Accurate Interprocedural Null-Dereference Analysis for Java

Mangala Gowri Nanda and Saurabh Sinha
IBM India Research Lab
{mgowri,saurabhsinha}@in.ibm.com

Abstract

Null dereference is a commonly occurring defect in Java programs, and many static-analysis tools identify such defects. However, most of the existing tools perform a limited interprocedural analysis. In this paper, we present an interprocedural path-sensitive and context-sensitive analysis for identifying null dereferences. Starting at a dereference statement, our approach performs a backward demand-driven analysis to identify precisely paths along which null values may flow to the dereference. The demand-driven analysis avoids an exhaustive program exploration, which lets it scale to large programs. We present the results of empirical studies conducted using large open-source and commercial products. Our results show that: (1) our approach detects fewer false positives, and significantly more interprocedural true positives, than other commonly used tools; (2) the analysis scales to large subjects; and (3) the identified defects are often deleted in subsequent releases, which indicates that the reported defects are important.

1 Introduction

Null dereference is a commonly occurring bug in Java programs, and many static-analysis techniques and tools have been developed for detecting such bugs (e.g., [3, 4, 5, 6, 7, 9, 10, 16]). The occurrences of such bugs often involve interactions among multiple procedures (e.g., as observed in Reference [3]); failure to take into account such interactions can limit the usefulness of a tool. Although, many interprocedural null-dereference analyses have been developed for C and C++ programs (e.g. [2, 3, 16]), fewer such techniques have been developed and evaluated for Java programs. In fact, some of the most widely used Java static-analysis tools, such as FINDBUGS [8], JLINT,[1] and ESC/JAVA [7], perform limited interprocedural analysis.

In recent work, two approaches for interprocedural null-dereference analysis for Java programs have been presented. Loginov et al. [12] present a sound analysis, based on abstract interpretation, that gradually expands the interprocedural scope of analysis to establish the safety of a dereference. A sound analysis discovers all potential bugs and, therefore, is appropriate as a *verification technique* (where the goal is to show the absence of bugs); but it can often report many spurious warnings (or, *false positives*). Tomb et al. [15] present a symbolic-execution-based analysis, which is more appropriate as a *bug-detecting technique* (where the goal is to identify as many bugs as possible—and the emphasis is not on reporting all potential bugs, but on reducing false positives.) Their approach is parametrized by the call depth to be explored (which, in their empirical study, is limited to two).

Our approach. In this paper, we present a bug-detection technique that is accurate, efficient, and addresses the limitations of existing bug-detecting tools. Our approach is interprocedural, context-sensitive, and path-sensitive. Starting at a statement s that dereferences variable v, our approach performs a backward analysis, taking into account branch correlations, to identify a path along which v may be null at s. A backward analysis is inherently *demand-driven*: it explores only those program paths and program states that are relevant for analyzing a dereference, which lets our approach scale.

A novel feature of our approach is the use of a simple, yet remarkably effective, abstract domain to model the program state. The simplicity of the domain causes the analysis to be efficient; and, although we do not use a constraint solver to handle integer arithmetic, the analysis has a surprisingly low false-positive rate.

Another key feature of our approach is that it does not limit the call depth to be explored, but follows call chains as far as necessary. This not only lets the analysis eliminate many false positives, but also detect more bugs than would be detected by a limited interprocedural analysis performed by tools, such as FINDBUGS, or the limited call-depth exploration described by Tomb et al. In fact, some of the bugs identified by our approach involved the propagation of null values through more than 15 methods. To perform efficient analysis of called methods, we compute and saves summary information at call sites; by reusing summary information, we avoid reanalyzing a method in a context in which the method has been analyzed previously.

Although we do not limit the call depth, we use other analysis parameters to control the program exploration.

[1]`http://artho.com/jlint`

These parameters can be used to bound the paths and states that are explored, and enable the approach to perform a cost-accuracy trade-off.

At a call to a library method, or at the entry of a method that has no callers, we use a notion similar to the idea of consistency checking [5] to determine how to classify a dereference that receives an unknown value. To elaborate, checking a variable against null establishes a programmer's belief that the variable may be null. Consequently, an unconditional dereference of the variable is reported as a potential null dereference. We extend this concept to identify interprocedural null checks.

Our approach can be used in a *demand mode*, in which the user specifies the starting statement for the analysis. Alternatively, the approach can be used in a *batch mode*, in which the backward analysis is performed starting at each dereference statement in a program. In the batch mode, the summary information computed at a call site during the analysis of a dereference is reused during the analyses of other dereferences; thus, the reuse is not limited to the analysis of one dereference. The demand-mode execution is suitable for providing on-line support for bug detection or debugging, whereas the batch-mode execution is appropriate for off-line comprehensive analysis of code builds. For the empirical studies reported in the paper, we performed the analysis in the batch mode.

Empirical evaluation. We have implemented the approach in a tool XYLEM, and conducted empirical studies using three large open-source projects and three commercial products (listed in Table 1). In the studies, we evaluated the accuracy and efficiency of XYLEM, the effects of the analysis parameters, and the relevance of the detected bugs (measured in terms of whether the bugs get deleted). Our results illustrate that XYLEM is significantly more accurate and effective than FINDBUGS and JLINT.[2]

- On average, XYLEM found 16 times more bugs than FINDBUGS and 3 times as many bugs as JLINT.
- Although XYLEM identifies more potential bugs, it generates very few false positives. For one of our subjects, `Ant`, we manually examined the reported bugs, and found that only four of the 86 reported bugs were false positives. For FINDBUGS, one out of 11, and for JLINT, 32 out of 42 bugs were false positives.
- For `Ant`, XYLEM caught each true positive, and invalidated each false positive, reported by either FINDBUGS or JLINT.
- For `Ant`, 75% of the bugs detected by XYLEM were interprocedural bugs. Only 9% and 17% of the bugs detected by FINDBUGS and JLINT, respectively, were interprocedural.
- On average, 24% of the bugs detected by XYLEM got deleted in subsequent releases compared to 10% for FINDBUGS. In absolute numbers, XYLEM and FINDBUGS detected 257 and 11 bugs, respectively, that got deleted.
- For none of the subjects, the null-dereference analysis required more than 11 minutes. To date, the largest application analyzed by XYLEM has over 1,009,000 lines of code.

XYLEM has been deployed internally, in pilot mode, with several development teams in IBM. In initial feedback from three teams, we have learned that approximately 31% of the null dereferences examined by these teams were considered "must-fix" by developers. Although preliminary, the data indicate the usefulness of the analysis.

Contributions. The main benefit of our approach is that it enables an accurate and efficient null-dereference analysis of Java software: it detects potential bugs that other tools miss; it eliminates false positives that other tools identify; and it scales to large software systems.

The contributions of the paper include

- A context- and path-sensitive interprocedural analysis for identifying potential null dereferences that is demand-driven and parametrized for cost-accuracy trade-offs
- Empirical studies, using large open-source and commercial products, that illustrate the effectiveness, efficiency, and usefulness of our approach.

2 Preliminary analysis

Prior to null-dereference analysis, our approach performs interprocedural data-flow analysis to compute, for each method, escape-in and escape-out variables. An *escape-in variable* of method M is a direct or indirect field of a formal parameter that is used, before possibly being defined, in M. An *escape-out variable* of M is either the return value of M, or a direct or indirect field of a formal parameter or return variable that is defined in M.

