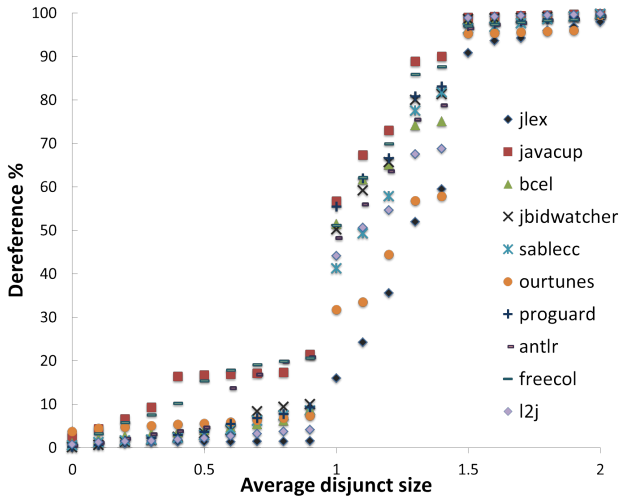


Figure 12. A point (x,y) in the graph indicates that in y percentage of dereferences the average disjunct size was $\leq$ x

of the impact of analysing all library methods and call-backs on the scalability the analysis.

5.2 Evaluation of the scalability of the analysis

We now present some metrics that highlights the scalability of our analysis. The most important measure of the scalability of a demand-driven analysis is the *response time*, which is the time taken by the analysis to report a dereference as safe or unsafe. The analysis is very efficient on the 8 programs other than antlr and freecol. In each of these 8 programs approximately 98% of dereferences take up to 250 ms each, while the rest take up to 14 seconds each. In the case of freecol, around 93% of dereferences take up to 250 ms each, while the rest take upto 51 seconds each. In antlr, 85% of dereferences take up to 250 ms each, 14.3% take up to 100 seconds each, while the remaining 0.7% take more than 100s and are timed out. We believe that the very low response time of our analysis makes it more suitable for use in a interactive development environment. In the later part of the section, we illustrate that in spite of having a very low response time our analysis is able to prove many complicated dereferences as safe by traversing long inter-procedural paths.

In our analysis there are three important factors that affect the scalability of the analysis wrt both time and memory, viz. summary hits/miss, number of disjuncts and the size of disjuncts in the DNF formulae computed after each propagation. We now present a few metrics that provides insight into these three factors. Figure 12 shows how the *average disjunct size* (averaged over all the disjuncts that arose at all program points) varies across the analyses of the dereferences for each of the benchmarks. Each point *(x,y)* in the graph means that in *y* percentage of dereferences the average disjunct size was less than or equal to *x*. The disjunct sizes can be less than one, because in our implementation the empty disjunct (disjunct with no predicates) models logical truth. From Figure 12, it can be seen that almost 100% of the dereferences have the average disjunct size less than or equal to 2 (across all benchmarks). It is to be noted that the maximum disjunct size can only be 4 as we bound the disjunct sizes. However, Figure 12 illustrates that at least 90% of all the dereferences in each program have average disjunct size less than or equal to 1.5. This is because (a) We do not include all the branch conditions encountered during a propagation of the disjunct into the disjunct (as discussed in Section 4.2, Issue 3), (b) We drop a predicate from a disjunct after it has been propagated through 1000 statements (see Section 2.1.2). Later, in Section 5.4.1, we show that in spite of having a very low disjunct size we are able to invalidate many unsatisfiable paths compared to a path-insensitive analysis.

We now discuss the metric *average number of disjuncts per propagation*, which refers to the ratio of the total number of disjuncts that were generated at all program points during the analysis of a dereference to the total number of *propagations* performed by the analysis (each application of a transfer function in Figure 3 is regarded as a propagation). It is only when a disjunct is propagated through a PUTFIELD statement that the number of disjuncts that results after the propagation can become more than one. As discussed in Section 2.1.2, after every PUTFIELD rule potentially 2^k disjuncts can be created, where k is total number of (unique) prefixes of the access-paths in the input disjunct. In our analysis, when a propagation through a statement results in a disjunct becoming false we stop its propagation. Hence, whenever a propagation invalidates an incoming disjunct, the number of disjuncts that results after the propagation is considered zero. For these reasons, the average number of disjuncts per propagation could be less than 1. We find that in all our benchmarks, for about 99% of the dereferences the average number of disjuncts per propagation is between 0.8 and 1, which indicates that the blow-up due to the PUTFILED rule is insignificant; i.e, most of the disjuncts that are added by the PUTFIELD rule get invalidated after a few propagations.

