Null Dereference Analysis in Practice

Nathaniel Ayewah
Dept. of Computer Science
University of Maryland
College Park, MD
ayewah@cs.umd.edu

William Pugh
Dept. of Computer Science
University of Maryland
College Park, MD
pugh@cs.umd.edu

Abstract

Many analysis techniques have been proposed to determine when a potentially null value may be dereferenced. But we have observed in practice that not every potential null dereference is a "bug" that developers want to fix. In this paper we discuss some of the challenges of using a null dereference analysis in practice, and reasons why developers may not feel it necessary to change code to prevent ever possible null dereference. We revisit previous work on XYLEM, an interprocedural null dereference analysis for Java, and discuss the challenge of comparing the results of different static analysis tools. We also report experimental results for XYLEM, Coverity Prevent, Fortify SCA, Eclipse and FindBugs, and observe that the different tools tradeoff the need to flag all potential null dereferences with the need to minimize the number of cases that are implausible in practice. We conclude by discussing whether it would be useful to extend the Java type system to distinguish between nullable and nonnull types, and prohibit unchecked dereferences of nullable types.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages—Program Analysis; D.2.4 [*Software Engineering*]: Software/Program verificationReliability

General Terms Reliability, Security

Keywords Static analysis, null pointer dereference

1. Introduction

Static analysis to detect potential null pointer dereferences is a well studied field. However, in memory-safe languages such as Java, a potential null pointer dereference isn't necessarily a sign of defective code. In some cases, the potential null pointer dereference could occur only in situations believed to be impossible by the developer, or in situations in which expected preconditions or invariants are violated. In such cases, the immediate "fix" to avoid dereferencing a null pointer would be to throw some other kind of runtime exception (e.g., an illegal argument exception). Recall that the Java coding standard recommends that if null is supplied for a parameter that is required to be nonnull, a `NullPointerException` *should* be thrown. So a dereference of potentially null parameter and an explicit throw if the parameter is null both result in the same behavior (although an explicitly thrown exception might include a message that names the parameter that is null). Even in cases where it might be appropriate to throw some other runtime exception, such a fix would likely have little or no impact on user observed execution behavior, although it could expedite debugging of executions in which an exception did occur.

In this paper, we review some examples of potentially null dereferences that are flagged as defective by some static analysis tools but not others. We compare the outputs of XYLEM [9], Coverity Prevent, Fortify SCA, Eclipse and FindBugs. Of the tools studied, XYLEM is the best tool for finding as many potential null dereferences as possible with minimal false positives. But we observe that many potential null dereferences that pass initial software testing do not cause subsequent null dereference failures, and reviewing all potential null dereferences is often not as important as many other undone software quality tasks.

We also reviewed some of the experimental data previously reported on XYLEM [9], and found some mistakes in their summary of the data. One of those mistakes removed some of the experimental support for one of the conjectures of the paper, that the null pointer issues they find are more important than the null pointer issues found by FindBugs.

We also discuss whether it would be useful to extend the Java type system to distinguish between nullable and nonnull types, and prohibit unchecked dereferences of nullable types. As part of this discussion, we present statistics on invocations of `Map.get`, a method that is widely used and is defined to return null if either the key has no mapping, or is mapped to a null value. We also discuss cases where one might prefer a null pointer exception (and subsequent program failure) over more subtle bugs that might otherwise occur.

> *"I've checked it very thoroughly,", said the computer, "and that quite definitely is the answer. I think the problem, to be quite honest with you, is that you've never actually known what the question is."*

1.1 What is a bug?

Several researchers have noted [9, 10] that their static analysis techniques find more "null pointer bugs" than FindBugs, a static analysis tool developed at the University of Maryland. FindBugs does relatively simple analysis, so the fact that other tools could find more potential null dereferences is unsurprising. But on closer examination, some researchers tend to be conflating the concepts of "null pointer bug" and "potential null pointer dereference" in a way that obscures certain important points.