The preliminary analysis also constructs the control-flow graph for each method.[3] The analysis adds nodes to the CFG to represent placeholder assignments to escape-in/out variables, formal parameters, and actual parameters. At method entry, the approach creates a `FormalIn` node for each formal parameter and each escape-in variable of the method. At method exit, the approach creates a `FormalOut` node for each escape-out variable and for the return value. At each call node, the approach creates an `ActualIn` node for each formal parameter and escape-in variable of the

[2]For the studies reported in this paper, we used FINDBUGS 1.3.4 and JLINT 3.1.

[3]The *control-flow graph* (CFG) for a method M contains nodes that represent statements in M and edges that represent potential flow of control among the statements.

```
 [1] foo( B b, C c ) {
 [2]   T t1 = bar( c );
 [3]   r = c;
 [4]   b.f = r.m();
     }
 [5] bar( C x ) {
 [6]   T t = null;
 [7]   C y = x;
 [8]   if ( y == null )
 [9]      t = new T();
[10]   t.g = 10;
[11]   return t;
     }
```

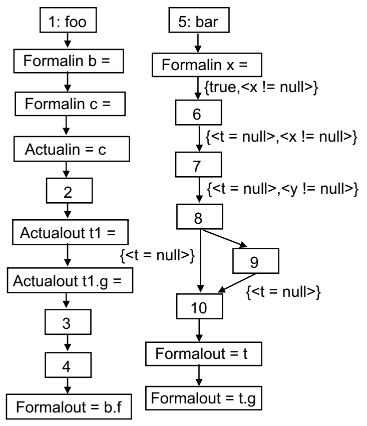


Figure 1. Example code fragment with CFGs.

callee; and an `ActualOut` node for each escape-out variable and the return value of the callee.

> **Example 1.** Consider the code fragment and CFGs shown in Figure 1. In the CFG for `foo()`, we create two `FormalIn` nodes for the two formal parameters, and a `FormalOut` node for the escape-out variable `b.f`. For the call to `bar()` at line 2, the approach creates an `ActualIn` node to represent the actual parameter `c`, and two `ActualOut` nodes: one to represent the assignment of the return value to `r`, and another to represent the escape-out variable `t.g`.

3 Null-Dereference Analysis

Our approach consists of a backward path-sensitive analysis. Starting at a dereference statement s that dereferences variable v, the analysis propagates a set of abstract state predicates backward in the CFG. Initially, the state contains the predicate $\langle v = \texttt{null}\rangle$. Each CFG node potentially transforms the state flowing in from its out-edges. For example, the predicate $\langle r = \texttt{null}\rangle$ generated at line 2 flows into line 1 and gets transformed to $\langle c = \texttt{null}\rangle$.

```
[1] r = c;      <c = null>
[2] r.foo();    <r = null>
```

We refer to the initial variable v as the *root reference* ($\mathcal{R}_r$) and the initial predicate $\langle v = \texttt{null}\rangle$ as the *root predicate* ($\mathcal{R}_p$). At an assignment statement $v = w$, the root reference is updated to w and the root predicate becomes $\langle w = \texttt{null}\rangle$. The goal of the analysis is to find an assignment of a null value to the root reference.

An any point during the analysis, the state predicates represent the conditions under which a null value for v can flow to s. If, at a statement, the transformed state becomes inconsistent (i.e., it contains a pair of conflicting predicates), the path is infeasible and the analysis abandons traversal along the path. Figure 2 presents the state predicates that are tracked by the analysis.

3.1 Inconsistency checking

Although our analysis is interprocedural, it makes approximations at program points where the code is not avail-

$$
\begin{array}{lcl}
\mathit{pred} & ::= & \langle \mathit{refVar}\ \ \mathit{refOp}\ \ \texttt{null}\rangle \mid \langle \mathit{refVar}\ \ \mathit{refOp}\ \ \mathit{refVar}\rangle \mid \\
& & \langle \mathit{refVar}\ \ \mathit{refOp}\ \ \mathit{strConst}\rangle \mid \langle \mathit{boolVar}\ \ \mathit{refOp}\ \ \texttt{true}\rangle \mid \\
& & \langle \mathit{intVar}\ \ \mathit{intOp}\ \ \mathit{intVar}\rangle \mid \langle \mathit{intVar}\ \ \mathit{intOp}\ \ \mathit{intConst}\rangle \\
\mathit{pred} & ::= & \neg \mathit{pred} \\
\mathit{refOp} & ::= & = \\
\mathit{intOp} & ::= & < \mid \leq \mid =
\end{array}
$$

Figure 2. Predicates tracked by the analysis.

able for analysis. This occurs at a call to an *external method* (e.g., a library method) or at the entry of a method that has no caller in the application being analyzed. At such points, the algorithm uses the notion of inconsistency checking [4, 5] to determine whether a null value may flow into the application. *Inconsistency checking*, introduced by Engler and colleagues [5], is a technique that identifies code fragments that represent contradictory programmer beliefs.

> **Example 2.** In Figure 1, parameter `c` of `foo()` is assigned to `r`, which is dereferenced unconditionally at line 4. This generates the belief that `c` is expected to be non-null. However, the program also contains a null check on `c` at line 8 in `bar()`, after `c` is passed as a parameter to `bar()` at line 2 and assigned to `y` at line 7. This indicates a belief that `c` could be null. Together these dereferences represent contradictory programmer beliefs about `c`. For parameter `b`, the program contains an unconditional dereference only; therefore, `b` is assumed to be non-null.

The following *null-check rule* describes the consistency checking performed by the algorithm. A reference r that (1) receives its value from outside the analysis scope and (2) is checked for a null value before a dereference along a path is assumed to be potentially null. If r is not checked for a null value or is dereferenced along all paths to a null check, r is assumed to be non-null. In Example 2, the algorithm would flag the dereference of `r` at line 4 as a potential null dereference, whereas it would consider the dereference of `b` at that line to be safe.

FINDBUGS uses such a rule, but it searches for null checks within a method only. The novelty in our approach is that the analysis tracks null checks across methods, as illustrated in Example 2. Moreover, our approach applies the null-check rule only at points where the callee or caller is not available for analysis. If the program in Figure 1 were to contain a caller of `foo()` that passed in only non-null values for parameter `c`, our algorithm would analyze the caller and establish the safety of the dereference of `r` at line 4.

3.2 State transformation

Figure 3 presents the state transformations that occur at some of the statements processed by our algorithm. (The figure shows only a few illustrative examples.) The notation $\Gamma[x/y]$ represents the state with each syntactic occurrence of variable x replaced with y. Consider the state transformation at statement `x = r.f`. The updated state contains the predicates in the incoming state, with each predicate on `x` replaced with a predicate on `r.f`, and predicate $\langle r \neq \texttt{null}\rangle$.

