Which Warnings Should I Fix First?

Sunghun Kim and Michael D. Ernst
Computer Science & Artificial Intelligence Lab (CSAIL)
Massachusetts Institute of Technology
{hunkim, mernst}@csail.mit.edu

ABSTRACT

Automatic bug-finding tools have a high false positive rate: most warnings do not indicate real bugs. Usually bug-finding tools assign important warnings high priority. However, the prioritization of tools tends to be ineffective. We observed the warnings output by three bug-finding tools, FindBugs, JLint, and PMD, for three subject programs, Columba, Lucene, and Scarab. Only 6%, 9%, and 9% of warnings are removed by bug fix changes during 1 to 4 years of the software development. About 90% of warnings remain in the program or are removed during non-fix changes – likely false positive warnings. The tools' warning prioritization is little help in focusing on important warnings: the maximum possible precision by selecting high-priority warning instances is only 3%, 12%, and 8% respectively.

In this paper, we propose a history-based warning prioritization algorithm by mining warning fix experience that is recorded in the software change history. The underlying intuition is that if warnings from a category are eliminated by fix-changes, the warnings are important. Our prioritization algorithm improves warning precision to 17%, 25%, and 67% respectively.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement – *Restructuring, reverse engineering, and reengineering,* D.2.8 [**Software Engineering**]: Metrics – *Product metrics,* K.6.3 [**Management of Computing and Information Systems**]: Software Management – *Software maintenance*

General Terms

Algorithms, Measurement, Experimentation

Keywords

Fault, Bug, Fix, Bug-finding tool, Prediction, Patterns

1. INTRODUCTION

Bug-finding tools such as FindBugs [12], JLint [2], and PMD [6] analyze source or binary code and warn about potential bugs. These tools have a high rate of false positives: most warnings do not indicate real bugs [17]. Most bug-finding tools assign categories and priorities to warning instances, such as *Overflow (priority 1)* or *Empty Static Initializer (priority 3)*. The tools' prioritization is supposed to put important warnings at the top of the list, but the prioritization is not very effective [17]. We performed two experiments that support this observation. Our experiments use three bug-finding tools (FindBugs, JLint, and PMD) and three subject programs (Columba, Lucene, and Scarab).

First, we measured the percentages of warnings that are actually eliminated by fix-changes, since generally a fix-change indicates a bug [7-9, 22]. We select a revision and determine warnings issued by the bug-finding tools. Only 6%, 9%, and 9% of warnings are removed by fix-changes during 1~4 years of the software change history of each subject program respectively – about 90% of warnings either remain or are removed during non-fix changes.

Second, we observed whether the tools' warning prioritization (TWP) favors important warnings. The maximum possible warning precision by selecting high priority warning instances is only 3%, 12%, and 8% respectively. This fact indicates that TWP is ineffective.

Our goal is to propose a new, program-specific prioritization that more effectively directs developers to errors. The new history-based warning prioritization (HWP) is obtained by mining the software change history for removed warnings during bug fixes.

A version control system indicates when each file is changed. A software change can be classified as a fix-change or a non-fix change. A fix-change is a change that fixes a bug or other problem. A non-fix change is a change that does not fix a bug, such as a feature addition or refactoring.

Suppose that during development, a bug-finding tool would issue a warning instance from the *Overflow* category. If a developer finds the underlying problem and fixes it, the warning is probably important. (We do not assume the software developer is necessarily using the bug-finding tool.) On the other hand, if a warning instance is not removed for a long time, then warnings of that category may be neglectable, since the problem was not noticed or was not considered worth fixing.

Using this intuition, we set a weight for each warning category to represent its importance. The weight of a category is proportional to the number of warning instances from that category that are eliminated by a change, with fix-changes contributing more to the weight than non-fix changes. The weight of each category determines its priority. Selecting the top weighted warnings improves precision up to 17%, 25%, and 67% respectively – a significant improvement in warning precision.

In previous research [13], we observed that some warnings are removed quickly, whereas others persist for a long time. Furthermore, we observed that warning lifetime is poorly correlated with tools' priority. We extend the previous work as follows:

- **Use bug-fix information**: We incorporate information regarding bug fixes instead of warning lifetime.
- **Prioritization algorithm**: We propose a prioritization algorithm rather than merely observing the varying lifetimes of warnings.
- **Evaluation**: We use the software change history to classify each warning as a true or false positive. Using that information, we show that the HWP algorithm improves the warning precision.

In the remainder of the paper, we start by presenting background on bug-finding tools (Section 2) and then report our experiment to measure the precision of warnings (Section 3). Section 4 introduces our new HWP algorithm that mines the software change history (Section 4). We discuss our assumptions and threats to the validity (Section 5). We round off the paper with related work (Section 6) and conclusions (Section 7).

2. BACKGROUND

Bug-finding tools for Java such as ESC/Java [10], PMD [6], JLint [2], and FindBugs [12] are widely used [21]. Most bug-finding tools use syntactic pattern matching, model checking, or type checking to identify potential bugs. These tools are good at detecting common bugs such as null pointer dereferencing.

In this research, we use three bug-finding tools, FindBugs, JLint, and PMD. FindBugs analyzes Java bytecode to find pre-defined errors [12]. JLint also analyzes Java bytecode and performs syntactic checking and data flow analysis to find potential bugs [2]. PMD finds syntactic error patterns from source code [6].

Bug-finding tools warn about potential bugs with location information (filename and line number). For example, Figure 1 shows a FindBugs warning example [12]. The warning indicates a potential bug: the bug category is *EI_EXPOSE_REP*, the priority is 2, and the location is line 139 of the ConstructorInfo.java file.

The priority given by tools represents the importance of the warning. If the priority is 1, the tool author believes the warning is likely to indicate a real, important bug. If the priority is 3 or 4, the warning may be neglectable.

EI org.apache.commons.modeler.ConstructorInfo.getSignature() may expose internal representation by returning org.apache.commons.modeler.ConstructorInfo.parameters

Bug type EI_EXPOSE_REP, **Priority: 2**

In class org.apache.commons.modeler.ConstructorInfo
[…]

Field org.apache.commons.modeler.ConstructorInfo.parameters
At **ConstructorInfo.java:[line 139]**

Figure 1. A FindBugs warning example [12].