Some researchers seem to assume that any feasible null dereference is, by definition, a bug. But such a definition doesn't take into account the circumstances under which a null dereference can

PASTE'10, June 5–6, 2010, Toronto, Ontario, Canada.

```
/**
  * Deletes the directory at dirName and all its files.
  * Fails if dirName has any subdirectories.
  */
 public static void deleteDir(File dir) {
   File[] files = dir.listFiles();
   for (int i = 0; i < files.length; i++) {
     files[i].delete();
   }
   dir.delete();
 }
```

Figure 1. A potential null pointer dereference

occur (e.g., perhaps it can occur only when method preconditions are violated) and whether a null dereference would produce a functional difference compared with the behavior of the code modified to avoid the potential null dereference.

We propose that a "bug" or "defect" in code is an issue that an informed developer would want to resolve by changing the code. Some "defects" are more important than others: a potential SQL injection might be considered far more important than an issue that results in incorrect logging. This classification is subjective, depending on both the project and the developer. A developer might want to resolve some issues just to make the code easier to understand and maintain, even if those issues can't result in application misbehavior. We want tools and techniques to help developers and projects find defects in a cost effective way, as too much effort on one path to finding defects may come at the expense of other techniques that are more effective at finding important defects.

For example, consider the code in Figure 1. Michael Ernst cited [10] this code as containing a null pointer bug missed by FindBugs. The issue is that the `listFiles()` method is defined to return null if the `File` object it is invoked on does not refer to a readable directory. There is an unstated precondition to the `deleteDir` method that `dir` should refer to a readable directory, and if this precondition is violated, a null pointer exception will be thrown.

Obviously, any execution that results in something other than a readable directory being passed to the `deleteDir` method is erroneous, but in such an execution it would seem that the most significant defect would be outside of the `deleteDir` method. Within the `deleteDir` method, the only reasonable "fix" is to throw an `IllegalArgumentException` if `dir` is not a readable directory. This would be more helpful in debugging erroneous executions, since the exception would more clearly indicate the problem and could include the value of the `dir` parameter as part of the exception message. However, it seems unlikely that there are situations in which this "fix" would make an application behave correctly.

The `deleteDir` method is written as a public utility method in a library. Having such a method throw a NullPointerException because an invalid argument was provided would be extremely confusing, so changing it would assist the debugging of code that invoked that method. On the other hand, if `deleteDir` were a private method used only within a specific application, and any potential null pointer exception would only be visible to code within the application, changing the code might not be important unless there was evidence the exception was occurring in practice. If it were to happen, the developers could be reasonably expected to track down the exception and understand why it was occurring.

The point of this discussion isn't to say that it is *wrong* to state that `deleteDir` contains a null pointer bug, but to say that the term "null dereference bug" is a vaguely defined term, subject to different interpretations. Clearly, the `deleteDir` method contains a potential null pointer dereference, and from a program analysis point of view, it would be more precise to talk about algorithms for finding potential null pointer dereferences. The question of whether code should be modified to prevent a potential null pointer dereference is a different question and subjective judgment call. Every software project and developer has a limited budget. Every change made to the code introduces a possibility, however slight, of introducing a new error, and takes time away from other software quality efforts.

In fact, there is an issue with the `deleteDir` method that is potentially more serious than the null pointer issue. The `delete()` method returns false if the deletion was not successful, and the `deleteDir` method ignores this return value. As a result, the `deleteDir` method might delete some but not all files in the directory and not provide any warning or signal that the deletion was incomplete.

	Null dereference warnings for Ant versions			
	1.5.0			
	true	false	1.6.0	1.6.5
[9]	82	4	130	82
corrected	75	11	110	62

Table 1. Corrections of XYLEM results from [9]

2. Static analysis

We revisit the result of the XYLEM static analysis tool [9], discuss the challenges of consistency of analysis results across versions, evaluate reasons why potential null dereferences found by static analysis may be impossible, implausible or of little concern in practice, and compare the results from different static analysis tools.

2.1 Revisiting XYLEM data

Nanda and Sinha presented a paper [9] at ICSE 2009 describing XYLEM, a static analysis tool to detect potential null pointer dereferences. We've described FindBugs as looking for "low hanging fruit", and have always been interested in understanding the difference between what could be found by tools such as FindBugs and techniques that look for fruit higher up the tree. The approach described by Nanda and Sinha is quite sophisticated, and their paper reported that they found many more issues than FindBugs with a very low false positive rate, and that the issues found by their tool tended to be removed at a higher rate [9, Section 4.4] than the issues found by FindBugs and thus might be more important.