Statement	State transformation	
`x = y`	$\Gamma' = \Gamma[x/y]$	
`x = r.f`	$\Gamma' = \Gamma[x/r.f] \cup \{\langle r \neq \texttt{null} \rangle\}$	
`r.f = x`	$\Gamma' = \Gamma[r.f/x] \cup \{\langle r \neq \texttt{null} \rangle\}$	
`if x op y`	$\Gamma' = \Gamma \cup \{\langle x \ op \ y \rangle\}$	(true branch)
	$\Gamma' = \Gamma \cup \{\langle \neg(x \ op \ y) \rangle\}$	(false branch)
`x = y op z`	$\Gamma' = \Gamma \setminus \Gamma[x]$	
`x = new`	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null} \rangle\}$	
`x = r.m()` (ext)	$\Gamma' = \Gamma[x/\texttt{null}] \cup \{\langle r \neq \texttt{null} \rangle\}$ if $x = \mathcal{R}_r \wedge null(x)$	(1)
	$\Gamma' = \Gamma \cup \{\langle x \neq \texttt{null} \rangle, \langle r \neq \texttt{null} \rangle\}$ if $x = \mathcal{R}_r \wedge \neg null(x)$	(2)
	$\Gamma' = \Gamma$ if $x \neq \mathcal{R}_r$	(3)
`x = r.m()` (app)	$\Gamma' = \sigma(r.m, \Gamma) \cup \{\langle r \neq \texttt{null} \rangle\}$	

Figure 3. State transformations at some of the statements. Γ represents the state following a statement; Γ' represents the state preceding a statement.

At the call to an application method, `x = r.m()`, the updated state consists of $\langle r \neq \texttt{null} \rangle$ and the transformation induced on Γ by `r.m()`. (Function σ computes the transformation of the given state by the given method.)

At a call to an external method, `x = r.m()`, the null-check rule is applied. There are three cases to consider. If `x` is the root reference and a null check was performed on `x` (indicated by $null(x)$ in Figure 3), the state is updated to replace the occurrences of `x` with `null` (i.e., the statement is treated as the assignment of a null value); additionally, $\langle r \neq \texttt{null} \rangle$ is added to Γ. If `x` is the root reference and no null check was performed on `x`, the statement is processed as the assignment of a new object (i.e., the transformation assumes that `r.m()` does not return a null); also, $\langle r \neq \texttt{null} \rangle$ is added to Γ. Finally, if `x` is not the root reference, the state is propagated with no changes.

3.3 Parametrized path exploration

Figure 4 presents the interprocedural path-exploration algorithm. It takes as inputs the statement, s_d, to start the traversal from and the variable, r_d, that is dereferenced at that statement. It returns as output the set of paths (and the associated predicates) over which a null value for r_d may flow to s_d.

The algorithm performs preprocessing (lines 1–5) to determine, for each method M, whether a null check is performed on a formal parameter of M or on a variable assigned the return value of a call to an external method; the null check may occur after transitive assignments either in M or in a method called directly or indirectly from M. This information is used to apply the null-check rule. Then, the algorithm initializes the state with predicates $\langle r_d = \texttt{null} \rangle$ and $\langle \texttt{this} \neq \texttt{null} \rangle$ and calls `analyzeMethod()` (lines 6–7).

Function `analyzeMethod()` uses a standard worklist-based approach to compute a fix-point solution over the abstract state predicates (lines 1–18). Because we abstract away integer arithmetic, the number of generated predicates is bounded, and the analysis is guaranteed to terminate.

```
algorithm NullDerefAnalysis
 input    $s_d$: dereference statement; $r_d$: variable dereferenced at $s_d$
 output   (path, state) pairs to dereference
 declare  $\mathcal{R}_r$: root reference; $\mathcal{R}_p$: root predicate; $CS$: call stack
   $null(r)$       true if a null check is performed on reference $r$
   $visited(s)$    set of states with which statement $s$ is visited
   $\mathcal{S}(M,\Gamma)$    summary information for method $M$ for state $\Gamma$;
                   consists of (path, state) pairs
begin
 1. foreach method $M$ in reverse topological order do
 2.     foreach reference $r$ that receives a value externally do
 3.         if there is a direct or indirect null check on $r$ then
 4.             $null(r)$ = true
 5.         else $null(r)$ = false
 6. $\Gamma = \{\langle r_d = \texttt{null}\rangle, \langle \texttt{this} \neq \texttt{null}\rangle\}$; path $\rho = s_d$
 7. return analyzeMethod($s_d, \rho, \Gamma$)
end

function analyzeMethod
 input    $s$: statement to start analysis from; $\rho$: path to $s$; $\Gamma$: state at $s$
 output   $T$: (path, state) pairs from method entry to $s$
 declare  $worklist$: list of $(s,\rho,\Gamma)$ triples; $T, T_p$: set of $(\rho,\Gamma)$ pairs
   $addWlist(s,\rho,\Gamma)$   adds $(s,\rho,\Gamma)$ to $worklist$; updates $visited(s)$, $\mathcal{R}_r$
begin
 1. $worklist = (s,\rho,\Gamma)$; $T = \emptyset$
 2. while $worklist \neq \emptyset$ do
 3.     remove $(s,\rho,\Gamma)$ from $worklist$
 4.     foreach predecessor $s_p$ of $s$ do
 5.         if ($s_p \neq$ call/entry) $\vee$ ($s_p$ calls an external method) then
 6.             compute $\Gamma'$ for the state transformation induced by $s_p$
 7.             if ($\Gamma'$ is consistent) $\wedge$ ($\Gamma' \notin visited(s_p)$) then
 8.                 $addWlist(s_p, \rho{\cdot}s_p, \Gamma')$
 9.         else if $s_p$ calls application method $M$ then
10.             if $\mathcal{S}(M,\Gamma) = \emptyset$ then // no summary exists
11.                 push $M$ onto $CS$ // analyze called method
12.                 $s_x$ = exit node of the CFG of $M$; $\Gamma'$ = map $\Gamma$ to $s_x$
13.                 $T_m$ = analyzeMethod($s_x, s_x, \Gamma'$); pop $CS$
14.                 foreach $(\rho_m, \Gamma_m) \in T_m$ do
15.                     add $(\rho_m, \Gamma_m)$ to $\mathcal{S}(M,\Gamma)$
16.             foreach $(\rho_m,\Gamma_m) \in \mathcal{S}(M,\Gamma)$ do $\Gamma'$ = map $\Gamma_m$ to $s_p$
17.                 $addWlist(s_p, \rho{\cdot}\rho_m{\cdot}s_p, \Gamma')$
18.         else add $(\rho,\Gamma)$ to $T$ // $s_p$ is entry
19. if $CS \neq \emptyset$ then return $T$
20. $T_p = \emptyset$
21. if this method is an entry method then
22.     foreach $(\rho,\Gamma) \in T$ do
23.         if $null(\mathcal{R}_r) \vee \mathcal{R}_p = \langle true \rangle$ then add $(\rho,\Gamma)$ to $T_p$
24. else foreach call site $s_c$ that calls this method do
25.     foreach $(\rho,\Gamma) \in T$ do $\Gamma'$ = map $\Gamma$ to $s_c$
26.         $T_m$ = analyzeMethod($s_c, \rho{\cdot}s_c, \Gamma'$); add $T_m$ to $T_p$
27. return $T_p$
end
```

Figure 4. The algorithm for null-dereference analysis.

Each element of the worklist is a triple (s, ρ, Γ), in which s is a statement, ρ is a path to s, and Γ is the state at the entry of s along path ρ. The function $addWlist()$ adds an element to $worklist$, and updates the root reference $\mathcal{R}_r$ and a visited map (to ensure that a node is not processed multiple times with the same state).