Figure 13 shows how the summary miss percentage varies across the dereference analyses for each of the benchmarks. We define summary miss percentage as the percentage ratio of the total number of times the summary lookups failed (see Line 18 in Figure 6) to the total number of times the summary lookups were performed during an analysis (the condition in Line 15). We do not count the summary lookup that happens the first time a method is encountered during the analysis of a dereference as it would be a compulsory miss. It is to be noted that we do not share the summaries across analyses of different dereferences, even within the same program. Figure 13 shows that in each of the benchmarks (except *bcel*), for about 45% of all the dereferences in

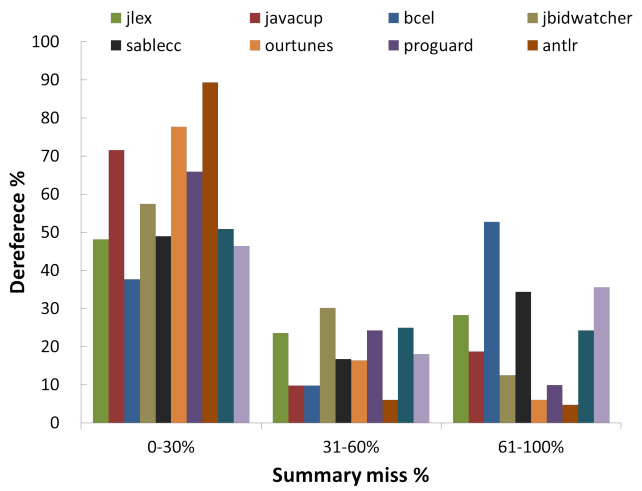


Figure 13. Summary miss percentages

the benchmark the summary miss ratio of the analysis is less than or equal to 30%.

5.3 Evaluation of the complexity of the dereferences reported as safe

In this section we present some metrics that highlights the characteristics of the safe dereferences identified by our analysis.

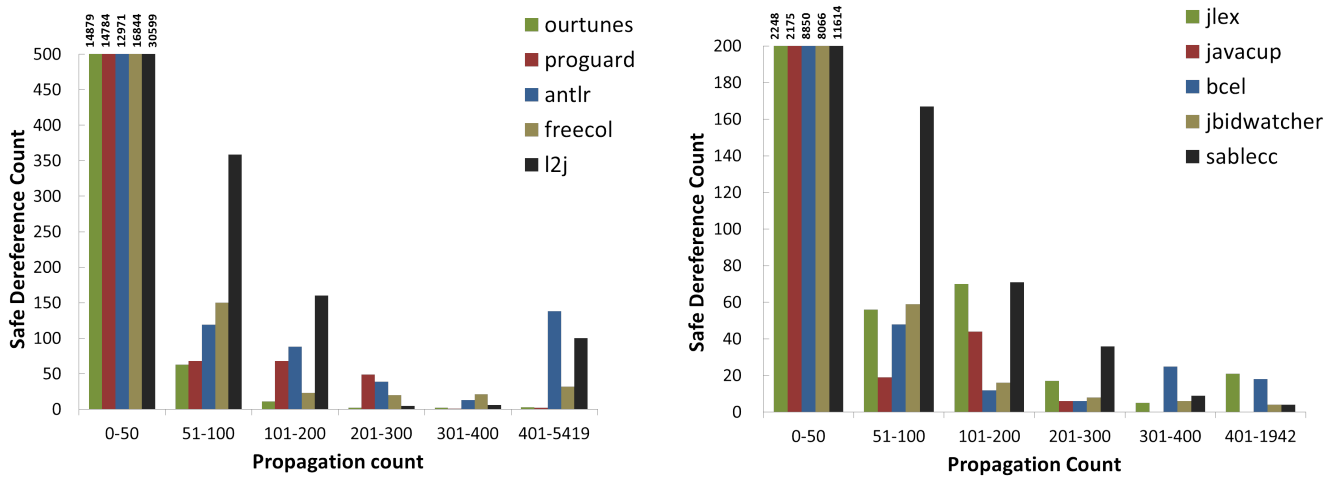
An advantage of our analysis is that it can propagate disjuncts over long inter-procedural paths. To measure the extent of the inter-procedural propagation our analysis performed, we use two metrics (a) *propagation count* (b) *context-depth*. We use the term *propagation count* to refer to the maximum length of the path along which a disjunct is propagated during the analysis of a dereference. A summary hit is counted as a single propagation. Figure 14 shows for each benchmark, the number of safe dereferences belonging to each of the six propagation count ranges shown along the x-axis. We limit the heights of the bars to 500, and show the actual height of each truncated bar on top of the bar. There are 81 (ourtunes) to 630 (l2j) safe dereferences in all the five larger programs that have a propagation count > 50 (see Part (a) of the figure). *l2j* and *antlr* have 100 and 138 safe dereferences with propagation count > 400. Interestingly, in *l2j*, the maximum propagation count was 1991 indicating that a dereference was proven safe after exploring a path of length 1991. Part (b) of the figure shows a similar trend for all the five smaller programs. In these programs (not counting *javacup*) there are 109 (bcel) to 287 (sablecc) safe dereferences with propagation count > 50.

