Automatic Construction of an Effective Training Set for Prioritizing Static Analysis Warnings

Guangtai Liang[1,2], Ling Wu[1,2], Qian Wu[1,2], Qianxiang Wang[1,2], Tao Xie[3], Hong Mei[1,2]

[1] Institute of Software, School of Electronics Engineering and Computer Science

[2] Key Laboratory of High Confidence Software Technologies (Peking University), Ministry of Education

Peking University, Beijing, 100871, China

{lianggt08, wuling07, wuqian08, wqx, meih}@sei.pku.edu.cn

[3] Department of Computer Science, North Carolina State University, Raleigh, NC 27695, USA

xie@csc.ncsu.edu

ABSTRACT

In order to improve ineffective warning prioritization of static analysis tools, various approaches have been proposed to compute a ranking score for each warning. In these approaches, an effective training set is vital in exploring which factors impact the ranking score and how. While manual approaches to build a training set can achieve high effectiveness but suffer from low efficiency (i.e., high cost), existing automatic approaches suffer from low effectiveness. In this paper, we propose an automatic approach for constructing an effective training set. In our approach, we select three categories of impact factors as input attributes of the training set, and propose a new heuristic for identifying actionable warnings to automatically label the training set. Our empirical evaluations show that the precision of the top 22 warnings for Lucene, 20 for ANT, and 6 for Spring can achieve 100% with the help of our constructed training set.

Categories and Subject Descriptors

D.2.4 **[Software/Program Verification]**: Reliability, Statistical methods; F3.2 [**Semantics of Programming Languages**]: Program analysis; G.3 [**Probability and Statistics**]: Correlation and regression analysis

General Terms

Algorithms, Experimentation, Measurement

Keywords

Static analysis tools, warning prioritization, training-set construction, generic-bug-related lines

1. INTRODUCTION

Lightweight static analysis tools such as FindBugs [15], PMD [24], Jlint [1], and Lint4j [22] aim at detecting generic bugs by analyzing source code or bytecode against pre-defined bug patterns without executing the program. Compared with formal verification techniques such as model checking and theorem proving, these bug-pattern-based tools use lightweight analysis techniques, and they are effective in detecting generic bugs in large software [10]. However, there are two main challenges for these existing tools: (1) their reported warnings often have a high false-positive rate [18, 21, 17, 2], and (2) even if some warnings reveal true bugs, they are not always acted on by developers [25]. Researchers pointed out that more efforts should be spent on refining these tools' warning reports [6].

To address these challenges faced by existing tools, various prioritization approaches have been proposed to reorder warnings by assigning each warning with a ranking score [18, 25, 21, 32, 17]. To calculate the ranking score for each specific warning, different approaches use different impact factors, e.g., warning category [18, 25], warning priority [25], warning accuracy [17], code features [25, 17], and code locality [3]. However, in order to generate an accurate ranking score for each warning, assigning reasonable weights for different impact factors of the ranking score is needed but challenging.

A training set, also called a sample set, plays a key role in learning weights for different impact factors [18, 25, 32, 20, 16]. A training set consists of a vector of input attributes (multiple impact factors for a specific warning) and an output attribute (the warning being actionable or not). A training set is used to train a predictor, e.g., a neural network or a naïve Bayes classifier [31], which learns and records a weight for each impact factor.

A training set can be constructed manually or automatically. A manual approach usually achieves high effectiveness but suffers from low efficiency (i.e., high cost) [25, 16], while an automatic approach achieves high efficiency but suffers from low effectiveness [18, 32, 20]. Given that software-related data is growing rapidly [4], an automatic approach with higher effectiveness is highly desirable.

For example, open-source projects usually have thousands of revisions stored in their source-code repositories. Among these revisions, "bug-fix revisions" are those revisions aiming at fixing bugs and they can be further divided into two groups: generic ones and project-specific ones. "Generic-bug-fix revisions" are responsible for fixing generic bugs while "project-specific-bug-fix revisions" are responsible for fixing project-specific bugs. Generic bugs are those bugs that appear across projects such as "dead lock", "null pointer dereference", and "resource leak". Most static analysis tools can report only generic bugs unless developers

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ASE'10, September 20–24, 2010, Antwerp, Belgium.

Copyright 2010 ACM 978-1-4503-0116-9/10/09...$10.00.

write project-specific bug patterns. The lines modified in generic-bug-fix revisions are called "generic-bug-related lines" and the lines modified in project-specific-bug-fix revisions are called "project-specific-bug-related lines". A warning reported by static analysis tools may have multiple manifestations or multiple possible fixes, but a good static analysis tool should be able to indicate at least one of them to a developer, and ideally should indicate lines that the developer can choose to fix. Therefore, these "generic-bug-related lines" can be used to evaluate static analysis warnings and then construct a training set for warning prioritization. However, a previous approach by Kim and Ernst [18] used all bug-related lines directly to evaluate these warnings. Actually we found that more than 60% of bug-related lines computed by their approach were related to project-specific bugs (which static analysis tools cannot detect or report). Those project-specific-bug-related lines need to be eliminated when being used to evaluate static analysis warnings; otherwise, effectiveness would be greatly compromised, as shown in our evaluation (Section 5).