Mangala Nanda graciously shared the raw data from her experiments with us. In reviewing that data, we found a number of data analysis errors in the results presented in [9]. We have worked with the authors to understand those errors, and present results from a newer version of XYLEM that we believe correct the errors.

Table 1 gives the data originally presented by [9], and necessary revisions based on our review of the raw data from that experiment. The XYLEM tool presented in [9] generates as output a text file with one line per potential null dereference, and the authors made several mistakes in their manual review of the output files. But these mistakes do not significantly change the results.

Nanda and Sinha [9, Section 4.4] cited the fact that 26% of the null pointer dereferences reported by XYLEM in Ant were deleted in later versions as evidence those issues were important defects since the software was changed to "fix" them. They reported [9, corrected] 110 null pointer dereferences in Ant 1.6.0, of which 57 were not reported in Ant 1.6.5. In Ant the `Project.createTask(String)` returns null when the argument to `createTask` isn't associated with a class that implements the

`Task` interface. 30 of the 110 warnings in Ant 1.6.0 are places where a constant naming a well known ant task (e.g., `"ant"` or `"move"`) is supplied to `createTask`, and the result is dereferenced without being checked for null. We will return (in Section 2.4) to the question of whether or not this is a coding mistake. But more relevant at the moment is the fact that none of these issues are reported by XYLEM in Ant 1.6.5; they account for 30 of the 57 issues that were "removed" between Ant 1.6.0 and Ant 1.6.5. However, the semantics of `Project.createTask(String)` didn't change between 1.6.0 and 1.6.5; it is still clearly specified to return null if the parameter doesn't match a known task definition. However, between the 1.6.0 and 1.6.5, the implementation of `Project.createTask(String)` introduced an additional level of indirection, which pushed the potential null dereference beyond the analysis horizon of XYLEM.

This removes some of the experimental support for the conjecture ([9, Section 4.4]) that the issues reported by XYLEM in Ant are important. Such data is tricky to interpret. The fact that an issue is no longer reported might not reflect whether the code was changed in order to remove the potential null dereference. For example, most of the potential null dereferences of the return value of `createTask` were eliminated in Ant 1.7.0, when the code was changed to directly invoke constructors to create standard tasks. Thus, the potential null dereferences disappeared. Were they "fixed", or was the elimination of the potential null dereferences a side effect of a refactoring with another purpose (perhaps performance or code clarity)? Even if a potential dereference is deliberately removed, the removal doesn't indicate whether the potential null dereference was important or occurring in practice. All 10 of the null dereference issues reported by FindBugs in Ant 1.6.5 were removed in Ant 1.7.0 or Ant 1.7.1; these were removed because Dave Brosius, a contributor to the FindBugs project, submitted patches to Ant that corrected mistakes pointed out by FindBugs. But there is no indication that these potential dereferences were causing field failures.

While it would be possible to further dissect the raw data published previously [9], the authors have provided us with output from a newer version of XYLEM. In addition to correcting a number of issues with the analysis, the new output format is significantly more detailed, providing additional information to help understand the issues. We wrote a tool to convert the new output format of XYLEM into FindBugs XML format. This allows us to use the FindBugs tool chain to review issues and automatically trace the occurrences of issues across different versions of software (e.g., across different versions of Ant). Table 2 gives the number of issues found by XYLEM in various versions of Ant. For example, the "as reported" columns show that XYLEM reported 69 issues in Ant 1.6.0, and of the 76 issues reported in Ant 1.5.4, 39 were no longer reported in Ant 1.6.0 (e.g., "disappeared").

Although XYLEM is accurate, it isn't sound and complete. A change that doesn't actually affect whether a null dereference is feasible can influence what interprocedural paths are explored, and whether or not a potential null dereference is reported. In followups to conversations on the topic, Nanda and Sinha have improved XYLEM to try to make it more consistent, but it still suffers from some inconsistencies. For example, of the 39 issues no longer reported by XYLEM in Ant 1.6.0, only 14 are gone forever; 25 of them are reported again in some later version. Manually examining those 25, there is no obvious code change that would appear to be responsible for them disappearing and then reappearing. The right two columns of Table 2 show the results with "resurrections", where if an issue was reported in one version of the code and a later version of the code, we assume it was present in all intervening versions, even if XYLEM didn't report it in those versions. In some cases, there may have actually been a change that makes the null dereference impossible, and then another later change that reintroduced it.