The results clearly illustrate that our analysis is able to prove significant number of dereferences safe by exploring long paths. It is to be noted that, as mentioned in Section 5.2 our analysis has a very low response time in spite of traversing long paths.

We now present a metric *context-depth* that captures the maximum length of the call chain (measured from the method containing the dereference) that had to be explored by the analysis to prove a dereference safe. Formally, we define *context depth* of a dereference as the sum of the maximum number of entries in the context stack (at any point in the analysis of the dereference) used in the our inter-procedural analysis algorithm (see Figure 6) plus the number of methods that were exited through their entry statements but were not previously entered through their exit statements (these are the transitive callers of the method that contains the root dereference). For every dereference the context depth will be at least 1 by the above definition. Figure 15(a) shows that for 3 out of 5 programs (viz. *l2j*, *antlr* and *freecol*) there are 883 (antlr) to 1161 (freecol) safe dereferences with context depth ≥ 3. Particularly, in *antlr* 150 dereferences are proven safe after exploring inter-procedural paths with context depth > 10, indicating the need for a deep inter-procedural analysis for these dereferences. In fact, it can be seen from the Figure 15(a) that there are dereferences with context-depth of up to 160. Figure 15(b) shows a similar trend for the five smaller programs. In four programs (viz.*sablecc*, *javacup*, *jbidwatcher* and *bcel*), there are 170 (jlex) to 766 (sablecc) safe dereferences with context depth ≥ 3. In particular *bcel* and *sablecc* each have more than 700 dereferences with context depth ≥ 3.

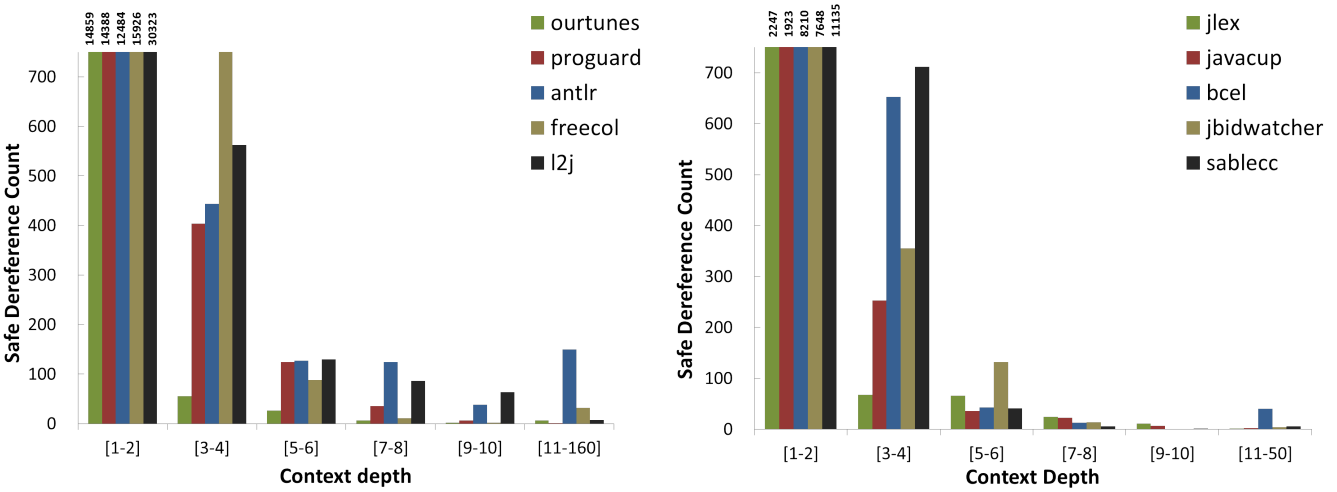
The dereferences with high context depth are difficult to discover as safe through manual code analysis. The Figure 16 shows a code snippet taken from the *l2j* benchmark that has a high context depth. In Figure 16 the root dereference is the variable t at line number 31. The root predicate before the line number 30 would be $airship.\_position = null$. It has to be propagated through the constructors of *L2AirshipInstance*, *L2Character*, *L2Object* and then through the methods *initPosition* and *setObjectPosition* where it is proven to be safe. In this example the context depth is 6 which is the length of the call chain from *L2AirshipInstanceParseLine* to *setObjectPosition*.