In this paper, we propose an automatic approach to construct an effective training set for warning prioritization, which is based on "**generic-bug-related lines**". To the best of our knowledge, we are the first to point out the importance of "generic-bug-related lines" in evaluating static analysis warnings. In addition, we also propose an automatic technique for identifying "generic-bug-related lines".

In order to automatically construct a training set for warning prioritization using "generic-bug-related lines", we need to address the following challenges:

(1) How to automatically identify "generic-bug-fix revisions" from thousands of revisions of open-source projects?
(2) How to accurately identify "generic-bug-related lines" for a specific revision?
(3) How to construct a training set based on "generic-bug-related lines" (i.e., how to extract input attributes and label output attributes for the training set)?

To address the first challenge, we propose a technique based on natural language processing (NLP) to assist in identifying "generic-bug-fix revisions" effectively. To address the second challenge, we propose a new algorithm to effectively identify accurate "generic-bug-related lines". To address the third challenge, we select three categories of impact factors as a warning's input attributes: descriptors of the warning, statistics of warnings from different tools, and features of the buggy source code related to the warning; we propose a new heuristic to label actionable warnings: a warning is labeled as actionable if it disappears in later revisions and is revised during a generic-bug-fix revision.

Based on our approach, we develop and release an online Code Defect Analysis Service (CODAS: http://codas.seforge.org), which integrates multiple static analysis tools (including FindBugs, PMD, Jlint, and Lint4j) and prioritizes warnings with the help of our predictor trained with our effective training set.

This paper makes the following main contributions:

- The first to identify the importance of "generic-bug-related lines" and use them to automatically construct an effective training set for warning prioritization.
- New techniques to identify "generic-bug-fix revisions" and "generic-bug-related lines".
- Empirical evaluations, using open-source projects, which show the effectiveness of our training-set construction.
- A publicly available online defect analysis service, which integrates four static analysis tools and prioritizes warnings with a predictor trained with our effective training set.

In the rest of this paper, Section 2 presents background on software repositories, static analysis tools, and machine learning workbench. Section 3 presents an overview of our automatic approach of building a training set. Section 4 describes the details of our training-set construction. Section 5 describes our evaluation results. Section 6 discusses related work and Section 7 concludes.

2. BACKGROUND

2.1 Software Repositories

In recent years, many software projects publish their software repositories over the Internet. These open-source projects provide sufficient historical data for building a training set. Our approach computes generic-bug-fix information for subject projects, and uses the information to label the training set (by determining whether the warnings are actionable or not). We retrieve bug-fix information with the help of software repositories, such as source-code repositories (e.g., CVS or SVN) and issue-tracking systems.

A source-code repository keeps track of changes performed on source code: who changed what, when, why, and how. A change transforms an old revision r1 to a new revision r2 by inserting, deleting, or modifying lines. Source-code repositories handle revisions of textual files by storing the difference between revisions. When comparing two different files, they use diff tools to find groups of differing lines [23].

2.2 Static Analysis Tools

Static analysis tools for Java, such as FindBugs [15], Fortify [13], PMD [24], Jlint [1], and Lint4j [22], are widely used in recent years [29]. These tools use bug-pattern-based matching techniques to detect potential bugs and report warnings.

FindBugs [2] generates warnings for 286 bug patterns [18] and assigns each warning with a priority (e.g., "High", "Medium", or "Low") according to its severity and accuracy. PMD [24] discovers suspicious or abnormal coding practices, which may imply serious bugs, by searching syntactic errors and stylistic conventions' violations from source code. Jlint [1] finds bugs, inconsistencies, and synchronization problems by doing data flow analysis and building lock graphs. Lint4j [22] detects issues about locking and threading, problems about performance and scalability, and violations against complex contracts such as Java serialization by performing different analyses (e.g., type, data flow, and lock graph analyses) against bytecode or source code.

2.3 Machine Learning Workbench

In our approach, we use machine-learning-based predictors to provide ranking scores for warnings. There are two phases for machine-learning-based approaches: (1) in the training phase, predictors learn and record weights for impact factors with the help of training sets, and (2) in the prediction phase, predictors predict ranking scores for warnings according to the given values of impact factors. An effective training set, which comprises a set of input attributes and an output attribute for each warning, is vital in exploring how each input attribute affects the output attribute for a warning.

In this paper, we use a machine-learning workbench named "Waikato Environment for Knowledge Analysis" (Weka) [31] to help carry out our training process. Weka is a popular suite of machine learning software written in Java. It supports standard data mining tasks such as data preprocessing, clustering, classification, regression, visualization, and feature selection. It also provides implementations for various machine learning algorithms (e.g., Bayesian Network, Logistic Regression, Bootstrap Aggregating, Random Tree, K-nearest Neighbors, and Decision Table).

3. APPROACH OVERVIEW

We extract a training set with warnings reported by static analysis tools, label it with the help of "generic-bug-related lines", and use it to train a predictor that can be further used to prioritize static analysis warnings. Figure 1 shows the overview of our approach.

