

February 23, 2007

Finding More Null Pointer Bugs, But Not Too Many

David Hovemeyer

Dept. of Physical Sciences
York College of Pennsylvania
dhovemey@ycp.edu

William Pugh

Dept. of Computer Science
Univ. of Maryland
pugh@cs.umd.edu

Abstract

In the summer of 2006, the FindBugs project was challenged improve the null pointer analysis in FindBugs so that we could find more null pointer bugs. In particular, we were challenged to try to do as well as a publicly available analysis by Reasoning, Inc on version 4.1.24 of Apache Tomcat. Reasoning uses their own static analysis tool, followed by manual auditing and removal of false positives before a report is given to their customers. Reasoning reported a total of 9 null pointer warnings in Tomcat 4.1.24, of which only 2 were reported by FindBugs 1.0. While we wanted to improve the analysis in FindBugs, we wanted to retain our current low level of false positives, and not depend on manual suppression of false positives before getting high quality results we were happy to show to developers.

As of result of the work presented in this paper, FindBugs now reports 4 of the 9 warnings in Tomcat, shows that one of the warnings reported by Reasoning is a false positive, and classifies the remaining 4 as being dependent on the feasibility of a particular path, which cannot be easier ascertained by a local examination of the source code. Moreover, we found 24 additional null pointer bugs in Tomcat that had been missed by Reasoning, and overall doubled the number of null pointer bugs found by FindBugs while improving the quality and significance of the defects we do report.

1. Introduction

In the summer of 2006, David Morgenthaler challenged the FindBugs project to improve the null pointer analysis in FindBugs. David Morgenthaler had been previously employed at Reasoning, a company that provided static analysis for software defect detection as a service. Dr. Morgenthaler was now employed by Google, where among other duties he was evaluating static defect detection tools for use at Google. Reasoning had previously made public [7] a sample of the output of their services on version 4.1.24 on Apache Tomcat [1]. Reasoning had applied their own static analysis tool and after filtering out the warnings they believed to be false positives reported 9 null pointer bugs. FindBugs 1.0 reported only two of the warnings reported by Reasoning. For some time we had wanted to improve our analysis to find more defects, and this would provide additional motivation and test cases.

The static analysis tool used by Reasoning could have a much higher false positive rate than was acceptable for FindBugs, since they performed internal manual suppression of false positives before results were shown to customers. In particular, the issue of infeasible paths is a standard problem for static analysis. Consider, for example:

```
PrintWriter log = null;  
if (anyLogging) log = new PrintWriter(...);  
if (detailedLogging) log.println("Log started");
```

This code will throw a null pointer exception if `anyLogging` is false and `detailedLogging` is true. The names of these boolean variables suggest that this situation cannot arise at runtime, but verifying that through static analysis might be difficult. As we can see in this example, the infeasible path problem involves a path through the program that depends on at least two branches. Even if both branches can be decided in both directions (true and false), they both have to be decided in a correlated way in order for the potential null-pointer bug to manifest itself. To address this problem, static analysis tools can compute *path conditions*—the conditions that must be true in order for the path to be executed. If the analysis can show that the path conditions are infeasible, then the potential defect is also shown to be infeasible. In the following modification of the above example, the path condition for the null pointer exception is `logLevel ≤ 0 ∧ logLevel > 3`, which can be shown to be infeasible by appropriate techniques.

```
PrintWriter log = null;  
if (logLevel > 0) log = new PrintWriter(...);  
if (logLevel > 3) log.println("Log started");
```

Unfortunately, many infeasible path conditions are not easy to disprove. Rather than worrying about disproving path conditions, FindBugs 1.0 used a null pointer analysis that reported a warning if there was a statement or branch in a program that, if executed/taken, would guarantee that a null pointer exception would occur. Since programmers strive to avoid writing unreachable code and redundant conditional tests, the existence of such a statement or branch is a good indicator of programming defect. If the statement or branch is unreachable, then that is generally taken as evidence of a programming fault that warrants correction, to improve the performance and understandability of the code.