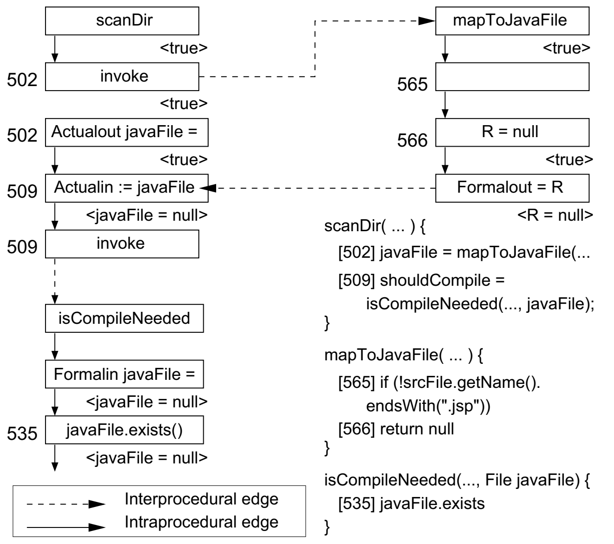
The function iteratively processes elements from the worklist (lines 2–18). For each element (s, ρ, Γ) removed from the worklist, the function examines each predecessor s_p of s. If s_p is not a call/entry or s_p is a call to an external method, the function updates the state (using the transformations shown in Figure 3), checks whether the updated state is consistent, and updates the worklist (lines 5–8).

Example 3. Consider the example in Figure 1. For the dereference of `t` at line 10 of method `bar()`, the algorithm creates predicates $\langle \texttt{t} = \texttt{null} \rangle$ and $\langle \texttt{this} \neq \texttt{null} \rangle$ (we omit the latter in the figure). At statement 9, the algorithm encounters a conflicting predicate $\langle \texttt{t} \neq \texttt{null} \rangle$. Along the backward path (10, 8), the condition at statement 8 generates predicate $\langle \texttt{y} \neq \texttt{null} \rangle$. Next, statement 7 transforms predicate $\langle \texttt{y} \neq \texttt{null} \rangle$ to $\langle \texttt{x} \neq \texttt{null} \rangle$. Finally, statement 6 transforms incoming root predicate $\langle \texttt{t} = \texttt{null} \rangle$ to $\langle \texttt{null} = \texttt{null} \rangle$, which is represented as $\langle \texttt{true} \rangle$.

To perform efficient analysis of called methods, the algorithm uses summary information for methods. The summary information for a method M maps a state Γ to a set of (ρ, Γ') pairs, which specifies how Γ is transformed along each path ρ in M. Using the summary information, the algorithm avoids analyzing a method multiple times for the same state. On reaching a call to an application method M, the algorithm first checks whether summary information exists for the current state (lines 9–10). If no summary exists, the algorithm descends into M to analyze it (lines 11–13). It uses a call stack to ensure context-sensitive processing of called methods.[4] After returning from M, the analysis saves the summary information for reuse in subsequent traversals (lines 14–15). Next, the algorithm updates the current state and path for the transformation induced by M, and adds an updated element to *worklist* (lines 16–17). On reaching the entry of the method, the algorithm collects (path, state) pairs (line 18).

If the current method is not being analyzed in a specific context (i.e., the call stack is empty) and the algorithm has reached an entry method, the algorithm determines which of the accumulated paths are true paths (lines 21–23). It classifies a path as a *true path* if a null check was performed on the root reference or the root predicate is $\langle true \rangle$. If an entry method is not reached, the analysis continues at each call site s_c that calls the method (lines 24–26).

Example 4. Consider the example from `Ant-1.5` shown in Figure 5, in which the dereference at line 535 is a potential null dereference. (This bug was reported in the Ant Bugzilla repository.) At line 535, the algorithm creates root predicate $\langle \texttt{javaFile} = \texttt{null} \rangle$. At the `FormalIn` statement, the algorithm creates a corresponding root predicate $\langle \texttt{javaFile} = \texttt{null} \rangle$ at the matching `ActualIn` at call statement 509. `javaFile` is defined at the `ActualOut` for the call to `mapToJavaFile()` at line 502. The algorithm adds the called method to the call stack and initializes the state at the `FormalOut` of `mapToJavaFile()` to root predicate $\langle \texttt{R} = \texttt{null} \rangle$, where `R` is a placeholder variable to represent the method return value. At line 566, the



```
scanDir( ... ) {
  [502] javaFile = mapToJavaFile(...
  [509] shouldCompile =
      isCompileNeeded(..., javaFile);
}
mapToJavaFile( ... ) {
  [565] if (!srcFile.getName().
      endsWith(".jsp"))
  [566] return null
}
isCompileNeeded(..., File javaFile) {
  [535] javaFile.exists
}
```

Figure 5. Example of interprocedural null dereference from `JspC.java` in `Ant 1.5`: `javaFile` can be potentially `null` at line 535.

`return null` statement assigns a `null` to `R`, which transforms the root predicate to $\langle \texttt{true} \rangle$. Line 565 generates more predicates, which are irrelevant to our discussion. At the top of the method, because the call stack is not empty, the algorithm ascends back to call site 502 to continue the analysis. On reaching the entry of `scanDir()`, because the root predicate is $\langle \texttt{true} \rangle$, the algorithm classifies path (502, 565, 566, 509, 535) as a true path.

Path exploration parameters. The algorithm performs a controlled path exploration: it uses three parameters that determine the extent to which different program paths and states are explored. The first parameter bounds the *traversal time* for a dereference. If the specified bound for traversal time is exceeded for a dereference, the algorithm aborts and ignores the dereference (i.e., it does not mark the dereference as either safe or unsafe). The second parameter bounds the *number of state predicates*. If the state limit is reached at a program point, the algorithm stops tracking additional predicates for that statement, but continues to analyze the program. The third parameter bounds the *number of true paths* collected at the entry of each method. If the path limit is reached at the entry of a method, the algorithm stops exploring new paths in the method, but continues to extend the current set of paths in the callers. The second and third parameters do not cause the analysis to abort—they only limit the number of predicates tracked, and the number of paths explored, by the algorithm. If the algorithm does not find a true path among the explored paths for the tracked predicates, it marks the dereference as safe.

These parameters enable a cost-accuracy trade-off: larger bounds can result in a more thorough program exploration and, therefore, more accurate results, but incur a

[4] A *context-sensitive* analysis propagates states along interprocedural paths that consist of valid call–return sequences only—the path contains no pair of call and return that denotes control returning from a method to a call site other than the one that invoked it.

higher cost. In Section 4.3, we present empirical data to illustrate the effects of the analysis parameters.

Limitations of the analysis. Although our approach performs a path-sensitive analysis to compute precise results, it has a few limitations. The algorithm does not handle reflection, dynamic class loading, or concurrency. To handle recursion, the implementation processes a recursive call as if it were a call to an external method and, therefore, applies the null-check rule on encountering such a call. Another limitation of the current implementation is that it does not perform interprocedural exception-flow analysis.

3.4 Abstract state predicates

One of the novel features of our approach is the use of a simple and efficient abstract predicate domain (Figure 2) that is remarkably effective in enabling the analysis to compute very few false positives.