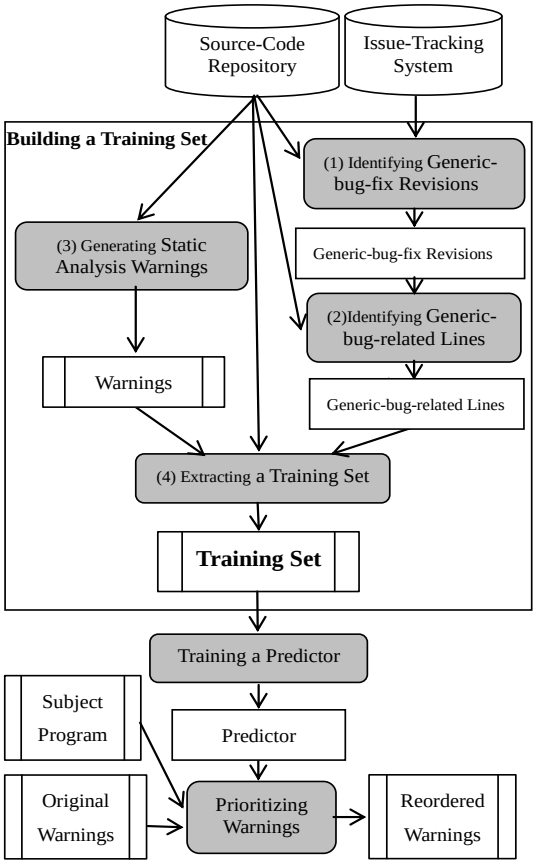

Figure 1. Approach Overview

Table 1. Projects under analysis

Project Name	Number of Revisions	Development Period	Community	Source-code Repository	Issue-tracking System
Lucene	2630	2001.9-2007.8	Apache	SVN	JIRA
Spring	3696	2007.4-2009.4	Apache	SVN	JIRA
ANT	746	1998.10-2003.5	Sable.MCGill	SVN	BUGZILLA
Log4j	120	2007.8-2009.2	Apache	SVN	BUGZILLA
JPF	1212	2002.7-2009.4	ObjectWeb	SVN	OW2-Gforge

3.1 Preparing Historical Data

Most open-source projects publish their source-code repositories and issue-tracking systems over the Internet. The source-code repository of a project is responsible for recording all change histories of its source code [32], and its issue-tracking system maintains all detailed information of its issues (e.g., bug issues). We first obtain source-code repositories and issue-tracking databases of five open-source projects under analysis before building our training set. Table 1 describes these projects.

3.2 Building a Training Set

Our approach includes four main steps in building a training set:

(1) **Identifying Generic-bug-fix Revisions.** Generic-bug-fix revisions are those revisions submitted to source-code repositories with the purpose of fixing generic bugs, only which most static analysis tools can detect. These revisions provide hints to locate generic-bug-related lines. Section 4.1 shows more details.

(2) **Identifying Generic-bug-related Lines.** Generic-bug-related lines are lines modified or removed by generic-bug-fix revisions. These lines are helpful in labeling our training set. Section 4.2 shows more details.

(3) **Generating Static Analysis Warnings.** We generate warnings by running static analysis tools against different revisions of different projects under analysis.

(4) **Extracting a Training Set.** We select three categories of impact factors as input attributes of our training set (i.e., warning descriptors, statistics for warnings of different tools, and features of the warning-related source code) and extract them from warnings and the warning-related source code. We label the output attribute of each warning in the training set with the help of "generic-bug-related lines". Section 4.4 shows more details.

3.3 Training a Predictor to Prioritize Warnings

Weka provides implementations of various machine-learning algorithms. We select the implementations of six well-know algorithms in Weka to train predictors: Bayesian Network, Logistic Regression, K-nearest Neighbors, Bootstrap Aggregating, Random Tree, and Decision Table.

We use default setting values of each algorithm during the training process and use "10-folds cross validation" during the validation process. To explore the most suitable machine-learning algorithm for our problem, we use the six selected algorithms to train different predictors against the same training set extracted from the five projects under analysis. The validation results show that the "K-nearest Neighbors" based predictor achieves the best Precision (98.7%), Recall (98.7%), and F-Measure (98.7%). Therefore, we select the "K-nearest Neighbors" as the best machine-learning algorithm for our problem and use it as the default algorithm in our evaluation (Section 5).

After training the predictor with our training set, we use the predictor to prioritize static analysis warnings: we first use the predictor to predict the actionability value for each warning (the probability value that the warning is actionable), and then prioritize all warnings according to their actionability values.

4. TRAINING-SET CONSTRUCTION

In this paper, we build a training set for warning prioritization by automatically determining whether static analysis warnings are actionable (i.e., accurate and worth fixing for developers). A warning reported by static analysis tools may have multiple manifestations or multiple possible fixes. However, a good static analysis tool should be able to indicate at least one of them to a developer, and ideally should indicate lines that the developer can choose to fix. Because most warnings provided by static analysis tools are related to generic bugs, we use generic-bug-related lines, which are modified or deleted by developers in generic-bug-fix revisions, to evaluate static analysis warnings. In order to identify generic-bug-related lines, we identify generic-bug-fix revisions first since the changes made in these generic-bug-fix revisions are related to generic bugs.

