Tracking Defect Warnings Across Versions

Jaime Spacco[*], David Hovemeyer[†], William Pugh[*]

[*]Dept. of Computer Science
A. V. Williams Building
University of Maryland
College Park, MD 20742 USA

{jspacco,pugh}@cs.umd.edu

[†]Dept. of Computer Science
Vassar College
124 Raymond Ave.
Poughkeepsie, NY 12604 USA

hovemeyer@cs.vassar.edu

ABSTRACT

Various static analysis tools will analyze a software artifact in order to identify potential defects, such as misused APIs, race conditions and deadlocks, and security vulnerabilities. For a number of reasons, it is important to be able to track the occurrence of each potential defect over multiple versions of a software artifact under study: in other words, to determine when warnings reported in multiple versions of the software all correspond the same underlying issue. One motivation for this capability is to remember decisions about code that has been reviewed and found to be safe despite the occurrence of a warning. Another motivation is constructing warning deltas between versions, showing which warnings are new, which have persisted, and which have disappeared. This allows reviewers to focus their efforts on inspecting new warnings. Finally, tracking warnings through a series of software versions reveals where potential defects are introduced and fixed, and how long they persist, exposing interesting trends and patterns.

We will discuss two different techniques we have implemented in FindBugs (a static analysis tool to find bugs in Java programs) for tracking defects across versions, discuss their relative merits and how they can be incorporated into the software development process, and discuss the results of tracking defect warnings across Sun's Java runtime library.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Diagnostics, Symbolic Execution; D.2.2 [**Design Tools and Techniques**]: Programmer workbench

General Terms

Human Factors, Languages, Verification

Keywords

FindBugs, Java, bug histories, bug tracking, static analysis

1. INTRODUCTION

There are many tools that perform static analysis of software to detect possible software defects. Each of these tools looks for some

mixture of security vulnerabilities, coding errors, poor programming practice and style violations.

It is naive to assume that after software is analyzed, the software will be immediately modified to eliminate all of the warnings generated by the static analysis tool. Much more typically, developers will choose, for some reason or another, to change the code in response to only a some of the warnings. Even if a static analysis tool is based on precise and sound reasoning, warnings have to compete with other demands on the developers time, and minor problems may not be worth fixing if there is a chance that a change may introduce an incompatibility or some new, more serious defect.

Thus, when the next version of the software is built and analyzed, many of the warnings will reflect issues from the previous version that were not addressed, while other warnings will correspond to new issues. Being able to pair up warnings generated from analyzing different builds of software is a vitally important task. It is not particularly difficult, but it has not been studied or discussed at length in the literature. In our work on the FindBugs static analysis tool [5, 3] we have implemented techniques and tools for tracking warnings across versions. In recent discussions with Fortify Software [4] we found that their approach to the problem is substantially different than the one we used in FindBugs. We have now implemented both approaches within FindBugs, and in this paper we describe them and report on their relative strengths and weaknesses.

2. THE PROBLEM

First, we need to identify the problem we wish to solve a little more precisely. There are actually a number of similar use cases, which are all largely addressed using the same techniques:

- Assume that a particular version of software was analyzed, generating a list of warnings about potential defects. Some of these warnings were audited, with some of them being flagged as harmless and others being flagged as serious problems (but perhaps not yet fixed). When a new version of the software is analyzed, we want to be able to associate the audit results from the previous analysis with the issues raised by analyzing the current version of the software. Thus, we can ignore the issues previously marked as harmless, and ensure that the ones previously marked as important continue to be flagged as important.

- A similar problem, except more decentralized. The development team might currently be working on build b55, while at the same time one security team is auditing build b48 of the servlet library, and another security team is auditing b50 of the persistence library. How do the two security teams relay their findings to the development team?

- A development team has just started using static analysis tools,

and the tool generates more than 300 serious warnings. The team doesn't have time to review all of the issues, so they want to review only the new warnings—those that did not occur in previous versions of the software.