Ant	as reported		with resurrections	
version	disappeared	present	disappeared	present
1.3		33		33
1.4.0	6	51	4	53
1.4.1	2	49	2	51
1.5.0	16	72	15	75
1.5.1	2	75	2	78
1.5.2	7	70	3	77
1.5.3-1	4	72	1	78
1.5.4	1	76	0	79
1.6.0	39	69	14	95
1.6.1	3	69	1	97
1.6.2	4	72	3	101
1.6.3	8	104	6	109
1.6.4	8	101	6	105
1.6.5	4	104	3	105
1.7.0	54	89	53	90
1.7.1	10	84	10	84

Table 2. XYLEM issues reported in versions of Ant

Note that the version in which issues "disappear" only to be later resurrected can change from version to version of XYLEM, the static analysis tool. For the results published in the ICSE 2009 paper [9], the issues involving the null return from `Project.createTask` were reported in Ant 1.5.4 and Ant 1.6.0, but not Ant 1.6.5. Using the most recent version of XYLEM, they were reported in Ant 1.5.4 and Ant 1.6.5, but not in Ant 1.6.0.

2.2 The challenge of consistency

We've previously [11] discussed the issue of consistency of static analysis results, and our subsequent experience in production deployments and conversations with static analysis tool vendors has only reinforced our belief in the importance of that issue.

XYLEM is performs effort-limited interprocedural analysis. As such, it isn't sound and complete, but this is a commonly accepted tradeoff in static analysis for bug detection. However, our examination of the XYLEM results suggest that *any* form of effort-limited interprocedural analysis may suffer from inconsistency problems that may limit the use of such techniques in production. We need to worry about consistency across

- different versions of the software artifact,
- different versions of the static analyzer, and
- different runs of the same version of the artifact and analyzer

By consistency, we mean that unless there is a good reason for the results to change, we report the same issues from run to run, and that we can identify a correspondence between the issues reported in one run and those reported in another. Doing any kind of effort-limited analysis (only searching interprocedural paths up to a certain depth or until a timer expires) can introduce inconsistencies as program or analysis changes change what is detected within the allowed effort. In theory, trivial changes in memory layout or timing can change the order in which hash table entries are enumerated, causing inconsistencies from run to run of the same analysis version on the same artifact version.

Consistency also requires that we be able to track the correspondence of issues between versions. Most production static analysis deployments don't insist on a "no issues" standard. Instead, they generally strive for either a "no unreviewed issues" or a "no unreviewed new issues" policy. Many tools will sometimes (or fre-

```
// JonasDeploymentTool.java
File f = new File(outputdir + File.separator + key);
// warning that f.getParentFile() might return null
f.getParentFile().mkdirs();

// Expand.java
zf = new ZipFile(srcF, encoding);
Enumeration e = zf.getEntries();
while (e.hasMoreElements()) {
  ZipEntry ze = (ZipEntry) e.nextElement();
  // warning that zf.getInputStream(ze) might return null
  extractFile(... zf.getInputStream(ze) ... )
}

// XMLCatalog.java
InputStream is = loader.getResourceAsStream(location);
if (is != null) {
  source = new InputSource(is);
  URL entryURL = loader.getResource(location);
  // warning that loader.getResource might return null
  String sysid = entryURL.toExternalForm();
  ...

// JJTree.java
while (root.getParent() != null)
  // warning that getParentFile might return null
  root = root.getParentFile();
```

Figure 2. Impossible dereferences reported by Coverity Prevent

quently) generate warnings for which no code change is appropriate. It may be important or required to review all new issues. But once an issue has been reviewed and marked as "harmless", "bad analysis" or "will not fix", it is important for the system to be able to associate that review with issues generated from rerunning the analysis. We discussed [11] two different techniques used for this in FindBugs. Fortify SCA and Coverity Prevent all use variants of one of the methods described (computing a hash value for each issue that is hoped will be invariant).

2.3 Impossible dereferences

Figure 2 shows some impossible dereferences reported by Coverity Prevent in Ant 1.6.5. In each case, the tool warns about a method that *might* return a null value which would be subsequently dereferenced. In each case, due logic associated with each API, the return value at that call site is guaranteed to be nonnull. XYLEM doesn't analyze methods defined by the JDK for possible null return values, so the experimental results we have don't say whether XYLEM would be confused by similar situations in user code.