Another interesting measure of the complexity involved in proving a dereference safe is the length of the longest access-path encountered during the analysis of the dereference. In our experiments we found that for four of our benchmarks, namely, *bcel*, *jlex*, *proguard*, and *antlr*, around 1% to 10% of the safe dereferences have maximum access-path length greater than 2. In particular, in *jlex* some of the safe dereferences discovered by our analysis required access-paths of length up to 6. It is to be noted that in Salsa [13] the access-path lengths are limited to 2. Our finding illustrates that such a limit can prevent the analysis from proving a significant number of dereferences as safe in some benchmarks.



(a) large programs (b) small programs

Figure 14. Distribution of dereferences with respect to the propagation count of their analysis



(a) large programs (b) small programs

Figure 15. Distribution of the number of safe dereferences with respect to the *context depth* of their analysis

5.4 Evaluation of optimizations

As discussed in Section 4.2, we various optimizations and heuristics to make the analysis scale to real world Java programs. In this section we measure the impact of these optimizations on precision/scalability through a series of experiments carried out on our benchmarks. Due to the high running time and memory overhead in analysing *freecol* and *antlr*, we exclude them from this series of experiments.

5.4.1 Evaluation of limited path-sensitivity

We evaluate our default analysis (with all our optimizations, including limited path sensitivity) that we used to report all the results above by comparing it with two variants of our analysis (a) a variant that includes *all* branch conditions in the path (upto a specified bound), referred to as *mode(a)* (b) a variant that includes no branch conditions at all in the path (i.e. path insensitive), referred to as *mode(b)*. In *mode(a)*, which tracks all branch conditions, we enriched the predicate domain to include predicates involving boolean variables and integers. However we do not model integer arithmetic and track only $<, >, =$ comparisons on integers. We also added new simplification rules to the simplifier so that it takes into account the new predicates (involving boolean variables and integers) during the simplification of a disjunct. In fact, the domain and the simplifiers we used are very similar to those used in Xylem [15]. We found that tracking all branches without any limit will not scale to even the smallest of our programs viz. *jlex*. Hence, we bounded the disjunct sizes using the parameters k_1 and k_2 (see Section 2.1.2), which limit the number of propagations

```
1 class L2Object {
     ObjectPosition _position
2    public L2Object(){
3        ...
4       initPosition();
5    }
6    public void initPosition(){
7        ...
8        setObjectPosition(new CharPosition(this))
9    }
10   public void setObjectPosition(...) {
11        _position = value
12   }
13   public ObjectPosition getPosition() {
14       return _position;
15    }
16 }
17 class L2Character extends L2Object {
18    public L2Character() {
19        super()
20    }
21 }
22 class L2AirShipInstance extends L2Character {
23    public L2AirShipInstance() {
24        super()
25    }
26 }
27 public void L2AirShipInstanceParseLine() {
28    ...
29    airship = new L2AirShipInstance(..);
30    t = airship.getPosition()
31    t.setHeading();
32    ...
33 }
```

Figure 16. A real world example with high context depth

of a predicate and the sizes of the disjuncts. We set k_1 and k_2 to the same values as in the default setting (which is 1000 and 3 respectively).

Figure 17 shows the number of unsafe reports and the time taken (excluding preprocessing time) for running the analysis on all the dereferences in each of the 3 different modes. In spite of having the bounds on the disjuncts, the version that tracks all branches, *mode(a)*, did not scale to many of the larger programs (which are not shown in Figure 17) within reasonable time limits.

Figure 17 also shows that the version without path-sensitivity, *mode(b)*, is very imprecise in comparison with the limited path-sensitivity mode. In fact, it even takes more time than limited path-sensitivity mode (because the limited path-sensitive mode can avoid propagation through unsatisfiable paths thereby, reducing the running time). An interesting observation is that for *jlex*, the precision of the *mode(a)* is actually less than that of the limited path-sensitivity mode. This is because in *mode(a)* some of the relevant branch predicates (conditions that can invalidate a path) get crowded out by less relevant predicates (due to bounds on disjunct sizes) before they come in useful to invalidate the disjunct.

5.4.2 Evaluation of limited library analysis

As discussed in Section 4.2, Issue 6, we over-approximate the effect of a library method using its Mod-Ref information, except in the cases where its return value is used in an access-path. We also provide two manually created lists, viz. the *skip* and *analyze* lists, to the analysis to mitigate the loss of precision due to the above approximation. Similarly, we also prevent the propagation of a disjunct reaching the entry of an analyzed method through its caller if it happens to be a library method (referred to as library call-back). To evaluate the precision/scalability trade-off in using this approach, we consider two variants of our analysis (viz. *mode(a)* and *mode(b)*) as explained below, and compare them with our default analysis (all optimizations turned on).