In this paper, we describe several techniques we developed and implemented in more recent versions of FindBugs in order to find more null pointer bugs, while retaining the property that null pointer warnings are only emitted if a branch or statement exists that if executed would guarantee a null pointer exception. Combined with strengthened analysis to track null values in fields, these improvements roughly double the number of null pointer defects found by FindBugs. Our improved analysis is substantially better than analysis based on proving the feasibility of path conditions at producing easy-to-understand warnings.

Going back to the challenge from Dr. Morgenthaler, we found 2 more of the null pointer bugs reported by Reasoning and disproved one of them. The remaining 4 are dependent upon the feasibility of path conditions that cannot be easily checked.

2. Reasoning report on Tomcat

Table 1 gives the null pointer warnings reported [7] by Reasoning. Figures 1, 2, and 3 show part of the code for the defects numbers

```

322 while(it.hasNext()) {
323     Object elt = it.next();
324     if((null == obj && null == elt)
        || obj.equals(elt)) {
325         count++;
326     }

```

Figure 1. Defect 01

```

519 HttpServletRequest hreq = null;
520 if (req instanceof HttpServletRequest)
521     hreq = (HttpServletRequest) req;
522
523 if (isResolveHosts())
524     result.append(req.getRemoteHost());
525 else
526     result.append(req.getRemoteAddr());
...
551 result.append(hreq.getMethod());

```

Figure 2. Defect 08

```

889 HttpServletRequest hreq = null;
890 if (req instanceof HttpServletRequest)
891     hreq = (HttpServletRequest) req;
892
893 switch (type) {
...
905     case 'c':
906         Cookie[] c = hreq.getCookies();

```

Figure 3. Defect 09

01, 08, and 09, respectively, by Reasoning. The defect numbers are those assigned by Reasoning, which also include 3 array bound errors, 12 resource leaks and 2 bad string comparisons.

Figure 1 is a relatively simple bug that was found by the null pointer analysis in FindBugs 1.0 [5]: if the test `null == elt` evaluates to false, we are guaranteed to get a null pointer exception when we dereference `obj` by invoking the `equals` method on it.

Figure 2 is a bug we want to report, but the analysis in [5] doesn't. If the test on line 520 fails, we are guaranteed to get a null pointer exception at line 551 (assuming no other exceptions prevent us from reaching line 551). However, there is no simple path¹ from the else branch of line 520 to line 551. The conditions at lines 523, 530 and 532 mean that there are 8 different paths from line 520 to line 551, and each of these paths have path conditions that traditional analysis would try to show are feasible or infeasible.

Figure 3 is a warning we do not wish to report in FindBugs. This null pointer exception can occur only if it is simultaneously possible for `req` to not be a `HttpServletRequest` and for `type = 'c'`. This is exactly the kind of infeasible path problem that we were concerned about. Clearly, this defect should be prioritized below defects 01 and 08, and at this point, we don't want to report this potential defect in FindBugs at all.

3. Improving null analysis

While the most significant changes to our analysis were partially motivated by the challenge of improving our results on Tomcat, we

¹We define a *simple path* to be one in which there are no conditional branches.

actually made quite a number of changes to our null pointer analysis since FindBugs 1.0. This section summarizes them, notes the impact on the number of warnings we report on Eclipse and Sun's 1.6 JVM implementation, and notes the connection to improving our results on Tomcat.

3.1 Warnings removed

FindBugs no longer reports a number of warnings reported by FindBugs 1.0. There are four basic reasons for this:

- **Suppression of warnings inside try/catch blocks** We found a number of cases where code could perform a null pointer dereference, but the code was inside a small try/catch block that was designed to anticipate and handle that situation.
- **Better handling of assertions.** In particular, FindBugs now analyzes code as if assertion checking is always enabled. This means that we will treat `assert x != null;` as an assertion that `x` is nonnull, since the analysis understands the successor of that statement can be reached only if `x` is nonnull.
- **Better handling of panic methods.** We've seen a fair number of false positives caused by code such as `if (x == null) panic();` followed by a dereference of `x`. Unless FindBugs understands that the `panic()` method won't return normally, it will report a null pointer defect here. We can't simply match on the name `panic`, since different projects use different names for such methods. We also unfortunately can't depend on only checking to see if the method returns normally, because some systems have methods that log a fatal error message and then return normally. This is a place where we have done some custom modeling of specific panic methods we have encountered in different frameworks, but encourage users of FindBugs to improve their analysis results by specifying any unique panic methods used in their own codebase.
- **Better modeling of the JDK.** FindBugs incorporates a model of which method parameters and return values in the core API methods must be nonnull. Violations of these API specifications—such as passing a possibly-null value to a method requiring a nonnull parameter, or returning a possibly-null value from a method declared to return a non-null value—are reported as warnings. An improved modeling of which JDK methods always return nonnull values eliminated some false positives. For one package of the JDK, `java.util.concurrent`, we have modeled all the method parameters as taking nonnull parameters unless explicitly specified otherwise. Additional methods were introduced into that package after FindBugs 1.0 were released, and thus FindBugs 1.0 wrongly marked some b105 methods as requiring nonnull parameters. This is something of an artificial case, due to the explicit modeling of the JDK in FindBugs.

3.2 Warnings reclassified

FindBugs 1.0 reported a null pointer correctness bug if the `equals(Object)` method would throw a null pointer exception if given a null argument (it should return false). In FindBugs 1.1, this was reclassified as a bad practice bug, rather than a correctness bug. While it is fairly easy to make the case that this is incorrect code, we found that developers didn't want to wade through these warnings when performing correctness bug triage. In general, each project or organization should adopt a rule as to whether this bad practice is acceptable. Similarly, we also reclassified `clone()` and `toString()` methods returning a null value as bad practices rather than correctness bugs.

| # | FB 1.0 | FB 1.1+ | library | File | line |
|----|--------|--------------------|---------------------------|---------------------------|------|
| 01 | yes | yes | commons-collections-2.1 | CollectionUtils.java | 324 |
| 05 | no | no, path-dependent | tomcat-4.1.24 | StandardWrapperValve.java | 185 |
| 08 | no | yes | tomcat-4.1.24 | AccessLogValve.java | 551 |
| 09 | no | no, path-dependent | tomcat-4.1.24 | AccessLogValve.java | 906 |
| 10 | no | no, path-dependent | tomcat-4.1.24 | CertificatesValve.java | 385 |
| 18 | yes | yes | jakarta-tomcat-connectors | IntrospectionUtils | 847 |
| 19 | no | no, disproven | jakarta-tomcat-connectors | IntrospectionUtils | 847 |
| 20 | no | no, path-dependent | jakarta-tomcat-connectors | IntrospectionUtils | 912 |
| 21 | no | yes | jakarta-tomcat-connectors | IntrospectionUtils | 915 |

Table 1. Null pointer warnings reported by Reasoning

3.3 More accurate modeling of core API methods

In going from FindBugs 1.0 to FindBugs 1.2.0, we have improved our modeling of which methods in the core Java API classes require non-null values. Some of the improvements come from explicit marking (using annotations) of methods that require non-null parameters. Others depend upon an iterative application of our guaranteed dereference computation to determine method parameters that are always dereferenced (or passed to methods that, in turn, dereference them). Our guaranteed dereference computation improved in FindBugs 1.2.0, and we don’t separately break out which improvements came from explicit marking and which came from computation of guaranteed dereferences. In both cases, the new bugs that are reported due to these improvements involve passing a possibly-null value as an argument to a method that will unconditionally dereference it.

3.4 Reporting of errors on exception paths

FindBugs 1.0 contained an error in how it tracked potential null pointer dereferences on exception paths. Since the quality of the results wasn’t very good, they were only being reported at a low priority level warnings (which are usually ignored as they are in this paper). One we fixed this problem, we were able to report those warnings as medium priority warnings.

3.5 Field tracking

We added limited tracking of static and instance fields in order to detect more cases where a null value could be loaded from a field. We ignore aliasing, don’t track volatile fields, and assume that any method call could modify any field of any object passed to the method call. We also assume that any synchronization could cause all non-final fields to change.