For the JonasDeploymentTool code, `getParentFile` returns null if the File doesn't contain any instances of the `File.separator` character, but the line above it guarantees that it does. In Expand.java , `zf.getInputStream(ze)` returns null if there is no matching entry in the ZipFile. But since we got the ZipEntry by enumerating the entries contained in the file, we are sure to get a nonnull value. In XMLCatalog, if `loader.getResourceAsStream` returns a nonnull value for a particular argument, `loader.getResource` is guaranteed to do so as well. In JJTree.java, if `root.getParent` returns a nonnull value, `root.getParentFile` is guaranteed to do so as well.

2.4 Implicit nonnull assertions

We also found a fair number of places where a return value was being dereferenced without a null check, the method might return null in some circumstance, and it wasn't easy to prove that the method invocation couldn't return null. Yet, it seemed like it would be unlikely to be null in any reasonable circumstance. For example, the Ant code expects that when the `createTask` method is invoked with a constant string specifying one of the set of standard ant tasks, the `createTask` method will always return a nonnull value.

Tool	Total	Plausible	Implausible	Impossible
Coverity	46	17	15	14
Eclipse	31	11	1	20
FindBugs	11	11	0	0
Fortify	44	14	1	29
XYLEM	57	35	15	7

Table 3. Null dereferences reported in Ant 1.6.5

#	review	why
15	plausible	clear coding mistakes
7	plausible	plausible error condition not handled
13	plausible	seems plausible, but not clear what situation would cause it to arise
15	implausible	calls to Project.createTask with well known String constant
4	impossible	coupled variables
1	impossible	call context guarantees nonnull return value
2	impossible	value previously dereferenced and thus can't be null

Table 4. Review of XYLEM warnings in Ant 1.6.5

If fact, if Ant is started with a corrupted properties file, it may not find some of these task definitions. But it seems reasonable to not require explicit null checks in this instance.

In our review of static analysis results in 2.5, we manually classified each reported issue as to whether it seemed impossible under any execution, implausible or impossible in any non-corrupted execution. The ones not so classified might or might not be ones that could occur in practice; local inspection didn't provide any reason to believe them to be infeasible.

2.5 Static analysis results

Table 3 lists warnings reported against Ant 1.6.5 by Coverity Prevent 4.5.0, Eclipse 3.5.0, FindBugs 1.3.8, Fortify 360 SCA 2.1.0 and XYLEM (November, 2009). The XYLEM analyzer reported (roughly) twice as many plausible null pointer dereferences than any other analysis engine, but our evaluation was that only 60% of the issues reported by XYLEM were plausible. Table 4 gives a more detailed breakdown/description of our (subjective) evaluations of the warnings reported by XYLEM. All of the issues reported by FindBugs seems to be clear coding mistakes, but it also found the fewest number of plausible issues. None of the plausible issues in Table 3 are known to have caused any field failures (one reported by XYLEM in Ant 1.5.0 is known to have caused a field failure: Bug 10360).

For consistency, we only report issues where a value is known to be or checked against being equal to null, and later dereferenced. Some tools also report other kinds of null pointer issues, such as checking a value to see if it is null *after* it has been dereferenced. For each tool, we report how many issues we believe are implausible (Section 2.4) and impossible (Section 2.3).

Fortify SCA reports the combined results of FindBugs and their own analysis engine. On the recommendation of Andy Chou of Coverity, we enabled an undocumented and unsupported *effects analysis* feature in Coverity Prevent. Without this feature, significantly more results, all impossible, were reported for Coverity Prevent.

Details of our experimental results are available at `http://findbugs.sourceforge.net/publications.html`.

3. Annotations for nullness

There have been many proposals to introduce annotations that specify whether values or types are allowed to be null [2–4, 10]. Most proposals simply allow for one of two annotations: that either a value or type is never allowed to be null, or that it is allowed to be null and that a warning or error should be generated if such a value is dereferenced without a null check.

In some proposals [3], the annotations can be provided to generic type parameters. Thus, for a `List` of nonnull strings, the `get(int)` method returns a nonnull value, while for a list of nullable strings, the same method may return a null value.