In *mode(a)* we allow the analysis to analyze all library methods that are found to have a side-effect on an access-path by the WALA's Mod-Ref analysis, and also allow a disjunct reaching the entry of a method to be propagated through all its callers irrespective of whether they belong to the library or not. In *mode(b)* we perform a limited library analysis but do not use the manually created *skip*/*analyze* lists. Note that the use of the analyze list can increase precision, but with extra cost in analysis time. The skip list neither increases or decreases the running time, impacting only precision.

Figure 18 shows the result of analysing the benchmarks in the three different modes. We use ∞ in Figure 18 to denote that the analysis of a benchmark did not complete within a reasonable time limit (which is more than twice the analysis time of the limited library analysis mode).

Figure 18 shows that *mode(a)* (in which all the libraries having side-effects are analyzed) does not scale to many of the large benchmarks, clearly indicating that analysing all library methods will not scale to real world applications.

In *mode(b)* (in which analyze and skip lists are not used), the analysis time is significantly lower than the limited library analysis mode for some benchmarks like *jbidwatcher*. This is because *mode(b)* analyses even fewer library methods than the limited library analysis mode. However, it results in some loss of precision in half of the programs shown in Figure 18. In fact, in *ourtunes*, *mode(b)* results in almost 5% increase in the number of reported unsafe dereferences compared to the limited library analysis mode.

5.4.3 Evaluation of weaker disjunct summary reuse optimization

As mentioned in Section 3.2, in cases where we do not have a mapping for a disjunct (say ϕ_1) in the summary table, but have a mapping for a weaker disjunct ϕ_2, we reuse the summary computed for the weaker disjunct ϕ_2 provided the the root predicate (if any) in both the disjuncts matches. Figure 19 shows the results of running the analysis, first

	Default setting *Limited path sens.: tracks a few branches*		**Mode(a)** *Tracks all branches*		**Mode(b)** *Tracks no branches*	
Benchmark	*Unsafe derefs*	*Time(s)*	*Unsafe derefs*	*Time(s)*	*Unsafe derefs*	*Time(s)*
bcel	1184	63	1119	8123	2501	97
javacup	607	19	463	881	1036	29
jlex	93	13	105	148	296	33

Figure 17. Result of running the analysis with different path-sensitivity

	Default setting Limited lib. analysis, *with both lists*		**Mode(a)** *Analyzes all libs*		**Mode(b)** *limited lib. analysis, w/o the lists*	
Benchmark	*Unsafe derefs*	*Time(s)*	*Unsafe derefs*	*Time(s)*	*Unsafe derefs*	*Time(s)*
jlex	93	13	93	13	93	12
javacup	607	19	607	19	607	18
bcel	1184	63	1184	74	1184	63
jbidwatcher	1490	390	-	∞	1493	103
sablecc	2116	252	-	∞	2116	239
ourtunes	1495	227	-	∞	1540	222
proguard	2778	207	-	∞	2783	183
l2j	5727	545	-	∞	5746	533

Figure 18. Evaluation of the precision/scalability trade-off in analysing library methods

	Default setting		w/o Weaker disjunct Summary reuse	
Benchmark	*Unsafe derefs*	*Time(s)*	*Unsafe derefs*	*Time(s)*
jlex	93	13	92	18
javacup	607	19	607	18
bcel	1184	63	1184	104
jbidwatcher	1490	390	1490	597
sablecc	2116	252	2116	275
ourtunes	1495	227	1495	228
proguard	2778	207	2778	204
l2j	5727	545	5727	550

Figure 19. Results of the analysis with and without the weaker disjunct summary reuse optimization

in the default mode (i.e., with all optimizations), and then without the reuse optimization.

It can be seen that for most of the programs shown in Figure 19, the analysis time increases in the absence of this optimization (particular for *bcel* and *jbidwatcher*). Figure 19 also shows that this optimization has *no negative impact* on the precision of the analysis (i.e, the number of error reports) in all programs, except *jlex* in which the number of error reports increased by just 1.

6. Comparison with related work

The approaches of Xylem [15], Salsa [13] and Spoto [19] are the most closely related approaches to ours, in the sense that they target null-dereference analysis of real Java programs. We discuss these approaches in detail first, and later give an overview of other related techniques.