Integer predicates. Although our approach does not use a constraint solver to handle arithmetic, it uses rules for establishing the validity of integer predicates. For example, given an existing predicate $\langle x < \texttt{const}_1 \rangle$ and a new predicate $\langle x = \texttt{const}_2 \rangle$, if $\texttt{const}_1 > \texttt{const}_2$, the predicate evaluator returns $\langle x = \texttt{const}_2 \rangle$; otherwise, it reports a conflict. The evaluator can determine for the following code that `x = null` implies `y = 0`; therefore, in the loop header, `i < y` is false and the loop body does not execute.

```
if (x == null) y = 0;
for (i = 0; i < y; i++) x.foo();
```

The limitations of not handling integer arithmetic can affect the accuracy of the algorithm. For example, for the following code, although XYLEM associates `y = 1` with `x == null`, it is unable to evaluate `i < y-1`. It conservatively assumes that the predicate may be true or false and incorrectly reports a potential null dereference for `x.foo()`.

```
if (x == null) y = 1; else y = z+1;
for (i = 0; i < y-1; i++) x.foo();
```

Our approach performs additional processing to accommodate dual-variable predicates and instanceof predicates, which improves the accuracy of the analysis.

Dual-variable predicates. At each statement, any new predicate has to be validated against the existing predicates. In general, it is easier to validate single-variable predicates than dual-variable predicates. Therefore, wherever possible, our analysis converts a dual-variable predicate into one or more single-variable predicates using the following rules:

- For a dual-variable predicate, $\langle x_1 = x_2 \rangle$, for each existing predicate $\langle x_1\ \textit{relOp}\ \texttt{const} \rangle$, the approach creates a predicate $\langle x_2\ \textit{relOp}\ \texttt{const} \rangle$ and vice versa.
- For a dual-variable predicate, $\langle x_1 \neq x_2 \rangle$, for each existing predicate $\langle x_1 = \texttt{null} \rangle$ (or, $\langle x_1 = \texttt{true} \rangle$), the approach creates a predicate $\langle x_2 \neq \texttt{null} \rangle$ (or, $\langle x_2 \neq \texttt{true} \rangle$) and vice versa.

Table 1. Subject programs used in the empirical studies.

Subject	Classes	Methods	Bytecode instructions
Ant 1.5	667	7095	168011
Lucene 1.9	272	2410	58653
Tomcat 4.1.20	216	4223	98361
App-A	204	3561	77867
App-B	2015	14886	340567
App-C	154	2504	46286

Although these rules have limited applicability, in our experience, they are effective in removing false positives. Consider the following example.

```
[1] b = x.foo();  {<x != null>,<y = null>,<x = null>}
[2] if (x == y)   {<y = null>, <x = null>}
[3]   y.bar();    {<y = null>}
```

For the dereference at line 3, our analysis creates an initial predicate $\langle \texttt{y} = \texttt{null} \rangle$. At line 2, instead of generating the dual-variable predicate $\langle \texttt{x} = \texttt{y} \rangle$, the approach creates $\langle \texttt{x} = \texttt{null} \rangle$. Then, at line 1, the dereference of `x` generates predicate $\langle \texttt{x} \neq \texttt{null} \rangle$, which conflicts with $\langle \texttt{x} = \texttt{null} \rangle$ and establishes that the dereference in line 3 is safe.

Instanceof predicates. Our algorithm does not track instanceof predicates, but it leverages a useful property of such predicates that they can indicate whether a reference might be non-null: if `(x instanceof T)` is true, `x` is not null. Therefore, along the true branch of such a conditional, the algorithm generates $\langle \texttt{x} \neq \texttt{null} \rangle$, which can potentially remove false positives. In the following example, the dereference of `ta` at line 3 is safe because two conflicting predicates are generated at the conditional at line 1.

```
[1] if (el instanceof T){ {<el != null>,<el = null>}
[2]   T ta = (T) el;      {<el = null>}
[3]   ta.get();           {<ta = null>}
```

4 Empirical Evaluation

To evaluate our approach, we implemented the null-dereference analysis and conducted empirical studies using three open-source projects and three commercial products (referred to as `App-A`, `App-B`, and `App-C`). Table 1 lists the subject programs, along with the number of classes, methods, and bytecode instructions in each subject. The null-dereference analysis is implemented in our tool XYLEM, using the WALA analysis infrastructure.[5] The XYLEM analysis is performed in two steps. In the first step, points-to analysis, escape analysis, and control-dependence analysis is performed; the CFG, along with the minimal necessary dependence information, is persisted on the disk. In the second step, the persisted data is read in blocks and null-dereference analysis is performed.

We conducted four empirical studies to evaluate: (1) accuracy, (2) efficiency, (3) effects of analysis parameters, and

[5]http://wala.sourceforge.net

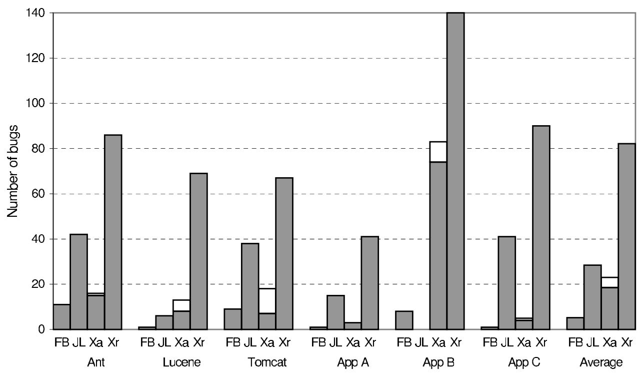


Figure 6. The number of bugs detected by FINDBUGS, JLINT, XYLEM-intra, and XYLEM-inter. The lighter segment in the bar for XYLEM-intra represents the percentage of bugs invalidated by interprocedural analysis.

(4) relevance of the detected bugs. Unless otherwise specified, we set the analysis parameters as follows: traversal time limited to 2 seconds, the state size limited to 80 predicates, and a maximum of 7 true paths per method. We selected these values based on multiple runs of the tool, during development, on a training sample that included `Ant` and 12 other applications not included (to avoid bias) in our study. For each of the studies, we ran XYLEM in the batch mode, in which all dereferences in a subject were analyzed. We ran our experiments on an Intel-based Linux machine (3 GHz, dual core, 7GB RAM).

4.1 Accuracy

Goals and method. To assess the accuracy of our algorithm, we computed the number of bugs reported by FINDBUGS, JLINT, XYLEM. We ran XYLEM in two modes: an intraprocedural mode (XYLEM-intra) in which the tool does not analyze called or calling methods; and an interprocedural mode (XYLEM-inter), in which the tool performs the analysis described in section 3. We manually examined the defects reported by FINDBUGS, JLINT, and XYLEM-inter for `Ant` to identify false positives. We also determined the number of interprocedural bugs identified by the three tools and whether the XYLEM-inter true positives included all the FINDBUGS and JLINT true positives.

Results and analysis. Figure 6 presents the data about accuracy. It contains four bars for each subject: from left to right, the bars represent FINDBUGS, JLINT, XYLEM-intra, and XYLEM-inter. The last set of bars represent the average over the subjects.[6] The vertical axis represents the number of bugs. For example, for `Ant`, FINDBUGS and JLINT reported 11 and 42 bugs, respectively (indicated by the first and second bars); XYLEM-intra found 16 bugs, and XYLEM-inter reported 86 bugs. The data show that

[6]JLINT could not analyze `App B`; therefore, the average for JLINT is over five subjects only.