3. MEASURING WARNING PRECISION

Each warning issued by a bug-finding tool either indicates an underlying issue that is important enough for a developer to notice and consider worth fixing, or it is a false positive (never noticed or not worth fixing). This section describes how we make this determination for each warning instance, and presents results from three subject programs. The intuition is that if a warning is eliminated by a fix change and the warning line number indicates a line that was modified in the bug fix, then the warning category probably indicates a real bug.

We consider a line to be a bug-related if it is modified by a fix-change, since in order to resolve a problem the line was changed or removed. Our approach is to mark each line of a file at a revision as bug-related or clean. A bug may have multiple manifestations or possible fixes, but a bug-finding tool should aim to indicate at least one of those to the developer, and ideally should indicate lines that the developer chooses to fix.

If a warning reports a bug-related line, then the warning is a true positive:

$$precision = \frac{\#of\ \ warnings\ \ on\ bug-related\ \ lines}{\#of\ warnings} \times 100\%$$

Otherwise, the warning is a false positive:

$$false\ \ positive\ \ rate = 100\% - precision$$

Table 1 briefly describes the three analyzed subject programs. They have about 2~5 years of program history with 1,398 ~ 2,483 revisions.

Table 1. Analyzed subject programs. The number *n* is the maximum revision of each program. We evaluate the HWP algorithm at revision *n/2*. *The number of FindBugs warnings in Columba is high and many are *STYLE* related (240 warnings). ** Similarly, Lucene has many race-condition-related JLint warning instances. Still, the HWP algorithm by mining the software history increases warning precision significantly for all three subject programs. † For Scarab, about half of the total revisions (1,241) were committed in one year.

Program	**Software type**	**Period**	**Revision *n/2***			**# of warning instances at revision *n/2***			**# of revisions (*n*)**	**# of compilable revisions**
			Date	**# of files**	**LOC**	**FindBugs**	**JLint**	**PMD**		
Columba	Email Client	11/25/2002 ~ 06/29/2004	09/11/2003	870	121K	*448	509	1,374	1,703	1,486
Lucene	Search Engine	10/19/2001~ 11/9/2006	08/30/2004	233	37K	66	**518	929	1,398	1,160
Scarab	Issue tracker	1/2/2002 ~ 11/8/2006	† 12/10/2002	314	64K	57	556	870	2,483	1,947

3.1 Extracting Software Change History and Warnings

Kenyon [3] is a system that extracts source code change histories from SCM systems such as CVS and Subversion. Kenyon automatically checks out the source code of each revision and extracts change information such as the change log, author, change date, source code, and change deltas.

Kenyon ignores revisions that cannot be compiled (see Table 1). Using the three bug-finding tools, FindBugs, JLint, and PMD, Kenyon gets warning instances for each revision of each subject program.

Usually bug-finding tools allow developers to tune output options to issue more or less warnings. We used the tools' default options for our experiments.

3.2 Fix Changes

We identify fix changes by mining change log messages in the software history. Two approaches for this step are widely used: searching for keywords such as "Fixed" or "Bug" [19] and searching for references to bug reports like "#42233" [7, 9, 22]. We use the former technique. The detailed keywords used to identify fixes for each program are shown in Table 2. Chen et al. studied open source change log quality and their correctness [5]. They checked the correctness of each change log and found that almost all logs are correct.

Some open source projects have strong guidelines for writing change logs. For example, 100% of Columba's change logs used in our experiment have a tag such as '[bug]', '[intern]', '[feature]', and '[ui]'. Lucene and Scarab do not use tags in change logs, but they have good quality change logs.

Table 2. Keywords for fix commit identification.

Program	Fix change identification keywords
Columba	[bug], [bugfix]
Lucene	Patch, fix, bug
Scarab	Patch, fix bug, issue number

3.3 Bug-Related Lines

After identifying fix changes, we observe what lines have been deleted or changed (delta) in these fix changes. A line, *l* is a bug-related line iff *l* is modified or removed during any fix-change.

Suppose there is a fix change between revision 6 and revision 7 as shown at the right of Figure 2. Two lines at revision 6 are fixed and the fixed code appears at revision 7. We mark the two lines at revision 6 as bug-related, since the two lines are modified to resolve a problem. Now, consider the previous revision 5 (assuming the revision numbers are consecutive). If the two marked lines at revision 6 are not changed between revision 5 and revision 6, the bug-related marks in these lines are propagated to the previous revision. If the change between revision 5 and revision 6 is a fix, the modified lines at revision 5 are marked as bug-related. Suppose the change between revision 4 and revision 5 is non-fix. The changed part of the code at revision 4 is not marked as bug-related.

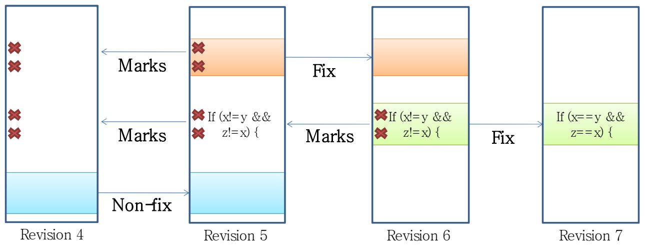


Figure 2. Marking bug-related lines. Code changes are highlighted. Bug-related lines are marked with "**X**".

Starting at the last revision, this process can mark all lines of the files at any revision as either bug-related or clean. Our experiments use revision n/2 (*n* is the maximum revision of each program). The bug-related marks of a revision are used as an oracle set – the marked lines are used to measure the precision of warnings assuming the marks are all correct.

Table 3 shows that 4% to 12% of lines at revision *n/2* are marked as bug-related.

Table 3. Marked bug-related line LOC and percentages, at revision n/2.

Program	LOC (K)	Bug-related marked LOC	Bug-related marked LOC %
Columba	121K	5,336	4%
Lucene	37K	2,608	7%
Scarab	64K	7,899	12%

This marking algorithm is similar to algorithms for identifying buggy changes [15, 22]. While identifying buggy changes focuses on finding when a bug was introduced, our buggy-marking algorithm tries to identify bug-related lines of files at a revision.