Xylem. Xylem is a *bug finding* technique rather than a verification technique, meaning they may miss real bugs, and may also report false positives. Our approach, though aiming at verification, has several attributes that are inspired by Xylem; e.g., a demand-driven backward dataflow analysis from each dereference, predicates as dataflow facts, custom simplification rules for predicates rather than a theorem prover, a context-stack during inter-procedural dataflow propagation for context-sensitivity, abstracting out arithmetic, and inter-procedural summary tables for efficiency. However, there are several key differences. Xylem aims for very high precision, and hence uses a richer set of predicates that result in a greater extent of path-sensitivity. This could have an adverse impact on scalability. In order to ensure reasonable analysis time they enforce various limits on the analysis, like the sizes of disjuncts (which we do, too), the number of paths analyzed from a dereference (we do not), and even on the analysis time (which we do not). When the analysis of a dereference gets terminated due to any of these thresholds being crossed, they ignore the dereference and go on to the next one, implying unsoundness.

In our approach, we aim to conservatively label *each* dereference as safe or unsafe, with reasonable precision, meaning we need to maximize the number of paths back from the dereference that we analyze, as well as the depths of these paths. Therefore, for scalability, we use a smaller

lattice, and more limited path-sensitivity. Within this setting, in order to maximize precision, we do not drop an entire disjunct when its size crosses a threshold, which would immediately terminate the analysis (with an "unsafe" answer); instead, we drop individual "old" predicates from the disjunct in order to reduce its size, and then continue its propagation (as discussed in Section 2.1.2). Also, when we encounter a library call that we choose not to analyze (as discussed in Section 4), we drop individual predicates that may be mutated by the call, and not the entire disjunct. Therefore, we give up precision in a fine-grained manner. We also address recursion completely, analyzing a recursive method until a fix-point is reached; Xylem does not compute fixpoints for recursive methods, instead using ad-hoc bounds to terminate the analysis. Another distinction is that we perform strong updates at put-field statements for precision, by keeping track of hypotheses on aliasing relations between variables, and validating (or invalidating) these hypotheses at assignment statements. Having mentioned these distinctions between the two approaches, it is worth reiterating that Xylem's objective is different from ours (i.e., to find a few, important bugs), which means some of these distinctions are sensible from both perspectives.

In subsequent work [14] the same authors apply Xylem, with a few modifications, to the problem of identifying necessary conditions on inputs to *model-transformation* programs that cause these programs to throw various exceptions, including null-dereference exceptions.

Salsa. Salsa is an approach that aims at sound null-dereference verification of Java programs. It is not based on propagating conditions. Rather, it is a (non-demand-driven) dataflow analysis that uses a custom designed lattice to track at each program point access paths that are known to be definitely non-null at that point. In their approach they perform limited-scope analysis (in terms of depths of call-chains considered) for scalability; we have shown scalability without scope limitation; in fact, in our evaluation we have found numerous dereferences that require analysis over deep call chains. Also, the extent to which Salsa can perform strong updates is dependent on the precision of a pre-requisite must-alias analysis, whereas we avoid the need for must-alias analysis by keeping track of aliasing relationships at each point in each path that we analyze.

In order to facilitate a quantitative comparison between our approach and theirs, we have chosen 8 benchmarks from their list of benchmarks (plus two more from outside their list). While on two of these benchmarks (namely, *jlex* and *bcel*) the percentage of dereferences we report as unsafe is less than their corresponding number, they do better on the other six benchmarks. This said, precision comparisons between the two approaches are difficult, for multiple reasons. For many of the benchmarks they report a benchmark size that is much smaller than our corresponding number; potential reasons for this include differences in version numbers (they do not indicate the version numbers they use), and in the entry points into the program that were considered. More importantly, they appear to never analyze library methods, and depend on programmer specifications of the behaviors of these libraries. We analyze library methods, in general, with some exceptions. In our experiments, over all benchmarks, we entered and analyzed the bodies of 625 distinct library methods. It is very difficult to write meaningful, precise specifications for library methods, especially when they have side effects. The only manual inputs we use are the skip-list and the analyze-list of library methods, which are a very simple form of specification.

The approach of Spoto. Spoto describes a flow- and context-sensitive forward, non-demand driven, static analysis to conservatively find null dereferences. They address exceptions, but not multi threaded programs. They use an abstract lattice of formulas that is more expressive than ours, wherein they encode formulas using Binary Decision Diagrams (BDDs). Somewhat surprisingly, it is actually not clear that their precision is better than ours in practice. There are two common programs in our benchmark set and theirs – jlex and javacup. They report separate precision numbers for "getfield", "putfield"[2], and "call" dereferences, whereas we report a single overall precision number over all dereferences. On jlex our overall precision is 96.3%, whereas theirs works out to 78.7%. In the case of javacup, our overall precision is 78.7%, while theirs works out to 85.4%. There a couple noteworthy caveats, though: they do not mention the version of each program they analyze, nor the number of dereferences in each program, nor the exact procedure they use to select the dereferences (we analyze all dereferences in methods reachable from the entry points). Regarding running time, our analysis time over all dereferences is approximately double theirs, but our analysis is demand driven, with no sharing of intermediate results at all between analyses of different root dereferences.