4.1 Identifying Generic-bug-fix Revisions

Open-source projects usually include thousands of revisions in their source-code repositories. We classify these revisions into three coarse-grained categories: bug-fix revisions, non-fix revisions, and multi-purpose revisions. "Bug-fix revisions" are those revisions aiming at fixing bugs. According to the type of their fixed bugs, fix revisions can be further divided into two groups: generic-bug-fix revisions and project-specific-bug-fix revisions. "Non-fix revisions" do not involve fix activities but involve other activities such as "new feature addition" and "code refactoring". "Multi-purpose revisions" are those revisions that involve not only fix activities but also non-fix activities.

Open-source projects are commonly co-developed by developers distributed all over the world. To facilitate their cooperation, strong guidelines for writing the log message of each revision are undertaken. Chen et al. [8] studied the quality of open source change log, and found that almost all log messages are consistent with their corresponding submitted changes. Therefore, it is reasonable to identify generic-bug-fix revisions with the help of log messages. In previous work, two techniques were proposed for identifying bug-fix revisions based on analyzing log messages: identification based on "bug issue key" references [7, 11, 30] and identification based on searching "bug-fix-related keywords" [18, 23]. These two techniques achieved acceptable precision and recall in identifying bug-fix revisions. However, through our evaluation, we find that about 95% of bug-fix revisions identified by the first technique for Lucene are project-specific-bug-fix ones, and about 90% by the second technique are project-specific-bug-fix revisions. Therefore, we cannot directly use these two techniques to identify generic-bug-fix revisions.

In this paper, we propose an automatic technique for identifying generic-bug-fix revisions. Figure 2 presents the pseudo code of our algorithm for identifying generic-bug-fix revisions.

The algorithm performs one-time identification process for each revision ***r***. First, it computes the number ***n*** of Java files modified by revision ***r.*** By bounding the value of ***n*** (Line 3), most multi-purpose revisions and non-fix revisions are filtered out, and most bug-fix revisions are kept. The rationale of this step lies in that, through our investigation, we find that usually generic bug-fix revisions modify only a few files whereas multi-purpose revisions and most non-fix revisions usually modify a lot of files. By bounding the number of modified Java files, we first remove most multi-purpose revisions and non-fix revisions (keeping most generic-bug-fix revisions).

Then Line 5 determines whether the log message l of revision r contains a bug issue key. A common phenomenon exists for open-source projects: when committing bug-fix revisions, developers tend to include only "the bug issue key" and avoid re-describing the bug issue in the log message since there is already a copy of detailed description for the bug in the issue-tracking system (see Figure 3). Therefore, if the log message contains a bug issue key, the corresponding issue description ***issueDes*** is first retrieved from the project's issue-tracking system (Line 6) and then used to conduct the following identification process because the issue description is more detailed than the log message in this case.

Lines 7-10 are responsible for computing the maximal similarity ***maxSimilarity*** between ***issueDes*** (the description of the bug issue) and each ***bugDes*** (the description of each generic bug that static analysis tools can detect). In this process, based on the theory of Vector Space Model (VSM) [27], the similarity between the ***issueDes*** and each ***bugDes*** is measured by computing the cosine value of the angle between their corresponding vectors. When ***maxSimilarity*** is large enough (Line 11), the revision ***r*** is identified as a generic-bug-fix revision.

If the log message does not contain a bug issue key, the log message is used directly to compute the maximal similarity ***maxSimilarity*** with each ***bugDes*** (Lines 14-17). When ***maxSimilarity*** is large enough (Line 18), the revision ***r*** is also identified as a generic-bug-fix revision.

```
Algorithm identifyGenericBugFixRevisions
Begin
1. foreach revision r do
2.   compute n (the number of Java files modified in r )
3.   if (n <= A) then
4.     maxSimilarity = 0;
5.     if (the log message l of r contains a bug issue key) then
6.       extract issueDes (description of the bug issue)
7.       foreach bugDes (description of each generic bug) do
8.         compute similarity s between issueDes and bugDes
9.         if (s > maxSimilarity) then
10.          maxSimilarity = s;
11.      if (maxSimilarity > B) then
12.        identify r as a generic-bug-fix revision
13.    else
14.      foreach bugDes do
15.        compute similarity s between l and bugDes
16.        if (s > maxSimilarity) then
17.          maxSimilarity = s;
18.      if (maxSimilarity > C) then
19.        identify r as a generic-bug-fix revision
End
```

Figure 2. The algorithm for identifying generic-bug-fix revisions

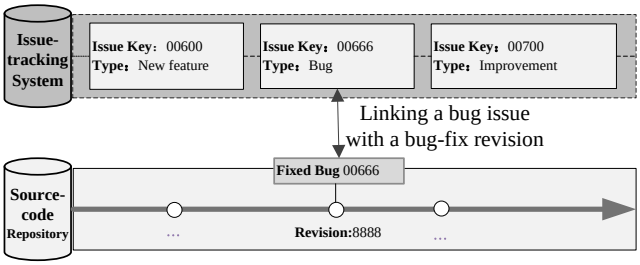

Figure 3. Associating a bug-fix revision with a bug issue by unique "issue key"

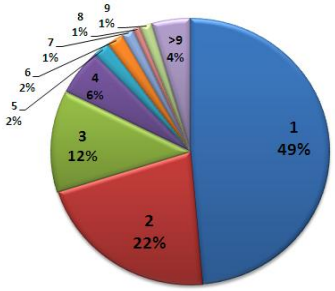

Figure 4. The pie chart of generic-bug-fix revisions divided by the number of their modified Java files

In our algorithm, three threshold values represented as ***A***, ***B***, and ***C*** should be determined empirically. Before determining the values, we first identify generic-bug-fix revisions, manually and

randomly, from all revisions of the five subject projects (including Lucene, ANT, Spring, Log4j, and JPF). As a result, 358 revisions are identified as generic-bug-fix ones.