3.4 False Positive Rates

To compute false positive rates, we compare the warnings to bug-related lines. If a warning matches any bug-related line, we assume the warning is correct. Otherwise, it is a false positive warning.

For example, suppose lines 3, 4, 6, 8, and 9 are marked as bug-related in a file at revision *n/2* as shown in Figure 3. Suppose a bug-finding tool warns about lines 1, 3, 5, and 8. Then, the precision of warnings is 50% (two correct warnings out of 4 warnings) and recall is 40% (found 2 bug-related lines out of 5 bug-related lines).

In the same way, we compute warning precision for three subject programs, Columba, Lucene, and Scarab. First we mark lines in all files at revision *n/2* (*n* is the maximum revision of each program) as bug-related or clean by mining the software change history between revision *n/2* and revision *n*. Next we run three bug-finding tools (FindBugs, JLint, and PMD) on all files at revision *n/2* to identify warnings. Finally, we compare warnings and marked bug-related lines to compute precision and recall.

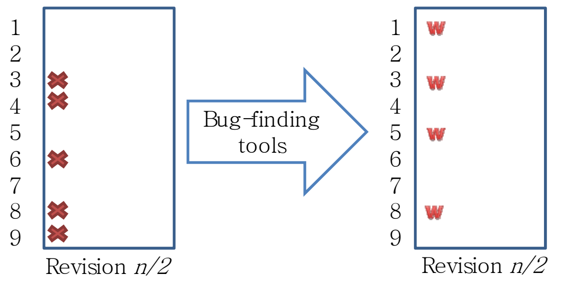


Figure 3. Measuring false positives. On the left, bug-related lines as indicated by mining fix-changes in the software change history. On the right, warnings issued by bug-finding tools.

3.5 Grouping Warnings

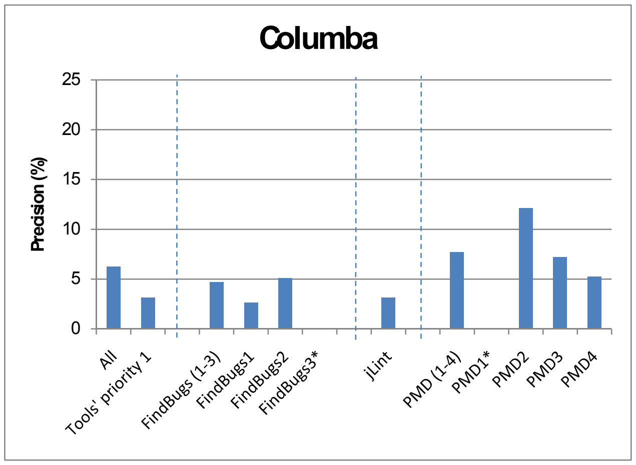
To help understand tool performance, we group warning categories by tool name and warning priorities, as shown in Table 4. The *All* group contains all warning categories output by bug-finding tools. The *Tools' priority 1* group includes *FindBugs 1*, *PMD 1*, and *JLint* warnings. In the same way, we group warning categories by tool and each tool-specific priority levels. For example, the *PMD 2* group includes all priority 2 warnings issued by PMD. Additionally, the *FindBugs(1-3)* group aggregates all warnings issued by FindBugs, and the *PMD(1-4)* group combines all PMD warnings.

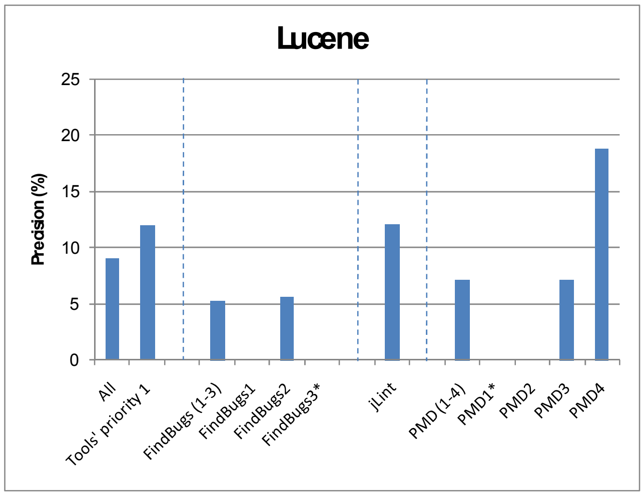
As shown in Table 4, the bug-finding tools have a total of 349 categories (such as *Overflow* or *Empty Static Initializer*). The bug-finding tools issue warnings from 89 categories for Columba at revision *n/2*. Among them, 25 warning categories are set as priority 1 by the tools.

Many warning categories are concentrated in one priority, which limits the ability of the tools to differentiate. For example, most PMD warning categories have priority 3. Similarly, *FindBugs 2* includes most FindBugs warning instances.

Table 4. Number of categories in each group. † The FindBugs warning priorities are context-sensitive and the same warning categories may have different priorities. ‡ By default, FindBugs reports only priority 1 and 2 warnings. * JLint does not provide priority information so we assume all warnings are priority 1.

	Number of categories	Categories observed at revision *n/2*		
		Columba	Lucene	Scarab
All	349	89	54	62
Tools' priority 1	N/A	25	17	20
FindBugs(1-3)	271	47	15	27
FindBugs 1	†	9	2	8
FindBugs 2	†	38	13	19
‡ FindBugs 3	†	0	0	0
* JLint	30	16	15	12
PMD(1-4)	44	26	24	23
PMD 1	1	0	0	0
PMD 2	3	3	1	3
PMD 3	37	22	21	18
PMD 4	3	1	2	2





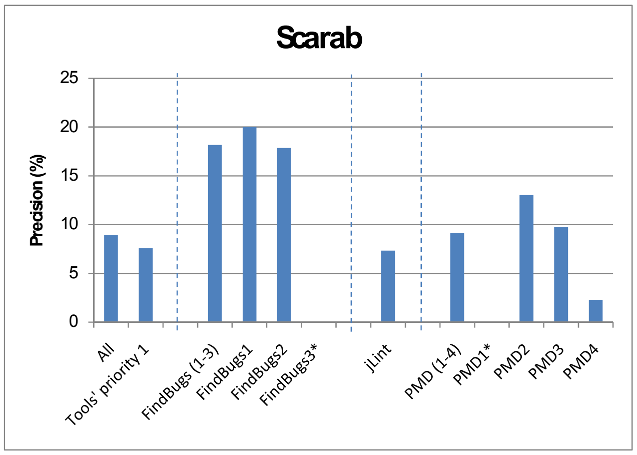


Figure 4. Warning precision for three subject programs. The '*' mark indicates that no warning from the group is issued for the subject program. For example, for the three subject programs, PMD issued no priority 1 warnings.