The problem with these proposals is that there are many existing methods that return null under some circumstances, but particular invocation sites may be reasonably expected to always return a nonnull value except under erroneous conditions that should lead to a runtime exception. Section 2.3 discusses some examples of methods in the Java libraries that return null under some circumstances, but are sometimes invoked in circumstances under which they can't possibly return null. Papi et al. [10] describe this phenomenon as a type system weaknesses, and as "application invariants". More sophisticated systems [5, 7, 8] for specifying method contracts may allow for compile time checking of such invariants, but uptake of such approaches in production environments has been slow.

A particularly common such situation is the `Map.get` method, discussed in Section 3.1. Similar situations also frequently occur in application code, such as the `createTask` method in Ant.

Arguably, many APIs would be better if they were more reserved in their use of null, and if they allowed a more consistent labeling of return values and parameters as either not allowing null or needing to be checked for null. None the less, an annotation system must serve the needs of existing APIs, as well as assist in the documentation and type checking of new APIs.

To handle nullness, static analysis tools have some advantages over type system extensions. Static analysis can look into methods, and attempt to discern provable conditions under which null will not be returned. They can also perform statistical modeling of which values are null checked: in one context, the return value of a particular method might be frequently dereferenced without a null check. In another context or project, the value might always (or almost always) be null checked before being dereferenced.

This isn't to say that annotations systems for nullness aren't useful. Just that they may benefit from allowing more flexibility than "never null" or "should always be checked for null". FindBugs allows for one of three nullness types: "never null" , "null in some circumstance", and "should always be checked for null". In most cases, dereferencing a value that is "null in some circumstance" will not generate a FindBugs warning. In theory, many refinements of "null in some circumstance" are possible, but the advantages of using more are not clear.

An alternative possibility is to use strict nullness type annotations, but provide for a very concise syntax to cast to a nonnull type. It has been proposed [6] that Java be extended with a null safe dereference operator (`?.`). In this proposal, expression x`?.`y is defined to be null if x is null, and x`.`y if it is not. Perhaps Java should provide a dereference operator (perhaps `!.`) that includes an implicit precondition or assertion that the left hand side value is nonnull, and another operator that acts on a concise cast to a nonnull value. Where a method is called and the developer believes the return value should never be null in this calling context, they can concisely express that belief in the code. The JSR308 checker framework [12, Section 3.4.2] suggests several ways to suppress nullness warnings in cases where the developer believes the null dereference cannot occur:

Software	invocations	null checked	unconditional dereferences
JDK 1.7.0	2516	1040	325
JBoss 5.1.0	3095	1680	105
Glassfish v3	1225	1672	90

Table 5. Invocations of Map.get

- An annotation to suppress warnings about nullness (e.g., `@SuppressWarnings("nullness")`)
- An explicit check for nullness, throwing an exception if it is null.
- An assertion that the value is nonnull.
- Invoking `NullnessUtils.castNonNull(...)` to cast the value to a nonnull value.

In addition, with a static analysis framework that supports persistent reviews of issues, the issue could be marked as "not a bug".

However, we think each of these is inferior to some new concise syntax. It is desirable to have all of the nullness assertions expressed in the source code and to be able to review them. However, if the syntax for such assertions is verbose, it can get in the way of developers who are just trying to understand the logic of the code.

3.1 Uses of Map.get()

Many methods that return null in some context are often invoked in situations where they are never expected to return null. Due to the ubiquitous use of the `Map` interface in Java, `Map.get()` seems to account for a plurality of such method invocations (although it may not be a majority of such invocations). Thus, it seemed reasonable to study the how invocations of `Map.get` deal with the possibility of a null return value.

We want to examine results we would get from a static analysis tool that generated a warning every time the result of `Map.get` was dereferenced without first being nullchecked. Looking at the warnings generated, we can try to evaluate how many of them might be logically impossible due to other considerations, and whether it would be possible or desirable to build those considerations into a static analyzer.