Determination of *A*. In order to identify the value of ***A*** (the threshold of the number of the modified Java files for most generic-bug-fix revisions), we conduct a statistical analysis on the number of their modified Java files for the 358 generic-bug-fix revisions. Figure 4 shows the pie chart of these 358 generic-bug-fix revisions divided by the number of their modified Java files. We can find that 49%, 22%, 12%, and 6% of them modify only 1, 2, 3, and 4 Java files, respectively. In total, most generic-bug-fix revisions (about 88%) modify no more than 4 Java files. In addition, we also find that most multi-purpose revisions (about 97%) modify more than 4 files. According to this statistical analysis result, we set A as 4.

Determination of *B*. In order to determine the value of ***B*** (the threshold of the maximal similarity between the description of a bug issue and the description of each generic bug that static analysis tools can detect), we execute the algorithm against only those revisions whose log messages contain bug issue keys (a subset of the 358 generic-bug-fix revisions) because the identification process for only those revisions is affected by ***B***. We fix ***A*** as 4 and ***C*** as 0 (***C*** can be set as an arbitrary value since it does not affect the identification process for those revisions). We set ***B*** as different values from 0 to 0.9 (increased with 0.1) and, for each value, evaluate the algorithm's precision, recall, and F-measure value, respectively. Figure 5 shows the result. From the result, we determine that the F1 value of the algorithm is the best when B is set as 0.5.

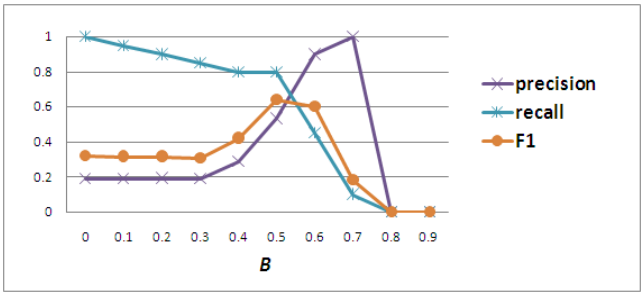

Figure 5. The precision, recall, and F1 of the algorhim against revisions whose log messages contain bug issue keys

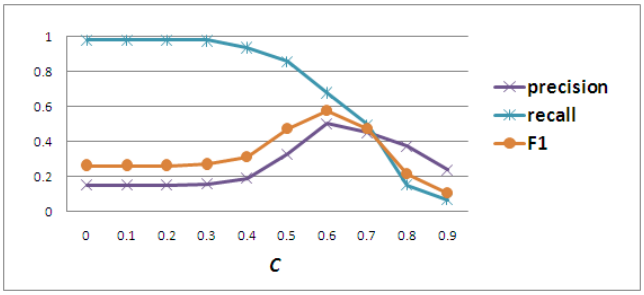

Figure 6. The precision, recall, and F1 of the algorhim against revisions whose log messages contain no bug issue key

Determination of *C*. In order to determine the value of ***C*** (the threshold of the maximal similarity between a log message and the description of each generic bug that static analysis tools can detect), we execute the algorithm against only those revisions whose log messages contain no bug issue key because the identification process for only those revisions are affected by ***C***. We fix ***A*** as 4 and ***B*** as 0.5. We set ***C*** as different values from 0 to 0.9 (increased with 0.1) and, for each value, evaluate the algorithm's precision, recall and F1 value, respectively. Figure 6 shows the result. From the result, we determine that the F1 value of the algorithm is the best when ***C*** is set as 0.6.

Our automatic identification algorithm can reduce large manual efforts for developers in identifying generic-bug-fix revisions but its precision (about 60%) is not high enough so far. After the automatic identification process, we conduct manual verification on its result to remove false positives (e.g., spending about 0.7 man hours on 635 automatically identified generic-bug-fix revisions).

4.2 Identifying Generic-bug-related Lines

After generic-bug-fix revisions are identified, the next step is to identify which lines of a specific revision have been deleted or modified during generic-bug-fix changes. A code line ***l*** is a generic-bug-related line if and only if ***l*** is modified or removed during any generic-bug-fix change.