3.6 Warning Precision and Recall

The precision of each warning group in Table 4 at revision *n/2* is shown in Figure 4. The overall precision in Figure 4 is low, around 6-9%, indicating that over 90% of warnings at revision *n/2* are not fixed by revision *n* (about 1~4 years later).

In the bug prediction literature, line-based prediction is considered a hard problem, and our numbers are consistent with previous results. Most research tries to predict bugs at the module, file, and function level [11, 16, 20]. Line-based prediction precision is about 7.9~16.1% [14].

Figure 4 shows that the prioritization of tools is not very effective. If tools' prioritization were effective, the higher priority warnings would have also high precision. For example, the precision of *tools' priority 1* warnings should be higher (probably much higher) than that of *All*: Columba shows the opposite pattern. Similarly, the precision of *PMD 3* in Lucene is lower than that of *PMD 4*.

Figure 5 shows the recall of all warnings for three subject programs. Recall shows how many bugs (marked as bug-related lines) are caught by warnings. The recall range is 2%~5%.

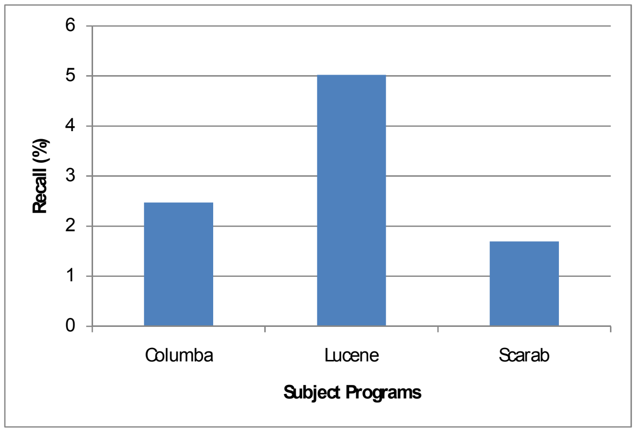


Figure 5. Warning recall of three subject programs.

The reason for this low recall is that line-based prediction is a hard problem, and that tools can only catch certain types of bugs, not all bugs. Recent studies indicate the most prevalent type of bug is semantic or program specific [14, 18]. These kinds of bugs cannot be easily detected by generic bug-finding tools. Most of all, we marked bug-related lines based on fix-changes. Usually a fix-change includes multiple lines, but bug-finding tools may predict only one or two on these lines correctly. This is probably sufficient to help programmers locate and fix the error.

Our goal is improving precision by mining the software change on warnings given by bug-finding tools. Improving recall requires modifying bug-finding tools and is beyond this research.

4. PRIORITIZATION ALGORITHM

We showed that the tools' warning prioritization is not effective at identifying important warnings. How can we find warnings that are more important? Our solution is mining the fix-changes and warning removal experience that is recorded in the software change history.

4.1 Warning Instance and Warning Category

Our technique sets the priority of each warning category (for example, *Zero Operand* is more important than *Empty Static Initializer*), but it does not differentiate among warning instances in the same category (for example, *Zero Operand* on line 10 has the same importance as *Zero Operand* on line 50). Once we set a weight for a warning category, the weight determines the priority of all warnings in the category.

This suggests that our technique will be most effective when the categories are relatively fine-grained and internally homogeneous (with respect to their importance and weight). If a bug-finding tool uses only one warning category, our technique would be unable to differentiate among warning instances, and hence unable to set weights.

4.2 Training Warning Category Weights

Consider three distinct warning categories, *c1*, *c2*, and *c3*. *c1* is often removed by fix changes, *c2* is removed by non-fix changes, and *c3* is not removed for a long time. We assume *c1* is more important or relevant to fixes (bugs), and *c3* is less likely a real bug or developers do not bother to remove the warnings.

The basic idea of training weights for categories is taking each warning instance as a bug predictor. If the prediction is correct (the warning is eliminated by a fix-change), we promote the weight.

We exclude removed warnings due to any file deletion. If there is a file deletion during a fix, all warnings in the files are removed. These all removed warnings are not necessarily the results of the fix.

The HWP algorithm is described in Figure 6. The initial weight w_c of category *c* is set to 0. After that, if a warning in a category *c* is removed during a fix change, we promote the weight by α. Similarly, if a warning in a category *c* is removed during a non-fix change, we increase the weight by β. Since there are only promotion steps, the warning category weights are decided by the ratio of α and β rather than the actual values of α and β. We make α an independent variable and β a dependent variable on α (i.e. $\beta = 1-\alpha$ and $0 \le \alpha \le 1$).

```
// initialize weight w_c
$w_c = 0$
for each warning instance i in category C
    // fix-change promotion step
    if i is removed in a fix change
        then $w_c = w_c + \alpha$
    // non-fix change promotion step
    if i is removed in a non-fix change
        then $w_c = w_c + \beta$
// weight normalization step
$w_c = w_c / |C|$ where |C| is the number of warning instances in category C
```

Figure 6. The HWP algorithm by mining the software change history.

A warning category gets a high weight if warning instances from the category are removed many times by fix-changes or non-fix changes. In contrast, a warning category gets a low weight if warning instances from the category are seldom removed.