3.6 Guaranteed Dereferences

While all of the things we’ve described are useful, none of the address the defects in Tomcat that we wanted to catch. In addition to the forwards data flow analysis used in FindBugs 1.0 [5], we added a backwards dataflow analysis of which values are guaranteed to be dereferenced on all non-exceptional paths to exit. We had already done a version of this analysis to determine which method parameters are unconditionally dereferenced, but we made some substantial improvements to it and also now incorporate it into our intraprocedural null pointer analysis. In Defect 08, the backwards propagation tells us that at else branch of the test on line 521, the current value of `href` is guaranteed to be dereferenced. When the backwards propagation of values guaranteed to be dereferenced encounters a statement or branch where that value is guaranteed to be null, we report an error.

We only consider non-exceptional paths due to the multitude of places that could throw an exception in a Java program. There are very few places where there is a value that is guaranteed to be deref-

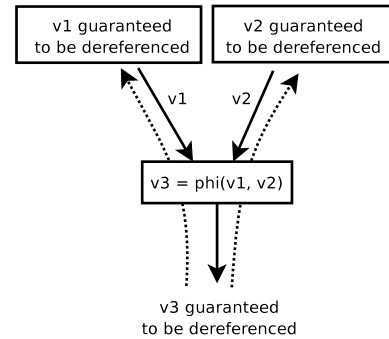


Figure 4. Propagating a guaranteed dereference backwards across a ϕ -node. (Dashed arrows show direction of propagation.)

erenced on all paths to method exit, including those paths caused by a runtime exception or an exception thrown by an invoked method.

4. Implementation notes

Our null-pointer analysis is based on a forward dataflow analysis that approximates static single assignment form (SSA) for values in local variables and on the operand stack. Specifically, the analysis tries to detect all cases where two values in local variables or the operand stack are the same, and assign such values the same “value number”. When two distinct values in the same local variable or operand stack location are merged as the result of a control join, the analysis creates a fresh value number distinct from any existing value number. Such join points are analogous to ϕ -nodes in SSA.

We compute the guaranteed dereferences as a backwards dataflow problem over our SSA approximation. At each program location where a value is dereferenced, we consider adding that value number to the set of values guaranteed to be dereferenced. As an optimization, we omit any dereference that occurs at a location where the dereferenced value is definitely non-null; such dereferences can be safely ignored. If a value is dereferenced at a location where the value is guaranteed to be null, we report the error directly without needing to perform propagation.

When the guaranteed dereference analysis encounters a ϕ -node going backwards, it must rewrite the dereferenced value sets on each control edge leading into the ϕ -node to translate from the value number assigned by the ϕ -node to the corresponding input value to the ϕ -node. This is illustrated in Figure 4.

The category of program locations we denote as “dereferences” are more inclusive than actual dereferences (i.e., instructions with a built-in null check); they are any location where a null value has a high likelihood of causing a null pointer exception. For example, we also count as dereferences passing a value to a parameter that

must be nonnull, returning a value from a method that must return a nonnull value, and storing a value into a field that has been annotated as nonnull.

We treat conditional branches on nullness specially. Specifically, in computing the guaranteed dereferences that propagate backwards from an `if (x == null)`, for the value in `x` we check to see if `x` is guaranteed to be dereferenced just on the true branch and ignore the else branch. Other values are marked as having a guaranteed dereference only if they have a guaranteed dereference on both branches.

4.1 Reporting defects

As part of our backwards guaranteed-dereference dataflow analysis, we compute, for each distinct known-null value, the set of locations such that one location in the set is guaranteed to dereference the value on a non-exception path to the method exit. Thus, when a guaranteed dereference and a known null value collide, we can report a warning that gives both the location where the value is known to be null and the set of locations, one of which is guaranteed to dereference it.

We prune the set of locations guaranteed to dereference the value: if the set contains both locations x and y , and y postdominates x , then remove x from the set of locations that are reported. This is rather effective, and in the vast majority of cases we can report a single location at which a dereference is guaranteed.