We used FindBugs to examine the invocations of `Map.get` in several software artifacts, and report the results in Table 5. We report the total invocations of `Map.get`, and the number of invocations for which there was an explicit null check within the same method. We also modified FindBugs to report the number of unconditional dereferences of the return value of `Map.get`

We manually reviewed the 325 places in the JDK where the return value of `Map.get` was unconditionally dereferenced. Looking over these, we found that many of these warnings seemed to be false positives, except in truly pathological cases. But most of the false positive warnings were also examples of questionable or inefficient coding idioms. For example, the calls to `get` in Figure 3 can reasonably expect that `id` is contained in the map `collectedReferences`, due to the call to `containsKey` (in theory, that expectation could fail if the map was modified between the two calls). But the code in Figure 3 performs 4 lookups in the `collectedReferences` data structure, which is inefficient and could confuse a developer examining the code. The code would be better if it were transformed as shown in Figure 4, which makes one atomic call to `get` to determine if `id` is contained in the map and to obtain the value it is associated with in the map.

Reviewing the 325 warnings, there were three common idioms that would seem to guarantee that at a call to Map.get, the key supplied was contained in the map:

```
// com.sun.codemodel.internal.JFormatter, lines 295-305
if ( collectedReferences.containsKey(id) ) {
  if ( !collectedReferences.get(id).getClasses().isEmpty() ) {
    for ( JClass type : collectedReferences.get(id).getClasses() ) {
      if ( type.outer() != null ) {
        collectedReferences.get(id).setId(false);
        return this;
      }
    }
  }
  collectedReferences.get(id).setId(true);
}
```

Figure 3. "unchecked" dereferences of Map.get()

```
ReferenceList refs = collectedReferences.get(id);
if ( refs != null ) {
  for( JClass type : refs.getClasses() )
    if ( type.outer() != null ) {
      refs.setId(false);
      return this;
    }
  refs.setId(true);
}
```

Figure 4. Improved version of code in Figure 3

num	preceding check
91	iterating through keySet
55	call to containsKey
46	previous check if get() != null
133	no obvious common idiom; null deference might be feasible

Table 6. Idioms used to ensure key present for Map.get call

- The code contained a loop over the keys in the map, and for each key was calling Map.get.
- The code contained an earlier call to Map.containsKey
- The code contained an earlier call to Map.get with the same key.

Of course, such idioms are useless if another thread might remove keys from the map. But even assuming that no other threads modify the map, doing such an analysis would be challenging. Instead, we wanted to implement a quick and dirty analysis to decide the landscape of coding idioms that would have to be recognized in order to correctly identify such cases. Our results are in Table 6.

Now, it is possible to augment a static analysis checker to understand some of these idioms, and deduce places where the return value of `get` on a Map with nonnull values can be expected to always be nonnull. In fact, the nullness checker developed as part of the JSR 308 effort does recognize some of these idioms. But in our examination, more than one third of the unchecked dereferences didn't correspond to any recognized idiom. Due to the number of such warnings involving `Map.get`, providing special handling might be worthwhile. But there are many such methods (such as those described in Section 2.3 and Figure 2), and trying to construct special idiom recognition for all of them seems impractical.

4. When NPE is Better

In reviewing some of the potential null dereferences reported by various tools, we sometimes found that a null dereference was impossible, but that the issue reflected a significant defect in the code. In other cases, the null deference was feasible, but the more common and severe manifestation of the defect occurred when no null dereference occurred. One such situation is the code fragment

```
// ...apache.xalan....xsltc.dom.DocumentCache
synchronized void replaceDocument(String uri,
    CachedDocument doc) {
  CachedDocument old =
     (CachedDocument)_references.get(uri);
  if (doc == null)
    insertDocument(uri, doc);
  else
    _references.put(uri, doc);
}
```

Figure 5. Mistake in Xalan DocumentCache

```
if (out == null) out.close();
```

Variations on this have shown up in a number of software projects, including in Ant 1.6.5 (MAudit.java, line 303). In this case, if `out` is null, a null pointer exception will be thrown. However, the real worry about this defect is what happens when `out` is nonnull; no exception will be logged or reported and the resource won't be closed (which can cause serious problems). Mistakes that manifest themselves by throwing exceptions are generally preferable to manifestations that silently and occasionally generate corrupted data or performance bottlenecks.

Another example is shown in Figure 5. This code is part of the Apache Xalan project, and has been included in Sun's JDK since Java 1.5. FindBugs reports that the call to `insertDocument` is incorrect, since `insertDocument` requires that its second argument be nonnull. The null check should have been `if (old == null)`.

A more sophisticated static analyzer might detect that at the one place where `replaceDocument` is called, the second argument is always nonnull, and thus decide `replaceDocument` cannot, in fact, pass a bad null parameter to `insertDocument`.