This algorithm is inspired by the Weighted majority voting and Winnow online machine learning algorithms [1]. These algorithms take features from learning instances. Each feature has a weight and the weight is adjusted based on the feature's prediction result. If a feature's prediction is correct, the feature weight is promoted. Otherwise, the feature's weight is demoted. These simple online machine-learning algorithms work well in practice.

The next step of our algorithm is weight normalization. Consider the following two warning categories shown in Figure 7. There are 9 warning instances (i_1) from the c_1 category, and there is one warning instance (i_2) from the c_2 warning category. Two warning instances from c_1 and one warning instance from c_2 are removed during a fix change shown in Figure 7

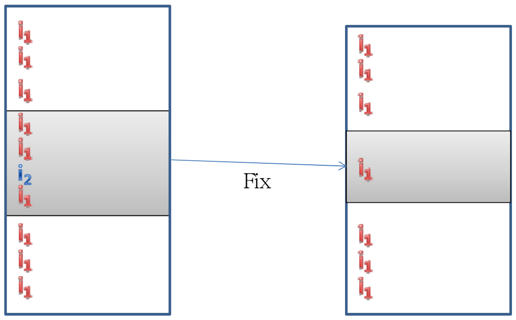


Figure 7. Warning instances from two categories are removed by a single fix change.

In the prioritization algorithm as described, the quantity of a warning category dominates the weight. For example if α =1 then w_1 is 2 and w_2 is 1, since two warning instances from c_1 are removed. In fact, the precision of c_2 is higher than that of w_1 in this example.

In order to avoid this problem, we normalize weights by the total number of warning instances of the category shown in Figure 6.

4.3 Evaluation Method

To evaluate our algorithm, we need to train weights for each warning category by mining the software change history during a training period: revision 1 to revision *n/2-1*. To measure the precision of the weights, we also use the software change history, but during a testing period: and revision *n/2* to revision *n*. To be a fair comparison, the two periods should not overlap.

4.4 Selecting α

As described in Section 4.2, the HWP algorithm uses a variable, $\alpha(\beta = 1-\alpha)$, that affects the category weights. Some examples are:

- $\alpha = 0.5, \beta = 0.5$: promotion for all changes equally.
- $\alpha = 1, \beta = 0$: promotion only for fix-changes
- $\alpha = 0.9, \beta = 0.1$: promotion for all changes. More promotion for fix-changes.

We experimentally determined that the best α for Columba, Lucene, and Scarab is 0.8, 0.9, and 1 respectively. We use α = 0.9 for the rest of experiments. In Figure 8, the y-axis indicates the precision of the top 30 warning instances, as weighted by HWP.

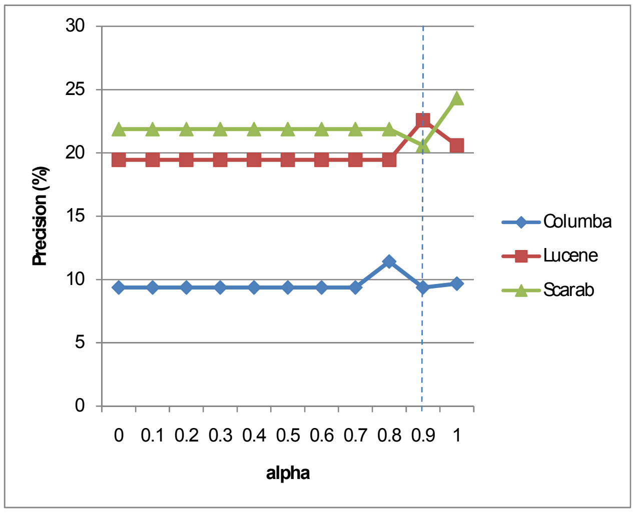


Figure 8. The top 30 warning instance precision for various α values at revision n/2. α = 0.9 is used for the rest of experiments.

4.5 Evaluation Results

After computing a weight for each warning category, we prioritized warning instances according to their category weights.

Figure 9 plots precision for the top 100 warning instances when all warnings output by the tools (when applied to revision *n/2* of the given subject program) are sorted by their priority – either TWP or HWP. For example, for the Columba subject program, the precision is 17% for selecting the 12 highest-priority warning instances according to HWP. By contrast, the precision is about 3% for the 12 highest-priority warning instances according to the tools' own built-in prioritization. When multiple warnings have the same priority, the figures use the expected value of precision if warnings of that priority are chosen at random.

Programmers are unlikely to look at all warnings output by a tool, so the most important part of the graphs is the portion near the origin, which indicates how precise the highest-prioritized warnings are.

Figure 9 demonstrates that our prioritization outperforms the built-in prioritization, often by a substantial margin. For our prioritization, the best precision for a subset of the warnings is 17% (top 12 warnings for Columba), 25% (top 16 for Lucene), and 67% (top 6 for Scarab) for the three subject programs respectively. For the top 30 warnings, the TWP precision is 3%, 12%, and 8% respectively. HWP improves this to 9%, 23%, and 21% respectively.