Table 2. Number of bugs and equivalence classes.

	Ant	Lucene	Tomcat	App-A	App-B	App-C
No. Bugs	86	68	65	42	140	91
No. Classes	32	22	24	21	97	25

consistently across the subjects FINDBUGS reports the least number of bugs; JLINT detects more bugs than FINDBUGS; and XYLEM-inter detects significantly more bugs than both these tools. On average, XYLEM-inter detects 16 times as many bugs as FINDBUGS and nearly three times as many bugs as JLINT.

The lighter segment in the XYLEM-intra bar represents the percentage of intraprocedural bugs that were invalidated by interprocedural analysis. The following example from class `IContract` in `Ant` illustrates such a bug.

```
[506] public void execute() throws BuildException {
[509]   preconditions();
[514]   boolean useCtrlFile=(ctrlFl != null);
[615]   if (updIctrl) {
[626]     ctrlFl.getAbsolutePath());

[655] private void preconditions()
[671]   if (updIctrl == true && ctrlFl == null) {
[672]     throw new BuildException("...
```

The intraprocedural analysis of `execute()` reports a potential null dereference of `ctrlFl` at line 626 because of the null check at line 514. However, the interprocedural analysis determines that, if `ctrlFl` is null and `updIctrl` is true, an exception is thrown at line 672, which causes line 626 to not be reached. Therefore, if `updIctrl` is true at line 626, `ctrlFl` must be non-null.

On average, less than 5% of the intraprocedural bugs were invalidated by the additional constraints computed by the interprocedural analysis.

Manual inspection of all the reported bugs for `Ant` showed that XYLEM and FINDBUGS identified very few false positives, whereas JLINT identified many false positives. FINDBUGS reported only one false positive in `Ant`. This is not surprising as FINDBUGS is known to remove false positives aggressively. However, the number of true positives is very low too, which lowers the usefulness of the tool. The false positive found by FINDBUGS shown below, exemplifies the difference between our null heuristic and that of FINDBUGS. XYLEM applies the heuristic only for methods unavailable for analysis, whereas FINDBUGS assumes that `sections` may be null at line 565 because of the null check at line 553.

```
[542] final ArrayList sections = new ArrayList();
[553] if (null != section ) { ... }
[565] sections.clear()
```

Of the 42 bugs reported by JLINT, 32 (more than 75%) were false positives. XYLEM-inter reported four false positives (less than 5% of the 86 bugs).

Of the 11 bugs identified by FINDBUGS, one was an interprocedural bug, which was a true positive; JLINT reported

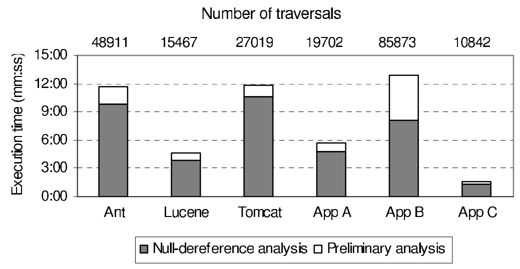


Figure 7. Execution time for null-dereference analysis and preliminary analysis.

Table 3. Complete, partial, and aborted traversals.

Subject	Number of Traversals	Complete traversals	Incomplete traversals: Partial traversals	Incomplete traversals: Aborted traversals
Ant	48911	46461 (94.99)	1907 (3.90)	543 (1.11)
Lucene	15467	14807 (95.73)	366 (2.37)	294 (1.90)
Tomcat	27019	25663 (94.98)	689 (2.55)	667 (2.47)
App-A	19702	19176 (97.33)	316 (1.60)	210 (1.07)
App-B	85873	85024 (99.01)	619 (0.72)	230 (0.27)
App-C	10842	10521 (97.04)	285 (2.63)	36 (0.33)
Average	34636	33609 (97.03)	697 (2.01)	330 (0.95)

seven interprocedural bugs, of which five were false positives. XYLEM-inter reported 66 true-positive interprocedural bugs. Additionally, XYLEM-inter reported each true positive (intra or interprocedural) reported by either FINDBUGS or JLINT, and invalidated each false positive reported by FINDBUGS and JLINT.

Although, we did not observe this in `Ant`, XYLEM-inter can report false positives that result from the limitation of not using a constraint solver to handle arithmetic (as discussed in Section 3.4). Given the simplicity, and the resulting efficiency, of our predicate handling, it is encouraging to find such a low false-positive rate.

Discussion. Overall, the data illustrate that our interprocedural analysis detects significantly more bugs than an intraprocedural or a limited interprocedural analysis. In addition, our approach could eliminate many of the false positives that may be identified by FINDBUGS or JLINT.

We have found that, often, the null value generated at a statement is dereferenced at several statements. In such cases, it is useful to group the bugs into equivalence classes based on the statement at which the null value originates. This makes it easier for the developer to browse the bug reports. In fact, in many cases, it may be sufficient to investigate one bug per class, which reduces the burden on the developer. Table 2 shows the number of bugs and equivalence classes for our subjects. The number of classes ranges from 27% to 69% of the number of bugs.

4.2 Efficiency

Goals and method. To evaluate efficiency, we collected data about the total analysis time, which consists of the time required to perform the preliminary analysis and the time required to perform the null-dereference analysis. We investigated another aspect of efficiency that is indicated by how a traversal terminates. A *complete traversal* is one in which the algorithm either finds a true path or finds only false paths after exploring all paths to the dereference. A *partial traversal* is one in which the path or state limits prevent the algorithm from performing complete path/state exploration, and the algorithm finds only false paths in the explored parts of the program. An *aborted traversal* is one in which the time limit is reached. For each subject, we computed the number of complete, partial, and aborted traversals.

Results and analysis. Figure 7 shows the execution times (in minutes) for the preliminary analysis and the null-dereference analysis. For each subject, the number at the top represents the number of traversals that were performed for the subject—one traversal starting at each dereference statement in the subject. For example, for `Ant`, the 48911 traversals completed in less than 12 minutes, of which the null-dereference analysis required approximately 10 minutes; the remaining time was taken by the preliminary analysis. The largest number of traversals (over 85000) were performed for `App-B`, which took 8 minutes.

Table 3 presents the data about the traversals: it lists the total number of traversals (column 2), and the number and percentage of traversals that were complete, partial, or aborted (columns 3–5). On average, the two-second time limit caused less than 1% of the traversals to abort. Partial traversals, caused by path and state limits, occurred more frequently, but, overall, were about 2%. For three subjects, more than 97% of the traversals were complete; for the remaining three, approximately 95% of the traversals completed.

Discussion. The data demonstrate the efficiency of the analysis: our analysis required no more than 11 minutes for any subject and, on average, performed complete traversals for more than 97% of the dereferences.

To date, the largest application analyzed by XYLEM, in deployment with a product team in IBM, contains over 1,009,000 lines of code. The analysis completed in approximately 3 hours and 30 minutes. Such performance is acceptable because defect-finding tools do not have stringent real-time constraints and are, typically, run overnight on a new build.