However, it would be useful for a static analysis tool to generate a warning about the fact that `replaceDocument` will never call `insertDocument`. This could be potentially significant, since `insertDocument` contains logic to cap the size of the cache. Calls to `replaceDocument` could cause entries to be added to `_references` without bound.

Fortunately, `replaceDocument` is only called in one place where we've already determined that `_references` contains an entry for `uri`. Thus, unless the entry is removed by another thread between the earlier check and the call to `replaceDocument`, both `old` and `doc` will be nonnull, and the fact that we are null-testing the wrong value will have no impact.

In general, static analyzers can detect not only potential null dereferences, but also inconsistent handling of null. It can be tricky to figure out when such redundant checks should be reported. Sometimes, they are performed after a value has already been dereferenced, reflecting a dangerously inconsistent understanding of whether the value can be null. At Google, over a 9 month period, 572 of 815 such warnings were removed from the codebase [1]. On the other hand, sometimes they represent a conservative nullcheck of a value that never can be null. At Google, over the same 9 month period, only 301 of 2189 such warnings were removed from the codebase. In some cases, such as shown in 5, they can be important even though a null pointer exception is impossible.

5. Conclusions

Overall, the results of our study are that there are no silver bullets, and that the data is intriguing but frustratingly hard to interpret. While null dereferences do cause some execution failures, many

potential null dereferences never manifest themselves in execution. This may be partially due to a "survivor" effect we have noticed in applying static analysis tools in practice. If a potential issue doesn't cause problems in practice, it is more likely to "survive" to the next version of the software. Issues that do cause real problems are more likely to get noticed and fixed, either before the code containing the issue is ever released, or shortly after it is released and used in production. If an issue has persisted in the code for a long period of time, it is likely that it has survived for so long because it isn't causing problems. This is particularly true for potential null pointer exceptions, which tend to leave clear signs when they fail. This contrasts with other defects that silently cause computation to be less efficient than intended or the wrong answer to be computed.

This isn't to say that static analysis isn't important. Just that when applying it to stable code that has been thoroughly tested and used in production, finding the mistakes that *matter* is difficult.

Acknowledgments

We thank Mangala Nanda and Saurabh Sinha for access to their raw data and their XYLEM analyze, and that Fortify and Coverity for access to their static analysis tools.

References

[1] N. Ayewah and W. Pugh. The Google FindBugs Fixit. submitted to ISSTA 2010, 2010.

[2] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *Journal Object Technology*, 6(9):455–475, Oct. 2007.

[3] M. D. Ernst and D. Corward. JSR 308: Annotations on Java types. `http://pag.csail.mit.edu/jsr308/`, Nov. 2007.

[4] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. *SIGPLAN Not.*, 38(11):302–312, 2003. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/949343.949332.

[5] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM. ISBN 1-58113-463-0. doi: http://doi.acm.org/10.1145/512529.512558.

[6] N. Gafter. Elvis and other null-safe operators for java. `http://docs.google.com/Doc?docid=ddb3zt39_78frdf87dc&hl=en`, 2009.

[7] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of jml: a behavioral interface specification language for java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1127878.1127884.

[8] K. R. M. Leino. Specifying and verifying software. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 2–2, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: http://doi.acm.org/10.1145/1321631.1321633.

[9] M. G. Nanda and S. Sinha. Accurate interprocedural null-dereference analysis for Java. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 133–143, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: http://dx.doi.org/10.1109/ICSE.2009.5070515.

[10] M. M. Papi, M. Ali, T. L. Correa, Jr., J. H. Perkins, and M. D. Ernst. Practical pluggable types for Java. In *ISSTA '08: Proceedings of the 2008 international symposium on Software testing and analysis*, pages 201–212, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-050-0. doi: http://doi.acm.org/10.1145/1390630.1390656.

[11] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across versions. In *MSR '06: Proceedings of the 2006 international workshop on Mining software repositories*, pages 133–136, New York, NY, USA, 2006. ACM. ISBN 1-59593-397-2. doi: http://doi.acm.org/10.1145/1137983.1138014.

[12] Univ. of Washington. The checker framework: Custom pluggable type for java. `http://types.cs.washington.edu/checker-framework/`, Jan. 2010.