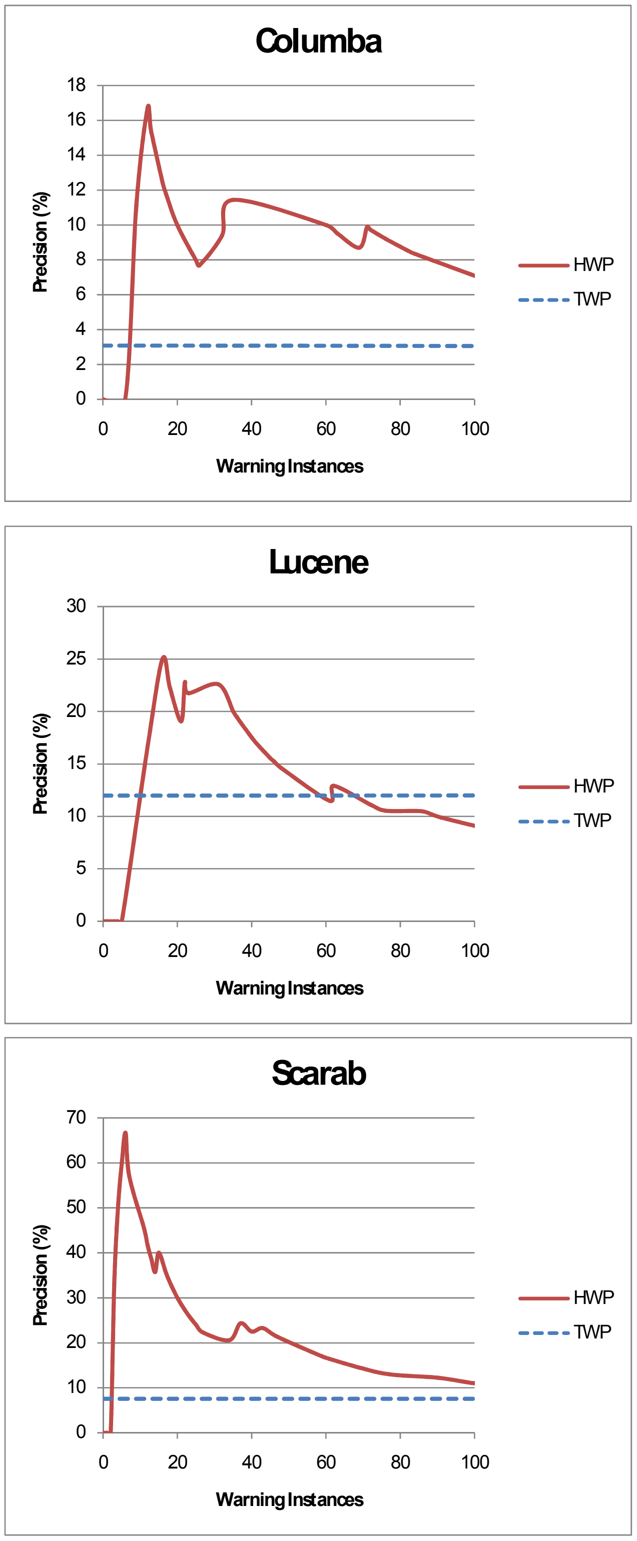


Figure 9. Precision of the top 100 warning instances sorted by HWP and TWP.

Table 5 and Table 6 compare the top priority warnings by HWP and TWP for Scarab at revision n/2. The top warnings from TWP and HWP are very different. The occurrences indicate the number of warning instances from the corresponding category for Scarab at revision n/2.

5. DISCUSSION

In this section, we discuss some limitations of the HWP algorithm and experiment results.

5.1 Homogeneous Categories

We assume that warning instances in a warning category are homogeneous with respect to importance and precision. The jitters in Figure 9 suggest that some categories are not homogenous. Some instances in a category are removed many times by fix-changes between revision *1* and revision *n/2-1*, but between revision *n/2* and revision *n* the instances are not removed by fix changes. After about 45 warnings for Lucene, the precision of HWP is worse than that of the prioritization of the tools. Adding warning instances from one or two non-homogeneous categories may reduce the precision significantly.

5.2 Initial Mining Data

The HWP algorithm uses the change history to train weights of warning categories. In order for a category to have a non-zero weight at least one instance from the category must be removed. If an instance from a new warning category appears, we do not have any weight information for the warning. In this paper, we used about 1~2 years of the program history to train weights of warning categories which yields a reasonable precision improvement.

5.3 Threats to Validity

We note some threats to the validity of this work.

The subject programs might not be representative. We examined 3 open source systems written in Java, and it is possible that we accidentally chose subject programs that have better (or worse) than average warning precision. Since we intentionally chose programs whose change log text indicates bug-fix changes, there is a program selection bias.

Some developers may use bug-finding tools in their development cycle. If individual developers use FindBugs, JLint, or PMD, and fix warnings issued by bug-finding tools, then the measured precision of the tool will be high. We queried developers of each subject program, and developers of Columba and Lucene confirmed that they are not using any bug-finding tools. It is also possible that our results indicate that programmers in different projects are culturally attuned for making, finding, or removing certain types of errors. All of our subject programs are open-source systems written in Java, with 12, 24, and 29 developers making commits, respectively.

Bug fix data is incomplete. Even though we selected programs that have change logs with good quality (see Section 3.), developers might forget to mark some changes as a bug fix in the history or use different keywords to indicate fix changes. Partial bug-fix data may decrease warning precision. It is also possible that developers check in both a fix and many other changes in a commit, or incorrectly mark a non-fix change as a fix. This could lead us to mark too many lines as bug-related, leading to unrealistically high precision.

Table 5. Categories of all property-1 warnings issues by bug-finding tools for Scarab at revision *n/2* (20 categories in all). The occurrences indicate the number of warning instances from the corresponding category at revision *n/2*.

Tool	Tools' priority	Category	HWP weight	Precision (%)	Occurrences
FindBugs	1	correctness: il infinite recursive loop	0.211	0	2
FindBugs	1	correctness: np null param deref nonvirtual	1.1	0	1
FindBugs	1	correctness: rcn redundant nullcheck would have been a npe	0.222	100	1
FindBugs	1	correctness: rv return value ignored	0	0	1
FindBugs	1	malicious code: ms should be final	0.125	0	2
FindBugs	1	performance: dm gc	0.3	0	1
FindBugs	1	style: dls dead local store	0.238	0	1
FindBugs	1	style: st write to static from instance method	0.475	100	1
jLint	1	bounds: maybe bad index	0.175	0	2
jLint	1	bounds: maybe neg len	0.175	0	2
jLint	1	field redefined: field redefined	0	1	184
jLint	1	not overridden: hashcode not overridden	0.046	0	8
jLint	1	not overridden: not overridden	0.001	0	98
jLint	1	null reference: null param	0.122	14	7
jLint	1	null reference: null var	0.007	33	36
jLint	1	race condition: concurrent access	0.013	0	28
jLint	1	race condition: concurrent call	0.004	1	67
jLint	1	redundant: same result	0.01	0	3
jLint	1	shadow local: shadow local	0.006	30	61
jLint	1	weak cmp: weak cmp	0.016	18	22

Table 6. Categories of the top 20 warnings by HWP for Scarab at revision *n/2* (20 categories in all).