3. MATCHING WARNINGS IN FINDBUGS

3.1 Pairing

The first and primary form of matching implemented in FindBugs is based on pairing warnings. We start with two sets of warnings. We then try to match up warnings with a progressively "fuzzier" series of WarningMatchers. Each matching object provides a hash function and equivalence predicate for warnings, with a property that for any WarningMatcher m, and warnings w1 and w2, m.equivalent(w1,w2) implies m.hashCode(w1) = m.hashCode(w2). We first start with a very precise warning matcher, which only considers two warnings the same if all recorded details about them are identical.[1]

The first matcher will generally pair up and remove from consideration that vast majority of warnings. We then apply fuzzier matchers, which allow for source lines to vary. If there are collisions (e.g., two warnings from version A and two warnings from version B both match), we pair them up according to their lexicographical order in the warning database, which is determined by the lexicographical order of named elements such as classes, fields, and methods, and by the order of the byte code offsets of any source line references.

As we move to even fuzzier matching algorithms, we look for package renaming. If, in one version, there is a warning in class org.apache.Foo, and in the next version there are no classes in the org.apache package, but there is a new package com.sun.org.apache containing a Foo class, we consider that to be a package renaming and allow warnings in the org.apache.Foo class to be matched to com.sun.org.apache.Foo.

At the moment, we do not try to accommodate refactorings that would move a bug warning from one method to another, even in trivial cases such as renaming a method.

3.2 Warning signatures

A second approach to matching warnings is *warning signatures*. For warning signatures, we compute, for each warning, a string including the names of classes, fields, and methods involved in the warning, but excluding source locations. We then compute the MD5 hash of the string and represent the hash value in hexadecimal so that all warning signatures are a consistent length.

The significant problem here is how to handle collisions: two different warnings producing the same MD5 hash. We do not expect actual MD5 collisions—different strings producing the same MD5 hash—to be an issue. Rather, the problem is what if, for example, the tool finds two possible SQL injections in the same method, such that everything other than the source line offsets (which are ignored by the signature computation) are identical?

As of version 3.5, Fortify Software used a similar mechanism for computing warning signatures, but upon collision they simply rehashed the signature: in other words, computing the MD5 hash of the original MD5 hash. We felt that this was a bad choice, since it makes it exceedingly difficult to recover information about where potential collisions occurred.

In our first implementation of warning signatures in FindBugs (available in FindBugs version 0.9.5), we compute occurrence numbers for each warning. Assuming there are no collisions, each warn-

---

[1] Any source locations here are denoted by bytecode offsets rather than source lines, because bytecode offsets are not affected by changes to methods elsewhere in the file.
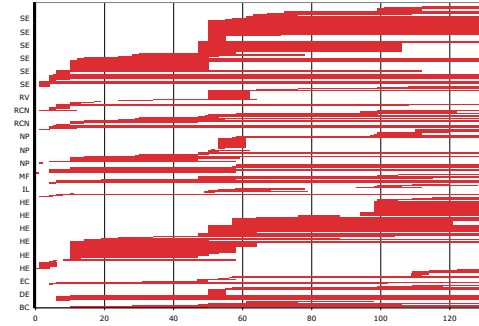
Figure 1: Lifetimes of high priority correctness warnings in Sun's JDK

ing gets an occurrence number of zero. If there are collisions, successive occurrence numbers are generated (in order of byte code offset for the first source line annotation). Thus, by concatenating the warning signature and the occurrence number, we get a string that is guaranteed to be unique for any collection of warnings. When matching warning signatures across versions, we know that any warnings with a non-zero occurrence number indicate a collision, and possible mismatching of warnings.

In reviewing the places where FindBugs can generate multiple warnings per method, we found that in many cases the warnings were all related, and it made sense to only report one such warning per method. Thus, a number of bug pattern detectors were changed so that they would report at most once per method, but each warning would contain a list of all the source lines where the issue arose. This allows correct matching without collisions, even if the number sub-issues or their source line offsets change between versions.