Function identifyGenericBugRelatedLinesForASpecificRevision
Input ***X*** : the number of the revision whose generic-bug-related lines are to be computed
Output $brls_X$: the generic-bug-related lines of the revision ***X***
Begin
1. **foreach** generic bug-fix revision ***N*** **do**
2. **if** (***N*** > ***X***) **then**
3. get the diff info $diff_{(N\text{-}1\text{^}N)}$ between revision ***N*** and ***N-1***;
4. parse $diff_{(N\text{-}1\text{^}N)}$ to identify $brls_{(N\text{-}1\text{^}N)}$ (generic-bug-related lines of revision ***N-1*** modified to result in revision ***N***);
5. **foreach** generic-bug-related line ***brl*** in $brls_{(N\text{-}1\text{^}N)}$ **do**
6. retrieve the introducing reversion number ***Y*** of ***brl***;
7. **if** (***Y*** <= ***X***) **then**
8. identify the corresponding line brl_X of ***brl*** in revision ***X***;
9. add brl_X to $brls_X$;
End

Figure 7. The identification algorithm of generic-bug-related lines for a revision *X*

Figure 7 presents our algorithm for identifying generic-bug-related lines. It takes as input ***X***, the number of the revision whose generic-bug-related lines need to be computed. The algorithm returns $brls_X$, the generic-bug-related lines of the revision ***X***, as output.

The algorithm uses each generic-bug-fix revision whose revision number is larger than ***X*** to compute generic-bug-related lines for revision ***X***. When revision ***N*** is a generic-bug-fix revision and its revision number is larger than ***X*** (Lines 1-2), the diff information $diff_{(N\text{-}1\text{^}N)}$ between revision ***N*** and revision ***N-1*** is obtained first (Line 3). Then the generic-bug-related lines $brls_{(N\text{-}1\text{^}N)}$ of revision ***N-1*** modified to result in revision ***N*** are identified by parsing $diff_{(N\text{-}1\text{^}N)}$ (Line 4). For each generic-bug-related line of $brls_{(N\text{-}1\text{^}N)}$, its introducing revision number is retrieved from the project's source-code repository (Line 6) and then compared with ***X*** (Line 7). If the introducing revision number is smaller than or equal with ***X***, it means that this line was introduced no later than revision ***X***, and thus an identical generic-bug-related line must exist in revision ***X***. Then the identical line in revision ***X*** is located and marked as a generic-bug-related line of revision ***X*** (Lines 8-9).

Example. Figure 8 illustrates an example on identifying generic-bug-related lines for revision 4. All generic-bug-fix revisions whose numbers are larger than 4 are listed in Figure 8:

revision 6 and revision 7. Revision 7 modified the lines of the top two gray blocks of revision 6. Therefore, these lines are identified as generic-bug-related lines of revision 6. The introducing revision number of the lines in the uppermost gray block of revision 6 is 3, which is smaller than 4, indicating that these lines must also exist in revision 4. Therefore, their corresponding lines in revision 4 are located and marked as generic-bug-related lines of revision 4. However, the introducing revision number of the line in the middle gray block of revision 6 is 5, larger than 4, indicating that these lines do not exist in revision 4 and the analysis for these lines terminates. After that, the next generic-bug-fix revision (revision 6) is analyzed with the same process. After analyzing all the generic-bug-fix revisions, we finish identifying the generic-bug-related lines of revision 4 as three lines.

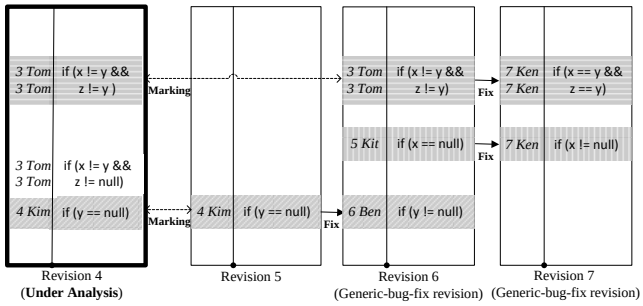

Figure 8. Identifying generic-bug-related lines for revision 4

4.3 Generating Static Analysis Warnings

Since different tools are complementary with each other to some degree [9] [32], we extract a training set with warnings from multiple tools. We generate warnings of four tools (FindBugs, PMD, Jlint, and Lint4J) by running CODAS, which integrates the four tools tightly, against the selected revisions of the projects under analysis.

4.4 Extracting a Training Set

A training set, which is used for training a predictor, is a set of effective warning examples (actionable warnings and non-actionable warnings). An effective training set should be accurate, representative, fair, and abundant [16, 28].

Our training set is extracted based on warnings, analyzed source code, and generic-bug-related lines. Table 2 lists 22 input attributes (impact factors) of our training set. These factors can be divided into three categories:

(1) Warning descriptors. A given warning's descriptors (e.g., the pattern name and tool name) are taken directly from the warning report.

(2) Statistics for warnings from different tools. These factors include the number of warnings that are reported for the same warning location, the same file, and the same project by each tool: FindBugs, PMD, Jlint, and Lint4j. The rationale behind these factors is that if there are more tools reporting more warnings for the same piece of code, the code is more fault-prone and the warnings generated from the code are more accurate and actionable.