Tool	Tools' priority	Category	HWP weight	Precision (%)	Occurrences
PMD	3	java.lang.string rules: stringtostring	2.8	0	1
FindBugs	1	correctness: np null param deref nonvirtual	1.1	0	1
FindBugs	2	style: st write to static from instance method	0.475	100	1
FindBugs	2	style: rec catch exception	0.475	100	1
FindBugs	2	malicious code: ms pkgprotect	0.475	100	1
FindBugs	1	style: st write to static from instance method	0.475	100	1
FindBugs	1	performance: dm gc	0.3	0	1
FindBugs	2	performance: dm string tostring	0.288	25	4
FindBugs	1	style: dls dead local store	0.238	0	1
FindBugs	2	performance: sbsc use stringbuffer concatenation	0.222	0	1
FindBugs	2	correctness: rcn redundant nullcheck would have been a npe	0.222	0	1
FindBugs	1	correctness: rcn redundant nullcheck would have been a npe	0.222	100	1
FindBugs	1	correctness: il infinite recursive loop	0.211	0	2
jLint	1	bounds: maybe neg len	0.175	0	2
jLint	1	bounds: maybe bad index	0.175	0	2
FindBugs	2	mt correctness: is2 inconsistent sync	0.175	0	2
FindBugs	2	correctness: sa field self assignment	0.175	0	2
FindBugs	1	malicious code: ms should be final	0.125	0	2
jLint	1	null reference: null param	0.122	14	7
FindBugs	2	performance: dm string ctor	0.112	67	3

Some revisions are not compilable. Some bug-finding tools such as JLint and FindBugs take jar or class files to generate warnings. This requires compilation of each revision. Unfortunately, some revisions are not compilable. To get warning changes along with source code change, two consecutive revisions should be compilable. It is possible to combine revisions, but then we have to merge change logs and separate changes in each revision to identify fix-changes. To make our experiments simple, we ignore revisions that are not compilable and this may affect the warning precision.

Latent bugs: Maybe some important bugs never got noticed or fixed between revision *n/2* and revision *n*, even though a tool would have indicated them. This would make the precision and recall results for such a tool too low. We suspect that most important bugs will be noticed and fixed over a 1~ 4-year period.

Prioritization and bug severity: The tools' priority may be based not on the likelihood that the warning is accurate, but on the potential severity of the fault. For example, maybe priority 1 warnings have a high false positive rate, but the true positives are so critical that the tool writers placed them in priority 1. Such a policy would probably be counterproductive, since it is widely reported that false positive rates for the first-presented warnings are critical to user acceptance of a bug-finding tool. If this is true, then HWP which ranks each bug fix equality is orthogonal to TWP.

Warning location vs. fix location: Different warning and fix locations may inflate the warning false positive rate. Additionally, Adding new code may fix an existing warning. For example, if a warning is about unused import statements in Java, it could be fixed by adding code that uses the imports.

6. RELATED WORK

Kremenek and Engler [17] prioritize checks (warning categories) using the frequency of check results. Software checkers output success (indicating a successful check of an invariant) or failure (the invariant did not hold). If the ratio of successes to failures is high, the failures are assumed to be real bugs. If we apply this idea to warning categories, a warning category that has fewer warning instances is important. Our HWP algorithm is different from their approach in that we use previous warning fix experience (the ratio of true positives) to identify important warnings.

Boogerd and Moonen [4] use execution likelihood analysis to prioritize warning instances. For each warning location, they compute the execution likelihood of the location. If the location is very likely to be executed, the warning gets a high priority. If the location is less likely to be executed, the warning gets a low priority. This technique may help developers to focus on warnings at the location which has high execution likelihood. However, a location which has low execution likelihood or warnings at the location could be important. In fact, severe bugs in lines with low chance of execution are more difficult to detect.

Williams and Hollingsworth use software change histories to improve existing bug-finding tools [24]. When a function returns a value, using the value without checking it may be a potential bug. The problem is that there are too many false positives if a bug-finding tool warns all source code that uses unchecked return values. To remove the false positives, Williams and Hollingsworth use the software histories and find what kinds of function return values must be checked. For example, if the return value of '*foo*' was always checked in the software history, but not checked in current source code, it is very suspicious. This approach is similar to ours, since they leverage the software history to remove false positives. However, they only focus on the small sets bug patterns such as return value checking, while our approach is generic to all warnings.

Spacco et al. [23] observed FindBugs warning categories across software versions. They measure lifetimes of warning categories, warning number rates, and the degree of decay using warning numbers. Kim et al. [13] observed warning lifetimes by observing warnings in each revision and suggested using the lifetime for reprioritizing warnings. As noted in Section 1, those approaches just reported the observed results, while we use the observation results to prioritize warning categories and evaluate the prioritized warnings.

7. CONCLUSIONS AND FUTURE WORK

We compared warnings and bug fixes in the software change history of three subject programs, Columba, Lucene, and Scarab. Only 6%, 9%, and 9% of the warnings issued by bug-finding tools are removed by a fix-change within about 1~4 years of the software change history. Over 90% of warnings remain in the program or are removed during non-fix changes. Only 3%, 12%, and 8% of tools' high priority warnings (priority 1) are eliminated by fix-changes. This fact indicates that the prioritization of bug-finding tools is not effective.

We proposed an automated history-based warning prioritization (HWP) algorithm that mines previous fix and warning removal experience that is stored in the software change history. If a warning instance from a warning category is eliminated by a fix-change, we assume that this warning category is important. For Columba, Lucene, and Scarab, selecting the top HWP warnings improves precision up to 17%, 25%, and 67% respectively – a significant precision improvement.