4. RESULTS OF TRACKING DEFECTS USING FINDBUGS

4.1 Sun's JDK

We have a reasonably complete history of the core runtime library (rt.jar) from releases of the Sun Java Development Kit (JDK), including 116 sequential builds, starting with release 1.0.2, and including bi-weekly or weekly beta builds of the 1.5.0 and 1.6.0 JDKs. We analyzed the longest possible sequence of versions where both release date and version number increased monotonically: in other words, once we analyzed b12 of 1.6.0, the first publicly released build, we didn't analyze any later builds in the 1.5 branch.

4.1.1 Warning lifetimes

Figure 1 shows the lifetimes of all the high priority correctness warnings across all versions of the JDK that we analyzed, excluding 5 other miscellaneous warning types that did not fall into any category large enough to depict in the figure. The ticks on the x-axis correspond to successive builds of the JDK. Each horizontal line corresponds to one warning and stretches from the build where the warning was first detected to the last build in which it existed. The defect warnings are grouped by FindBugs bug type (e.g., IL is an infinite recursive loop).

There are several interesting things to note about this diagram. First, there are builds where bulk changes occur; where many defects are introduced or fixed. The JDK development process includes a fair bit of parallel development, and many of these places represent
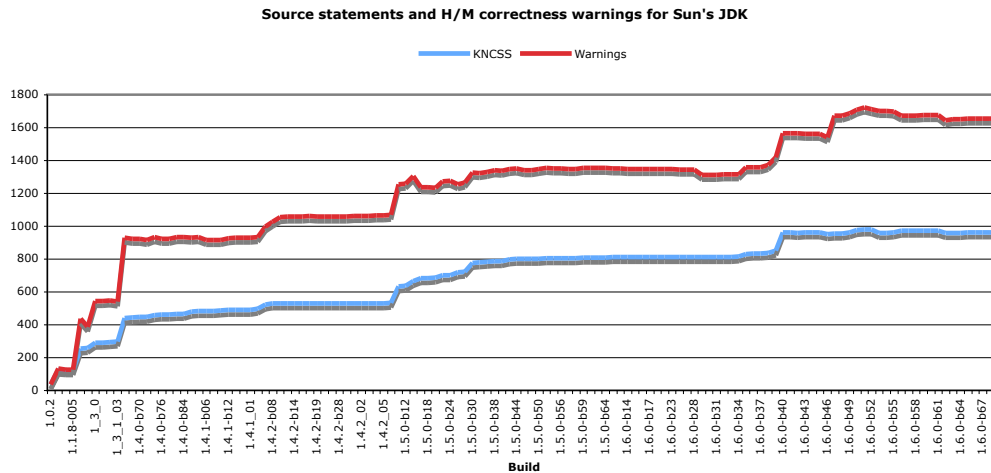
Figure 2: Code size and number of defect warnings in Sun's JDK

places where updates to particular packages are being integrated into the main branch.

Also, the majority of the high-priority defects we found using FindBugs have been fixed. We found one bug pattern, infinite recursive loops, to be so compelling and amusing that we filed bug reports on all of the infinite recursive loops we found. All of those have been fixed, save one. That bug originated in build 1.3.0, more than 5 years ago, and we believe the reason that bug remains unfixed is that the code is stable without an active group developing or maintaining it.

4.1.2 Code size and defect density

Figure 2 shows the size of the JDK builds over time, and the number of both medium and high priority correctness warnings existing in each build.

The size of the JDK builds is given in thousands of non commenting source statements. For most classes, this is computed from the table associated with each method that maps byte code offsets to source line number. Since there is only one entry per statement, this correctly handles whitespace and statements that spread over several lines. In the few cases where we analyze classfiles that do not contain line number tables, we extrapolate from the empirically observed value of 10 bytecodes per non commenting source statement.