Although our analysis does not find as many bugs as purely path-based approaches, it has the considerable advantage that the bugs reported are easy to understand. A path-based analyzer might find many paths between a location where a value is known to be null and a dereference of that value. Reporting all paths might overwhelm the user, while picking one path arbitrarily might result in reporting an infeasible path. Our analysis reports warnings the user can verify with relative ease, since the null value mentioned by the warning is guaranteed to be dereferenced at one of the reported locations (unless an exception occurs).

4.2 Defect prioritization

FindBugs uses a number of heuristics in trying to prioritize defect warnings. As mentioned, we only report a null pointer warning if there is a statement or branch that, if executed/taken, guarantees a null pointer exception, assuming our program model is correct². We further refine this by raising the priority of any warning where there is a statement that, if executed, guarantees the null pointer exception. In our auditing of defect warnings we have found a fair number of cases where developers write conditional tests that they never expect to be fully covered. However, we have it found it much rarer for developers to intentionally write statements that they never expect to be executed.

5. Results

We were quite happy with the results of our new analysis. Table 1 notes the changes in our results on the warnings reported by Reasoning. In addition to reporting two more of the defects found by Reasoning, we report 24 additional null pointer defects missed by Reasoning. Figure 5 shows an interesting example of a defect found by the new analysis in Tomcat but not reported by Reasoning. FindBugs reports that if `container` is null on line 549, then `container` will be dereferenced at either line 552 or line 555. In fact, only the dereference at line 555 is feasible if `container` is null. When asked about this example, Dr. Morgenthaler said that Reasoning static analysis tool had reported a potential null pointer

²Our program model could be wrong due to field aliasing, not understanding which methods modify which functions, or not understanding when exceptions will be thrown and execution will not continue normally

```
// org.apache.cataline.core.StandardPipeline:
546 protected void log(String message) {
547
548     Logger logger = null;
549     if (container != null)
550         logger = container.getLogger();
551     if (logger != null)
552         logger.log(... + container.getName()
553                 + "]: " + message);
554     else
555         System.out.println(... + container.getName()
556                           + "]: " + message);
557
558 }
```

Figure 5. Null pointer bug missed by Reasoning

```
// sun.awt.X11.XMSelection
// lines 242-246
public synchronized void removeSelectionListener(
    XMSelectionListener listener) {
    if (listeners == null) {
        listeners.remove(listener);
    }
}
```

Figure 6. The Null Pointer Bug That Didn't Bark

exception on the path from line 549 to line 552, but had not reported a potential null pointer exception on a path from line 549 to line 555. Since the NPE at line 552 was infeasible, they filtered it out.

Table 2 shows the overall changes in our results on Sun's JDK 1.6.0-b105 and Eclipse 3.2.1. This table details the changes in the high/medium priority null pointer correctness warnings about paths where a value is known to be null and guaranteed to later be used in a way that required it to be nonnull, and elsewhere is to roughly double the number of null pointer warnings we report, while improving the quality of the defects we report. Because Java is a memory-safe language, not all of these defects are important, and trying to identify the important null pointer defects is ongoing work. Figure 6 shows an interesting situation where the most severe impact of the defect is felt when the null pointer exception does *not* occur. If a listener is removed before any have been added, a null pointer exception will occur. But that situation seems unlikely and if it does occur, hopefully the null pointer exception will get logged and reported, leading to the software being fixed. However, in the typical case, where a listener is removed after having been added, the impact of the defect is to make removal of listeners fail silently.

5.1 Other Analysis Tools

This section tries to provide some understanding of where the current FindBugs analysis fits in the world of static analysis for null pointers. It is important to understand that there is no one way to design a null pointer analysis. FindBugs has been tuned to have a very low false positive rate, so that it can be reasonably applied to multi-million line programs. Other tools have made different trade offs for false negatives and false positives.