(3) Source code features. We consider eight factors that may provide insight into the warning-related code's features. The first one is the depth of the warning-related code in the file, indicating that how far down (%) this warning is in the file, in terms of the percentage of the lines of the file. Some other factors are code length, comment length, and comment-code ratio (where the length denotes the number of lines). The rationale behind these factors is that more comments may reflect higher maintainability and quality of the code. The fifth factor considers how far down (%) this warning is in the method of the warning-related code (named warning-related method), in terms of the percentage of the lines of the warning-related method. The length of the warning-related method is considered as the sixth factor. The last two factors are the number of "callers" and "callees" of the warning-related method. The rationale behind these two factors is that a method with more callers is more likely to be the kernel part of the project, which is often tested more sufficiently and is less fault-prone; a method with more callees tends to be of higher complexity and more fault-prone.

Table 2. Input attributes of our training set

Input attributes	Description
Warning Descriptors	
Pattern Name	Bug pattern name of the warning
Tool Name	Name of the tool reporting the warning
Statistics for Warnings of Each Tool(4*3 attributes)	
Location Warnings	Number of warnings reported for the same location by each tool
File Warnings	Number of warnings reported for the same file by each tool
Project Warnings	Number of warnings reported for the same project by each tool
Source Code Features	
File Depth	How far down (%) in the file this warning is
Code Length	Number of lines of code of the warning-related file
Comment Length	Number of lines of comments of the warning-related file
Comment-Code Ratio	The ratio of comment length and code length of the warning-related file
Method Depth	How far down (%) in the method this warning is
Method Length	Number of lines of the warning-related method
Method Callers	Number of callers of the warning-related method
Method Callees	Number of callees of the warning-related method

We extract the impact factors for warnings by collecting statistics for the warnings reported by each tool and conducting simple static analysis for the warning-related source code.

The output attribute of a warning in the training set is "isActionable", which indicates whether the warning is accurate and actionable. If the location of a warning contains some generic-bug-related line(s), we can assume that this warning is very likely accurate and actionable for developers. However, it is often the case that some code fragment is related to several different warnings. When the fragment is revised to fix some certain warning (i.e., lines of the fragment are marked as generic-bug-related lines), a problem arises: other warnings could be wrongly identified as actionable ones. To address the problem, we use a new heuristic: if a warning disappears in a later revision and also reports at least one generic-bug-related line, the warning is treated as actionable and its output attribute is labeled as "isActionable"; otherwise, it is non-actionable. The rationale of this heuristic lies in that if a warning disappears in a later revision without reporting any generic-bug-related line, we cannot regard it actionable because it may disappear due to some code changes introduced by code refactoring or new features addition; if a warning reports some generic-bug-related line but still exists at later revisions, we cannot label it as actionable because the generic-bug-related line

may be revised by developers aiming at fixing its other associated warnings.

After constructing the training set, the first three authors of this paper conduct manual verification on it with the principle of "majority voting": when there is no consensus during verification, the opinion of majority people is adopted. Manual verification shows that about 95% of actionable warnings in the training set are confirmed to be actionable.

Before training a predictor, we apply seven factor-selection algorithms against our generated training set. The final statistical analysis shows that there are four factors (input attributes) ranked last with their effectiveness way behind the other factors by all of the seven algorithms: **"method callers", "tool name", "location warnings of Jlint", and "location warnings of Lint4j"**. Such analysis result indicates that these four impact factors have small correlation coefficients with the output attribute. Therefore, we ignore these four impact factors when training a predictor.

5. EVALUATION

In this section, we present the evaluation for the effectiveness of our proposed approach of constructing a training set with different strategies. Because a training set is used to train a predictor, which is finally used to prioritize warnings, we can explore the best strategies of constructing a training set by evaluating the prioritized warnings. Given a certain training set, we train a machine-learning-based predictor, collect original warnings from multiple tools (FindBugs, PMD, Jlint, and Lint4J), apply the predictor to predict actionability information (ranking score) for each warning, and then reorder these warnings based on their actionability.

To evaluate the quality of reordered warnings, we use the precision of top warnings as the criterion. Since developers are unlikely to scrutinize all reported warnings due to their limited time, the top warnings' quality is of considerable significance. We use "precision" to evaluate top warnings:

Top n Warnings' Precision: $P(n) = N_{actionable}/n$

$N_{actionable}$ represents the number of actionable warnings among the top n warnings. An actionable warning should be accurate and also acted on during some generic-bug-fix procedure.

In our evaluation, we address the following research questions (**RQ**):

RQ1: How effective is our training set for warning prioritization compared with previous work?

RQ2: Should we construct a training set from multiple tools' warnings instead of a single tool's warnings for a given project under analysis?

RQ3: Should we construct a training set from multiple projects' history instead of a single project's history for a given project under analysis?

5.1 RQ1: Effectiveness Evaluation

We evaluated the performance of our prioritization compared with Kim and Ernst's [18]. In their evaluation, they prioritized warnings of three tools (FindBugs, PMD, and Jlint) for the revision submitted in August 30, 2004 of Lucene. We also applied our approach to prioritize warnings of the three tools for the same revision.

In our approach, we first constructed a training set from warnings reported by three tools (FindBugs, PMD, and Jlint) for five projects (Lucene, Spring, JPF, Log4j, and ANT) from revisions 1 to m/2-1 (m is the maximum revision number for each project: 2630 revisions for Lucene, 3696 for Spring, 746 for ANT, 120 for Log4j, and 1212 for JPF). With the training set, we trained a "K-nearest Neighbors" based predictor and then used the predictor to prioritize warnings for that revision submitted in August 30, 2004 (which was among revisions m/2 to m).