Other related techniques. There have been several approaches reported in the literature for verifying assertions in programs using very precise forms of reasoning. (Note that safety checking of assertions is a more general problem, of which dereference verification is an instance.) For instance, Slam [1] uses predicate abstraction, and counter-example guided abstraction refinement, to initially over-approximate the weakest at-least-once pre-condition of a given condition, and then successively strengthen the over-approximation until a concrete trace is produced or a timeout is reached. Synergy [10] follows a similar approach, but uses *concolic*, i.e., simultaneous concrete and symbolic execution, to accelerate the strengthening of the over-approximation. The approach of Dillig et al [6] over-approximates the weakest at-least-once pre-condition directly, without predicate abstraction.

[2] Although it is not fully clear from their paper, we assume that they include `arrayload` and `arraylength` instructions within their getfield category, and `arraystore` instructions within their putfield category.

These systems have been designed for property-verification in C programs, and have been shown to be precise in practice. While in theory they could be used for null-dereference verification for Java, their practical applicability in this setting is not clear. Large Java programs have certain characteristics not shared by C programs, such as extensive use of heap references, virtual method dispatch, deeply nested method calls (with small methods), and call backs. Moreover, these approaches do not appear efficient enough for use by an individual developer as a part of their desktop development environment to verify dereferences in a large application.

ESC/Java [8] and Spec# [2] use an abstraction-free weakest pre-condition analysis to verify assertions. They rely extensively on programmer-given annotations, e.g., method pre- post- conditions, and loop invariants, for completeness, modularity, and scalability.

There are several approaches, e.g., Snugglebug [3], Java PathFinder [20], and DART [9], that use precise symbolic or concolic analysis to search for a concrete execution path that ends at a given program point with a state that satisfies a given condition. In other words, these approaches *under-approximate* the weakest at-least-once pre-condition, rather than over-approximate it (as we do). These approaches may not always terminate in their search in the presence of loops and recursion. Their approach is applicable when one is trying to confirm a potential bug, but is not applicable to the problem of proving that an assertion is safe. (The approach of Dillig et al, as well as Synergy, are capable of under-approximation also.)

It is noteworthy that there are several interesting ideas used in the over-approximating as well under-approximating approaches mentioned above, that could potentially be incorporated into our own approach to improve its precision, scalability, or suitability for checking properties other than dereference safety.

7. Conclusions and Future Work

We have presented a demand-driven approach for verification of null-dereferences, based on over-approximation of the weakest at-least-once pre-condition, using a novel set of design decisions to make the approach practical. We have implemented the approach, and have evaluated it on a set of real Java programs. Our experimental results indicate the our approach is scalable to large programs, has very quick response time per dereference, and has reasonable precision. To the best of our knowledge ours is the first practical weakest pre-conditions-based verification approach to be demonstrated on large, real Java programs. Future work will be guided by objective of increasing the precision of the approach, while still retaining its practicality and demand-drivenness. In particular, we would like to investigate more precise techniques to reason about references to arrays and recursive data structures, perhaps by inferring and using specifications for container classes (which often encapsulate arrays and recursive data structures). We would also like to investigate efficient approaches to deal with difficult idioms such as call backs. Finally, we would also like to investigate application of our approach to verification problems other than null-dereference analysis.

Acknowledgments. We thank Amogh Margoor, graduate student, and Ankur Sinha, project intern at IISc, for their valuable and substantial help with the experimental evaluation. We thank our funding partners Infosys, IBM Research India, and Microsoft Research India for their support.

References

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *Model Checking Software*, volume 2057 of *Lecture Notes in Computer Science*, pages 102–122. Springer Berlin / Heidelberg, 2001.

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The spec# programming system: An overview. In G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet, and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin / Heidelberg, 2005.

[3] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI '09: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 363–374, New York, NY, USA, 2009. ACM.

[4] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[5] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.

[6] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI '08: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 270–280, 2008.

[7] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In A. Gordon, editor, *Programming Languages and Systems*, volume 6012 of *Lecture Notes in Computer Science*, pages 246–266. Springer Berlin / Heidelberg, 2010.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proc. SIGPLAN Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[9] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. In *PLDI 05: Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[10] B. Gulavani, T. Henzinger, Y. Kannan, A. Nori, and S. Rajamani. Synergy: An new algorithm for property checking.