One comparison is to examine the total number of warnings reported for a large code base. Table 3 reports the number of null pointer warnings generated by various static analysis tools on Sun's jdk1.6.0-b105. The FindBugs numbers in this table differ from the number in 2 because this table includes all medium/high priority

| Eclipse | JDK | Explanation |
|---------|-----|--|
| 101 | 70 | Number of NP correctness warnings reported by FindBugs 1.0 |
| -16 | -20 | Warnings removed due to bug fixes and better modeling of asserts, panic methods, JDK libraries |
| -23 | -13 | Warnings about equals method not handling null reclassified as bad practice |
| +8 | +38 | New warnings due to better modeling of JDK libraries |
| +3 | +16 | New warnings due to fix in tracking of null dereferences on exception paths |
| +37 | +12 | New warnings that required field tracking |
| +50 | +17 | New warnings that required guaranteed dereferences |
| +13 | +1 | New warnings that required both field tracking and guaranteed dereferences |
| 173 | 121 | Number of NP correctness warnings reported by FindBugs 1.2.0-dev |

Table 2. Changes in null pointer warnings between FindBugs 1.0 and 1.2.0-dev

```

// 5 intraprocedural microbenchmarks
// for null pointer defect detection
// Each of these contains a defect
//
// Some microbenchmarks also have a variant that
// is a false positive.
// Substitute the operator or expression
// in the /* comments */ to get a version
// with no defect: reporting a warning
// in those cases is a false positive

int intra1(int level) {
    Object x = null;
    if (level > 0)
        x = new Object();
    if (level < /* > */ 4)
        return x.hashCode();
    return 0;
}

int intra2(boolean b) {
    Object x = null;
    if (b) x = new Object();
    if (!b /* b */) return x.hashCode();
    return 0;
}

int intra3(Object x) {
    Object y = null;
    if (x != null)
        y = new Object();
    if (y != null)
        return x.hashCode() + y.hashCode();
    else
        return x.hashCode() /* 0 */ ;
}

int intra4(boolean b) {
    Object x = null;
    Object y = null;
    if (b) x = "x";
    if (x != null) y = "y";
    if (y != null)
        return x.hashCode() + y.hashCode();
    else
        return x.hashCode() /* 0 */ ;
}

int intra5(Object x) {
    if (x == null) {
        return x.hashCode();
    }
    return 0;
}

int intra6(Object x) {
    if (x == null) {
        Object y = x;
        return y.hashCode();
    }
    return 0;
}

int inter1(boolean b) {
    Object x = null;
    if (b /* !b */) x = new Object();
    return helper1(x, b);
}

int inter2() {
    return helper2(null);
}

int inter3(boolean b) {
    Object x = null;
    if (b) x = "x";
    return helper2(x);
}

// Bug when x is null and b is false
private int helper1(Object x, boolean b) {
    if (b) return 0;
    return x.hashCode();
}

private int helper2(Object x) {
    return x.hashCode();
}

```

Figure 7. Null pointer microbenchmarks, v2

| | intraprocedural cases | | | | | | interprocedural | | |
|-----------|-----------------------|-------|-------|-------|-----|-----|-----------------|-----|-----|
| | 1 | 2 | 3 | 4 | 5 | 6 | 1 | 2 | 3 |
| CodeSonar | v,f | v,f | v,f | v,f | v,f | v,f | v,f | v,f | v,f |
| Eclipse | vv | vv | vv | vv | v | v | | | |
| FindBugs | | | v,f | v,f | v,f | v,f | | v | v |
| Fortify | vv,ff | vv,ff | vv,ff | vv,ff | v,f | v,f | | | |
| IntelliJ | vv,ff | v,f | v,f | vv,ff | v,f | v,f | | | |
| Jtest | v,f | v,f | | v,f | | v,f | v,f | v,f | v,f |
| Klocwork* | | | | | | | | | |

* - due to licensing terms, Klocwork results can not be disclosed

Table 4. Microbenchmark, v2, results from different tools

| # | Tools |
|-------|--------------------|
| 182 | FindBugs 1.2.0-dev |
| 1,018 | Eclipse 3.3M2 |
| 272 | IntelliJ 6.0.4 |

Table 3. Null pointer warnings on JDK

correctness null pointer warnings, including warnings about unwritten fields being read and dereferenced and about values being dereferenced and later checked to see if they are null.