Over time, we observe a warning density that grows from about 1 warning per KNCSS in early builds to 2 warnings per KNCSS. However, this does not necessarily reflect that the quality of the JDK codebase has decreased over time. Rather, the warnings in a build correspond both to unfixed defects and false positives. Because false positives accumulate over time (since they do not warrant corrective action to the code) we would expect that the total combined density of false positives and defects would grow over time, even if the defect density remains constant.

4.1.3 Defect warning decay over time

In Figure 3, we show the number of correctness warnings in each build of the JDK that satisfies the following:

- The defect was first reported before 1.4 builds

- The defect did not disappear because the class that contained it disappeared

- The defect was not a warning about classes that are not serializable.

The warnings about non-serializable classes were excluded since a systematic effort was made at Sun to add serialVersionUID fields to all the classes that might need them; this resolved several hundred medium priority issues and would otherwise swamp the results. We exclude the defects in removed classes because their removal gives us very little information about why the warning was removed.

From this, we can see that over time, more than half of the high and medium priority correctness warnings are removed. The fact that a lesser portion of the low priority warnings are removed over time gives us reason to believe that this is due to reasons other than code churn, and that the issues we identify as high and medium priority correctness issues are more likely to be things that developers independently determine to need fixing than low priority warnings.

4.1.4 The java.util experience

The java.util package, which contains various classes such as the Collections libraries, is some of the most carefully scrutinized Java code in Sun's JDK implementation. It has been widely reviewed, and has largely been written by two developers, Joshua Bloch and Martin Buchholz, who are highly skilled and highly dedicated to getting their code correct. They also use and advocate the use of the FindBugs tools, and there has been substantial discussion between the FindBugs team and the maintainers of the java.util package. We file a bug report on every defect that FindBugs finds in the java.util package, and also examine every false positive generated by FindBugs on this package. We have not tuned FindBugs to specifically exclude any false positives we might generate on java.util, although we do look for reasons we report false positives in java.util that might be more widely applicable.

Given this background, it is useful to see how FindBugs performs on the java.util package. In the latest build of JDK 1.6.0 (build 69), the java.util package consists of 273 classes and 18,765 noncommenting source statements; using wc to total lines in source files gives 59,175 source lines. As of build 1.6.0-b69, FindBugs identifies 4 warnings in java.util. Of these, one is a real and serious defect which will be fixed before the 1.6 release. The remaining 3 are false positives. FindBugs generates 23 warnings in previous versions of the java.util that are no longer generated in the current version. These results suggest both

- in comparing the density of both active and dead defect warnings, the java.util package has a defect density less than half of that of the JDK overall,
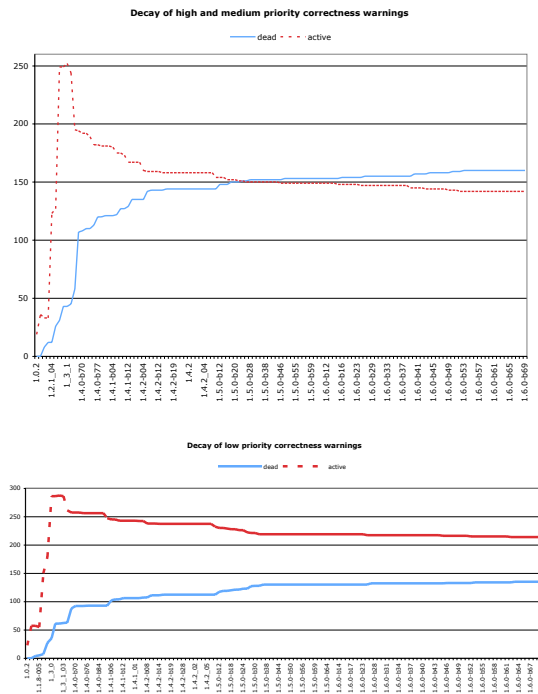
Figure 3: Decay over time of defect warnings introduced before 1.4 builds

- with more attention to improving code quality and improving FindBugs, the ratio that 8 out of 9 of the issues identified by FindBugs are ones that developers believe should be fixed should be more widely reproducible.