4.3 Effects of analysis parameters

Goals and method. In the third study, we investigated the effects of the three analysis parameters (traversal time, number of predicates, and number of true paths) on accuracy and efficiency. There is a trade-off involved in increasing the value of a parameter as it may cause the limits to be

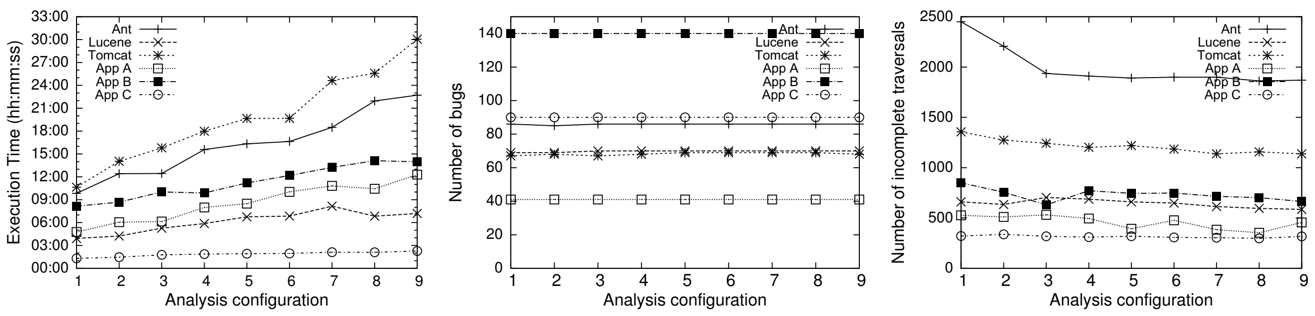


Figure 8. Effects of nine analysis configurations on total execution time (left), number of bugs (middle), and incomplete traversals (right). In the nine configurations: the traversal time increases in 0.5-second increments from 2 to 6 seconds; the state size increases from 80 to 160 predicates in increments of 10; the number of paths increases from 7 to 15 in increments of 1.

reached for the other two parameters more often. For example, increasing the time limit could decrease the number of aborts, which is desirable. However, it could also cause the path and predicate limits to be reached more often, thereby increasing the number of partial traversals.

For this study, we created nine analysis configurations by successively increasing the values of each parameter in fixed increments. We increased (1) the traversal time in 0.5-second increments from 2 seconds to 6 seconds, (2) the number of predicates from 80 to 160 predicates in increments of 10, and (3) the number of paths from 7 to 15 in increments of 1. For each analysis configuration, we ran XYLEM and collected data about the total execution time, the number of bugs, and the number of incomplete traversals (the sum of partial and aborted traversals).

Results and analysis. Figure 8 presents the data about the effects of the nine configurations. The plot on the left shows that the execution time increases for each successive configuration, but not significantly. Even for the last configuration—which corresponds to traversal time of 6 seconds, state size of 160 predicates, and 15 true paths—the analysis required no more than 30 minutes to complete for any of the subjects. The maximum increase from configuration 1 to configuration 9 occurred for `Tomcat`, from about 10 minutes to 30 minutes; `App-A` too showed a three-fold increase in execution time. The remaining four subjects showed a two-fold increase or less. (The plot shows a decrease in the execution time from configuration 7 to 8 for `Lucene` and `App-A`, which can be attributed to variations in the CPU load on our server.)

The middle plot shows the effects on the number of bugs. Clearly, there is no significant gain in the number of bugs detected. Beyond the first configuration (which is the default configuration), a total of only three additional bugs were detected: one bug in `Lucene` at configuration 3; and two bugs in `Tomcat`, one at configuration 2 and another at configuration 5.

The plot on the right shows the effects on the number of incomplete traversals. The general trend shows a very gradual decrease in the number of incomplete traversals. The maximum decrease occurs for `Ant`, from 2450 to 1869 incomplete traversals. However, the additional 581 complete traversals that resulted from this reduction detected no additional bug. For the other subjects, the reduction in incomplete traversals ranged from 1.2% for `App-C` to 21.8% for `App-B`. In a few cases, the number of incomplete traversals increased from one configuration to the next (e.g., from configuration 3 to 4 for `App-B`), which we suspect to be caused by variations in the server load too. Because the analysis executor measures traversal time in terms of the real time (and not system time), a high CPU load can cause more aborts even when the traversal time is increased. We leave it to future work to perform more rigorous investigation of the effects of analysis parameters.

Discussion. The data indicate that, although a larger-scope analysis is feasible, there is no significant gain in the number of bugs. To validate this further, we ran XYLEM on `App-C`, with all limits turned off: the analysis completed in 48 minutes, with neither aborts nor partial traversals, but no additional bug was detected. However, for the other subjects, this did not prove to be successful—the analysis ran into insufficient memory problems. This also indicates the trend that the few remaining dereferences for which the analysis is incomplete require expensive computation.

Future research can investigate how the memory problem might be overcome. For example, an approach might be to not save and reuse summary information across traversals. Although this could increase the execution time significantly, the memory overhead would decrease; consequently, more traversals might complete.

4.4 Relevance of the detected bugs

Goals and method. In the next study, we evaluated the relevance of XYLEM-detected bugs by investigating how often these bugs get deleted in subsequent code versions. We ran XYLEM on five releases each of the open-source subjects and two builds each of the commercial products. Next, for each pair of versions, we identified the bugs that appeared

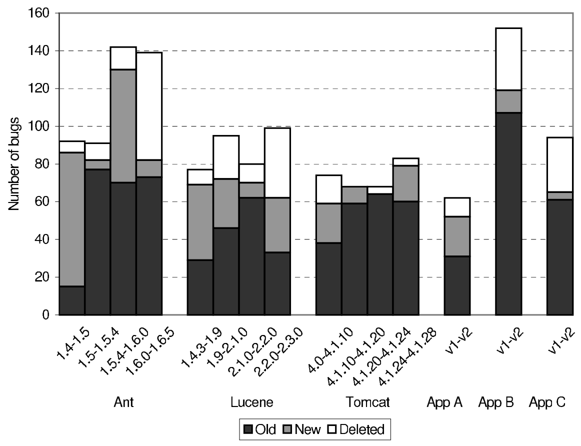


Figure 9. Old, new, and deleted bugs between pairs of versions.

in (1) both versions (*old bugs*), (2) the second version only (*new bugs*), and (3) the first version only (*deleted bugs*). We used the number of deleted bugs as an indicator of the relevance of the identified bugs.

Results and analysis. Figure 9 presents data about the differences in bugs for pairs of versions of our subjects. There are four pairs of versions for each open-source project, and one pair each for the commercial products. Each pair is represented by a segmented bar, in which the segments represent old bugs, new bugs, and deleted bugs. The height of a bar represents the union of the bugs for the corresponding version pair. For example, for pair 1.6.0–1.6.5 of `Ant`, a total of 139 bugs are reported. `Ant-1.6.0` has 130 reported bugs, of which 57 got deleted in `Ant-1.6.5`; `Ant 1.6.5` has 82 reported bugs, of which 73 bugs are reported in the previous version too. On average over the versions of `Ant`, 26% of the reported bugs got deleted; for `Lucene` and `Tomcat`, 31% and 9% of the bugs got deleted, respectively. The percentage of deleted bugs for the commercial products varies from 23% (`App-A`) to 32% (`App-C`). Over all subjects and versions, approximately 24% of the bugs reported by XYLEM got deleted.