This research makes the following contributions:

- **Measuring false positive warnings**: By mining fix-changes in the software change history, we measured the precision (true and false positive rates) of warnings issued by bug-finding tools.
- **Line-based bug evaluation**: In the bug prediction literature, mostly module, file, and function level entities are used to train a prediction model and evaluate the model. By mining the fix-changes in the software history, we mark each line as bug-related or clean. We use the marked lines as an oracle set for evaluating bug prediction.
- **Generic warning prioritization algorithm**: The proposed automated warning prioritization algorithm is generic and is applicable to any warnings.
- **Finer-grained prioritization**: We observe that the prioritization of tools is coarse grained and a prioritization group (such as FindBugs 2 and PMD 3 in Table 4) includes many warning categories. There is no way to prioritize warning categories in the same priority group. Our prioritization provides finer-grained priorities so that it is possible to select an arbitrary number of important warnings.
- **Leveraging the software change history for warning prioritization:** Our approach uses the software change history for warning prioritization. We show that the software change history is useful for warning prioritization.

Our prioritization algorithm increases the precision of warnings significantly. Even so, we still see room for improvement. Combing tools' priorities with HWP priorities may lead to better precision. We analyzed three subject programs and thee bug-finding tools; more programs and tools should be analyzed.

Overall, we expect that future approaches will use the software change history for warning prioritization – and as a source for continued correction and adaptation of warning priorities.

8. ACKNOWLEDGEMENTS

We thank E. J. Whitehead, Jr., Yoav Zibin, Miryung Kim, Andreas Zeller, and the ESEC/FSE reviewers for their valuable feedback on earlier revisions of this paper. This work was supported by DARPA contracts FA8750-06-2-0189 and HR0011-07-1-0023 and by NSF grant SOD-HCER 0613793.

9. REFERENCES

[1] E. Alpaydin, *Introduction to Machine Learning*: The MIT Press, 2004.

[2] C. Artho, "Jlint - Find Bugs in Java Programs," 2006, http://jlint.sourceforge.net/.

[3] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey, "Facilitating Software Evolution with Kenyon," Proc. of *the 2005 European Software Engineering Conference and 2005 Foundations of Software Engineering (ESEC/FSE 2005)*, Lisbon, Portugal, pp. 177-186, 2005.

[4] C. Boogerd and L. Moonen, "Prioritizing Software Inspection Results using Static Profiling," Proc. of *the Sixth IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2006)*, Philadelphia, PA, USA, pp. 149-160, 2006.

[5] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller, "Open-Source Change Logs," *Empirical Software Engineering*, vol. 9, no. 3, pp. 197-210, 2004.

[6] T. Copeland, *PMD Applied*: Centennial Books, 2005.

[7] D. Cubranic and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," Proc. of *25th International Conference on Software Engineering (ICSE 2003)*, Portland, OR, USA, pp. 408-418, 2003.

[8] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 446-465, 2005.

[9] M. Fischer, M. Pinzger, and H. Gall, "Populating a Release History Database from Version Control and Bug Tracking Systems," Proc. of *19th International Conference on Software Maintenance (ICSM 2003)*, Amsterdam, Netherlands, pp. 23-32, 2003.

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, "Extended Static Checking for Java," Proc. of *the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI 2002)*, Berlin, Germany, pp. 234-245, 2002.

[11] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, "Predicting Fault Incidence Using Software Change History," *IEEE Transactions on Software Engineering*, vol. 26, no. 7, pp. 653-661, 2000.

[12] D. Hovemeyer and W. Pugh, "Finding Bugs is Easy," Proc. of *the 19th Object Oriented Programming Systems Languages and Applications (OOPSLA '04)*, Vancouver, British Columbia, Canada, pp. 92-106, 2004.

[13] S. Kim and M. D. Ernst, "Prioritizing Warning Categories by Analyzing Software History," Proc. of *Int'l Workshop on Mining Software Repositories (MSR 2007)*, Minneapolis, MN, USA, pp. 27, 2007.

[14] S. Kim, K. Pan, and E. J. Whitehead, Jr., "Memories of Bug Fixes," Proc. of *the 2006 ACM SIGSOFT Foundations of Software Engineering (FSE 2006)*, Portland, OR, USA, pp. 35-45, 2006.

[15] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead, Jr., "Automatic Identification of Bug Introducing Changes," Proc. of *21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, Tokyo, Japan, 2006.

[16] S. Kim, T. Zimmermann, E. J. Whitehead, Jr., and A. Zeller, "Predicting Bugs from Cached History," Proc. of *the 29th International Conference on Software Engineering (ICSE 2007)*, Minneapolis, USA, pp. 489-498, 2007.

[17] T. Kremenek and D. R. Engler, "Z-ranking: Using statistical analysis to counter the impact of static analysis approximations," Proc. of *the 10th International Symposium on Static Analysis (SAS 2003)*, San Diego, CA, USA, pp. 295-315, 2003.

[18] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have Things Changed Now? An Empirical Study of Bug Characteristics in Modern Open Source Software," Proc. of *1st Workshop on Architectural and System Support for Improving Software Dependability*, San Jose, CA, USA, pp. 25-33, 2006.

[19] A. Mockus and L. G. Votta, "Identifying Reasons for Software Changes Using Historic Databases," Proc. of *16th International Conference on Software Maintenance (ICSM 2000)*, San Jose, CA, USA, pp. 120-130, 2000.

[20] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Where the Bugs Are," Proc. of *2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, Boston, Massachusetts, USA, pp. 86-96, 2004.

[21] N. Rutar, C. B. Almazan, and J. S. Foster, "A Comparison of Bug Finding Tools for Java," Proc. of *15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04)*, Saint-Malo, Bretagne, France, pp. 245-256, 2004.

[22] J. Śliwerski, T. Zimmermann, and A. Zeller, "When Do Changes Induce Fixes?" Proc. of *Int'l Workshop on Mining Software Repositories (MSR 2005)*, Saint Louis, Missouri, USA, pp. 24-28, 2005.

[23] J. Spacco, D. Hovemeyer, and W. Pugh, "Tracking Defect Warnings Across Versions," Proc. of *Int'l Workshop on Mining Software Repositories (MSR 2006)*, Shanghai, China, pp. 133-136, 2006.

[24] C. C. Williams and J. K. Hollingsworth, "Automatic Mining of Source Code Repositories to Improve Bug Finding Techniques," *IEEE Trans. Software Engineering*, vol. 31, no. 6, pp. 466-480, 2005.