5. SOFTWARE EVOLUTION

Two common software engineering practices, creating or merging branches in a repository and renaming packages, create difficulties that the two warning-matching techniques handle differently.

5.1 Non-linear branches

As software evolves, developers need to branch repositories into separate development streams, or merge separate repositories into a single development stream. For example, a version of the JDK-1.4 was branched to provide a starting point for the development of JDK-1.5, while Doug Lea's util.concurrent library was merged into the JDK as java.util.concurrent.

However, after a branch for a new version is created, maintenance on the branch for the old version continues in parallel for some amount of time, often months or year. Similarly, maintenance can continue on a module after it is merged into another repository.

In this environment, a developer who encounters a bug warning needs to know if the same issue occurs in any other branches of the development process, because the issue may already have been fixed or marked as a false positive on a different branch.

The two techniques for matching warnings, *pairing warnings* and *warning signatures*, handle this problem differently, with advantages and disadvantages to each approach.

The algorithm for pairing warnings used by FindBugs was designed with a linear sequence of software versions in mind. Pairing warnings is more fine-grained than warning-signatures in that it can determine, for example, which potential null-pointer dereference in a method was fixed between two versions. This type of fine-grained

information is very useful to a developer actively working on a linear branch who needs detailed information about bug warnings in order to best focus his resources.

However, the pairing implementation currently assumes that the lifespan of a warning is defined by the version in which it is introduced and the last version in which it still exists; there is currently no support for a warning with a set of disjoint lifespans. Thus, as it is currently implemented it would be difficult to use the pairing approach to match warnings between branches of software. In the future we hope to improve the matching algorithm to address this limitation.

The warning-signatures approach used by Fortify Software computes a unique hash for each instance of a warning based on a string representation of the name of the bug pattern and the name of the method and classfile in which it occurs. If there are collisions, the string value is simply re-hashed to resolve the collision. A major benefit is that the hash can easily be used to find warnings in earlier versions of a linear development stream as well as in separate branches of parallel development.

The major drawback to this approach is that collisions cause the analysis to lose information about which bugs are fixed. Because the hashes don't take into account line number information or byte-code offsets, there is no way to determine which bug warning was removed between versions of code: it will always appear as if the second (re-hashed) value was removed.

Thus, the warning-signatures approach is more appropriate for developers who need to fix bugs across branches of software, or who need to integrate branches of software together.

6. CONCLUSION

The issue of matching warnings between versions has only recently been addressed, perhaps because many people assumed the problem was easy and focused their attention on building new bug detectors instead. However, as we've illustrated in this paper, matching warnings is a tricky, interesting problem that needs to be properly addressed for static error checkers to find their way into mainstream software design. In addition to the obvious practical applications (false positive suppression, applying audit results between versions, and construction of warning deltas), studying the lifecycle of defect warnings provides an interesting new perspective on code evolution.

The lack of previous attention paid to the issue is reflected by the fact that several approaches have appeared, each with its own relative advantages and disadvantages, but nobody has studied the various approaches or tried to unify them into a common framework that leverages the advantages of each approach.

7. RELATED WORK

A recent thread on Slashdot [1] discusses the difficulties involved in handling bug reports across branches for bug reporting systems such as Bugzilla [2]. However, we are not aware of published work on this subject.

8. REFERENCES
[1] Bug tracking across multiple code streams?
    `http://ask.slashdot.org/article.pl?sid=05/10/06/2248259&tid=128`, 2006.
[2] bugzilla.org. `http://www.bugzilla.org/`, 2006.
[3] FindBugs—Find Bugs in Java Programs.
    `http://findbugs.sourceforge.net`, 2006.
[4] Fortify Software. `http://www.fortifysoftware.com`, 2006.
[5] D. Hovemeyer and W. Pugh. Finding Bugs is Easy. In *Companion of the 19th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, Vancouver, BC, October 2004.