In contrast, for FINDBUGS, two reported bugs got deleted across all versions of `Ant`, none in `Tomcat`, one each in `Lucene`, `App-A`, and `App-C`, and six in `App-B`—a total of 11 deleted bugs (10%) compared to 257 deleted bugs detected by XYLEM.

Discussion. The results indicate that a fairly large percentage of bugs got deleted, which suggests that the detected bugs may be important. However, our study leaves open questions about a postmortem analysis: a bug might be deleted because of unrelated program changes that are not intended to fix a bug. For example, a change in a class hierarchy could delete a null-propagation path to a dereference. Investigating the effects and types of unintentional changes is an interesting topic for future research.

5 Related Work

In previous work, we investigated a *sound* null-dereference analysis, in a tool called SALSA. The purpose of SALSA is verification, i.e., to identify dereferences that are definitely non-null; the remaining ones are suspect. The metric of success is the reduction in the number of suspect references by aggressive static analysis. XYLEM focuses on bug detection instead of verification. The metric of success is to offer "useful" defect reports, but without attempting to be sound. These tools represent different trade-offs and can be complementary.

At a technical level, the two tools implement different approaches for interprocedural analysis. SALSA implements a forward iterative analysis, where the analysis scope is expanded incrementally to be able to prove more dereferences safe. By contrast, XYLEM implements a backward demand-driven strategy to find whether there is sufficient basis to report a dereference as a possible bug. We believe that for bug detection, the backward demand-driven approach is better in terms of scalability than a forward iterative one. Furthermore, because of parametrized path exploration, useful bug reports can be produced even in a partial, time-bound analysis run, which is attractive from a developer's point of view.

Tomb et al. [15] present a symbolic-execution-based analysis that is parametrized with respect to the call depth to explore. In addition to null dereferences, their analysis detects other types of bugs, such as invalid type casts. They conclude that although interprocedural analysis is useful for reducing false positives (by generating more constraints), it does not detect noticeably more bugs than an intraprocedural analysis. For null dereferences, our results contradict their conclusions, and suggest that a limited interprocedural null-dereference analysis may be of little value.

Engler et al. [5] introduced the notion of detecting bugs based on contradictory programmer beliefs, which are implied in the code. Our null-check rule uses a similar idea, but only for those dereferences that receive unknown values from outside the application being analyzed. Dillig et al. [4] formalized the notion of consistency checking by relating it to type inference.

Several approaches for accurate interprocedural null-dereference analysis have been developed for the C language (e.g., [2, 3, 16]). Bush et al. [3] and Xie et al. [16] present similar approaches for detecting a broad class of memory errors. Their approaches feature a bottom-up analysis of procedures to compute summaries, and a forward path-sensitive analysis within each procedure that prunes out infeasible paths. In contrast, our approach is a demand-driven backward analysis.

Other approaches for improving the accuracy of static analysis use programmer annotations (e.g., [7, 10]), use heuristics to avoid reporting certain types of warnings (e.g., [10]), or build predictive models from historical data (e.g., [11, 13]). These approaches are complementary to our approach: they can be used to prioritize the bugs identified by our approach.

Other researchers (e.g., [1, 14]) have investigated the evolution of bugs over successive versions of applications. For example, Ayewah et al. [1] manually examined the bugs that were removed during the development of JDK 1.6.0. They classified a bug as fixed if it was reported in a build, not reported in the next build, and the class containing the bug was present in the second build. They found that more than half of the bug removals were because of small targeted changes that seemed to be focused on fixing the bug.

6 Summary and Future Work

We have presented an accurate analysis for detecting null-dereference bugs in Java programs. Our analysis detects bugs that many commonly used tools miss, and it eliminates the false positives that other tools report. For the subjects that we studied, our approach detected significantly more bugs than tools that perform limited interprocedural analysis. Our studies also illustrated the efficiency of the analysis and the relevance of the detected bugs. Our tool has been deployed with several product-development teams in IBM, and preliminary feedback has been positive about the value of the tool in detecting bugs that are worth fixing.

Although our studies demonstrate the value of our approach, they leave open questions, which could be investigated in future work. In our current studies, we have not compared our approach with a sound analysis, such as the one implemented in SALSA. Such a study would indicate the extent to which our approach may miss potential bugs. Future studies could also investigate how our results generalize to more subjects, perform a rigorous evaluation of the effects of analysis parameters, and explore implementation optimizations that could reduce the incomplete traversals.

Another area of future work is prioritization of true positives, which is an important aspect of improving the usefulness of static-analysis tools. This may involve combining static and dynamic analysis and building predictive models based on historical data. Finally, future work could extend our analysis to other types of bugs, notably resource-leak bugs (e.g., failure to close database connections).

Acknowledgements

We would like to thank Satish Chandra and Mary Jean Harrold for feedback on earlier drafts of the paper, and the reviewers for their comments, that have improved the presentation significantly.

References

[1] N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. In *Proc. of 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Softw. Tools and Eng.*, pages 1–8, June 2007.

[2] D. Babić and A. J. Hu. Calysto: Scalable and precise extended static checking. In *Proc. of the 30th Intl. Conf. on Softw. Eng.*, pages 211–220, May 2008.

[3] W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Software—Practice and Experience*, 30(7):775–802, June 2000.

[4] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proc. of the 2007 ACM SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 435–445, June 2007.

[5] D. Engler, D. Y. Chen, S. Hallem, A. Chou, and B. Chelf. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proc. of the 18th ACM Symp. on Operating Syst. Principles*, pages 57–72, Oct. 2001.

[6] D. Evans. Static detection of dynamic memory errors. In *Proc. of the ACM SIGPLAN 1996 Conf. on Prog. Lang. Design and Impl.*, pages 44–53, May 1996.

[7] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Prog. Lang. Design and Impl.*, pages 234–245, June 2002.

[8] D. Hovemeyer and W. Pugh. Finding bugs is easy. *ACM SIGPLAN Notices (Proceedings of Onward! at OOPSLA 2004)*, 39(10):92–106, Dec. 2004.

[9] D. Hovemeyer and W. Pugh. Finding more null pointer bugs, but not too many. In *Proc. of 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Softw. Tools and Eng.*, pages 9–14, June 2007.

[10] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *Proc. of 6th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Softw. Tools and Eng.*, pages 13–19, Sept. 2005.

[11] T. Kremenek and D. Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proc. of the 2003 Static Analysis Symp.*, pages 295–315, June 2003.

[12] A. Loginov, E. Yahav, S. Chandra, N. Fink, S. Rinetzky, and M. G. Nanda. Verifying dereference safety via expanding-scope analysis. In *Proc. of the 2008 Intl. Symp. on Softw. Testing and Analysis*, pages 213–223, July 2008.

[13] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: An experimental approach. In *Proc. of the 30th Intl. Conf. on Softw. Eng.*, pages 341–350, May 2008.

[14] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *Proc. of Intl. Workshop on Mining Softw. Repositories*, pages 133–136, May 2006.

[15] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *Proc. of the 2007 Intl. Symp. on Softw. Testing and Analysis*, pages 97–107, July 2007.

[16] Y. Xie, A. Chou, and D. Engler. ARCHER: Using symbolic, path-sensitive analysis to detect memory access errors. In *Proc. of the 9th European Softw. Eng. Conf. / 11th ACM SIGSOFT Intl. Symp. on Found. of Softw. Eng.*, pages 327–336, Nov. 2003.