In *FSE '06: Proc. ACM SIGSOFT Symp. on Foundations of Software Engineering*, pages 117–127, 2006.

[11] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to null pointer bugs. In *PASTE '05: Proc. ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering*, pages 13–19, 2005.

[12] G. Kildall. A unified approach to global program optimization. In *POPL '73: Proc. ACM Symposium on Principles of Programming Languages*, pages 194–206, New York, NY, USA, 1973.

[13] A. Loginov, E. Yahav, S. Chandra, S. Fink, N. Rinetzky, and M. Nanda. Verifying dereference safety via expanding-scope analysis. In *ISSTA '08: Proc. International Symposium on Software Testing and Analysis*, pages 213–224, New York, NY, USA, 2008. ACM.

[14] M. G. Nanda, S. Mani, V. S. Sinha, and S. Sinha. Demystifying model transformations: an approach based on automated rule inference. In *OOPSLA '09: Proc. ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications*, pages 341–360, 2009.

[15] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE '09: Proc. International Conference on Software Engineering*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society.

[16] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. *Logic in Computer Science, Symposium on*, pages 55–74, 2002.

[17] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Trans. Program. Lang. Syst.*, 24:217–298, May 2002.

[18] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. S. Muchnick and N. D. Jones, editors, *Program Flow Analysis: Theory and Application*. Prentice Hall Professional Technical Reference, 1981.

[19] F. Spoto. Precise null-pointer analysis. *Software and Systems Modeling*, 10:219–252, 2011. 10.1007/s10270-009-0132-5.

[20] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. In *ISSTA '04: Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[21] *T.J. Watson Libraries for Analysis (WALA), http://wala.sf.net*.

A. Formalizing our analysis as an abstract interpretation

In this section we sketch a formulation of our weakest at-least once pre-condition analysis as an abstract interpretation. As is conventional, we first formulate the concrete semantics as a dataflow analysis. The concrete lattice is the powerset lattice of the set of states (i.e., concrete stores). The backward concrete transfer function c_{st} of any statement *st* is as follows: If *st* is any statement other than an ASSUME then $c_{st} = \lambda S.\{s_1 \mid \exists s_2 \in S : st \text{ transforms } s_1 \text{ to } s_2\}$. If *st* is "ASSUME b" then $c_{st} = \lambda S.\{s \mid s \in S \text{ and } s \text{ satisfies } b\}$. The join operation is union (therefore, $\leq$ is $\subseteq$). Let p be a program point, and C be a post-condition at p. The initialization of the concrete analysis is as follows: the set of stores satisfying C at p, and the empty set at all other program points. It is easy to show that the join over all paths solution at the entry of the program according to the above concrete analysis is precisely the set of states that satisfy $wp_1(p, C)$.

Our abstract lattice and transfer functions were presented in Section 2. The concretization function γ is $\lambda F \in Formula.\{s \mid s \text{ satisfies } F\}$, which is monotonic. For each statement type *st* the abstract transfer function f_{st} for *st* is the composition of its flow function in Figure 3 and the simplifier fules in Figure 5 (as was discussed in Section 2.2). Each abstract transfer function f_{st} is monotonic, and also conservatively over-approximates the corresponding concrete transfer function c_{st}; i.e., for any formula $F \in Formula$, $\gamma(f_{st}(F)) \supseteq c_{st}(\gamma(F))$. The initialization for the abstract analysis is as follows: the given disjunct (i.e., post-condition) C at the given program point p, and the empty set (i.e., false) at all other points. Therefore, it follows that the pre-condition computed at the program's entry by our analysis is equal to or weaker than $wp_1(p, C)$.

It is instructive to contrast our analysis with a weakest pre-conditions analysis. Whereas we over-approximate the weakest at-least once pre-condition, it is natural to *under-approximate* the weakest pre-condition. In this setting, the backward abstract transfer f_{st} of any statement *st*, when applied to any formula F, ought to return a formula F' such that when *st* is executed on the set of states S' that satisfy F', *every* state that results satisfies F. We omit the details of the lattice and transfer functions in this setting, which are somewhat different from the ones we use. For instance, for the GETFIELD instruction $v = r.f$, the transfer function when applied to a post-condition ϕ would return a pre-condition $r = null \vee \phi[r.f/v]$, rather than what we return, namely $r \neq null \wedge \phi[r.f/v]$; also, in order to bound any access path safely, the transfer function would need to make the predicate that contains the access path $false$, rather than $true$. Furthermore, while the concretization function γ would be the same as in our setting, the join operation on the abstract lattice would be logical AND (rather than the union we use, which implements logical OR).