Now, no particular number of warnings is right or wrong. Both IntelliJ and Eclipse undoubtedly correctly report some null pointer defects that are missed by FindBugs. However, the number of warnings reported by Eclipse make the idea of systematically reviewing null pointer warnings reported by Eclipse tools on a large software project a daunting task.

To further clarify the differences, we devised a series of null pointer microbenchmarks, give in Figure 7 and in a Subversion repository [6]. It is important to note that this is a microbenchmark, designed to elucidate certain aspects of how null pointer analysis works rather than to gauge the effectiveness or value of an analysis. We analyzed both this class, and a variant in which x is an instance field rather than a local variable. The results from several different analysis tools are reported in Table 4. An entry of v shows that the tool reported a warning for the version of the benchmark where x is a local variable, and where the defect is actually possible. An entry of vv shows that a defect is reported in both the true positive and false positive case. Entries of f and ff note the results in x is a field.

Several notes about the results in this table. Klocwork makes a trial version of their analysis tool freely available on their website, but their license strictly prohibits us from disclosing any information about the capabilities of their tool. We have left a blank line for Klocwork and encourage readers to download the Klocwork tool and fill in their own results. Fortify Software notes that a revision planned for release this summer incorporates some null pointer interprocedural analysis. Coverity declined to share any information on their tool. CodeSonar from GrammaTech, a tool for C/C++, does an exceptionally precise job on a C++ version of the benchmark, as appropriate for a defect detection tool designed for a language that lacks memory safety. Jtest 8.0 from Parasoft don't report defects in two cases in the microbenchmark because they decided not to report potential defects when the only reason to believe that the value might be null is that it was previously compared to null. They report that their users felt that doing so generated too many false positives, but they have decided to modify their analysis to allow users to select whether to report potential defects in these cases.

6. Related work

There have been quite a number of papers on static analysis for defect detection, and many of these [4, 3, 2] touch substantially but

not exclusively on finding null pointer defects. We previously published work [5] on the null pointer bug detection used in FindBugs 1.0.

7. Conclusions

Since our work is done in the context of a memory safe language, the impact of null pointer errors is typically much less significant than in languages such as C/C++. Thus, we believe we have reached a happy medium with our null pointer analysis. We are pleased with the quality of the defects we are reporting, and given the number of null pointer warnings we find in existing code, we don't want to expand the analysis to report warnings only feasible on some paths.

The most interesting question to us is not how to lower our false positive or false negative rate, but how to identify high-impact null pointer bugs. Once we can do that, then trying to apply those techniques over a larger field of null pointer bugs, including ones FindBugs does not now report, would be of interest.

8. Acknowledgments

Thanks to David Morgenthaler for challenging us, and to all those who have contributed to the FindBugs project. Fortify Software is the sponsor of the FindBugs project, and additional current support is provided by Google and Sun Microsystems.

References

- [1] Apache tomcat. <http://tomcat.apache.org>, 2006.
- [2] I. Dillig, T. Dillig, and A. Aiken. Static error detection using semantic inconsistency inference. In *Proceedings of the Conference on Programming Language Design and Implementation*, June 2007.
- [3] 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 *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, pages 57–72, New York, NY, USA, 2001. ACM Press.
- [4] D. Evans. Static detection of dynamic memory errors. In *PLDI '96: Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation*, pages 44–53, New York, NY, USA, 1996. ACM Press.
- [5] D. Hovemeyer, J. Spacco, and W. Pugh. Evaluating and tuning a static analysis to find null pointer bugs. In *PASTE '05: The 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 13–19, New York, NY, USA, 2005. ACM Press.
- [6] W. Pugh. Null pointer detection microbenchmarks. <http://findbugs.googlecode.com/svn/trunk/NullPointerBenchmark/>, 2006.
- [7] I. Reasoning. Reasoning inspection service defect data report for tomcat, version 4.1.24, January 2003. http://www.reasoning.com/pdf/Tomcat_Defect_Report.pdf.