Figure 9 shows the precision of our prioritization, Kim's prioritization, and the built-in prioritization of each tool for Lucene. In Figure 9 and the remaining figures in this paper, the X axis represents the top n warnings in the warning list, and the Y axis represents the precision of the top n warnings correspondingly.

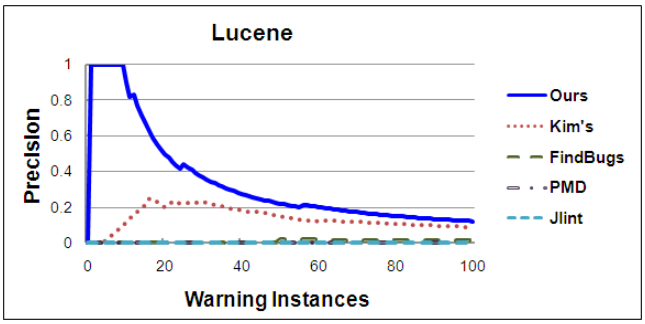

Figure 9. Precision of top n (up to 100) warnings prioritized by our prioritization and Kim's prioritization

From Figure 9, we can observe that, for Kim's prioritization, the best precision for a subset of the warnings is only 25% (top 16) for Lucene; for our prioritization, the precision of top 9 warnings is always 100%. Moreover, our prioritization can keep the precision of top 100 warnings always higher than tools' default prioritization. In summary, the result shows that our prioritization has a remarkable improvement over previous work, and our training set is effective for warning prioritization.

5.2 RQ2: Training-set construction using single tool vs. multiple tools

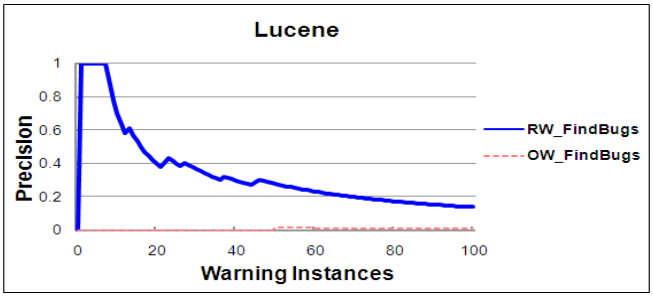

Figure 10. Prioritizing warnings based on training set constructed from warnings of the single tool FindBugs

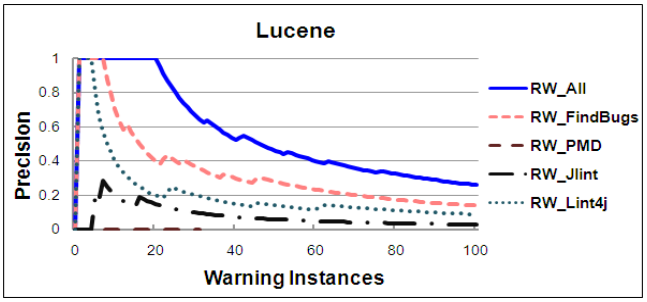

Figure 11. Prioritizing warnings based on training sets constructed using warnings of multiple tools vs. single tool

In this section, we made a comparison between a training set constructed using one single tool and that using multiple tools.

For both strategies, we attained warnings for revisions (from revisions 1 to m/2-1) of all the five projects under analysis (the next section shows that using multiple projects can achieve better effectiveness than using one specific project in most cases).

Based on the constructed training set, we trained predictors to reorder warnings for revision m/2 of the projects under analysis. Since the results of all projects are similar, we show only the result of Lucene here due to space limit. Figure 10 shows the effectiveness of our prioritization when using one single tool (Findbugs here). In Figure 10, RW_FindBugs denotes the reordered warnings by our prioritization based on the training set built using FindBugs, and OW_FindBugs denotes the original warnings of FindBugs. The figure shows that our approach improved top warnings' precision compared with FindBugs' default prioritization.

Figure 11 shows results of our warning prioritization based on a training set constructed using warnings of four tools (FindBugs, PMD, Jlint, and Lint4j), where RW_All, RW_FindBugs, RW_PMD, RW_Jlint, and RW_Lint4j denote the reordered warnings by our approach based on the training sets extracted from all tools, FindBugs, PMD, Jlint, and Lint4j, respectively. The result shows that constructing a training set using warnings of more tools can help warning prioritization achieve better effectiveness.

5.3 RQ3: Training-set construction using single project vs. multiple projects

Given a program **a** under analysis, which belongs to a project A under analysis, we have three strategies to construct a training set for it:

(1) using only historical data of project **A**;

(2) using not only historical data of project **A**, but also historical data from other projects **B**, **C**, …;

(3) using only historical data of other projects. This strategy can be used in two situations: (i) the project **A** is in an initial stage and thus lacks historical data; (ii) the program **a** is sent to a third-party analysis service, such as CODAS, where it is usually difficult to attain **a**'s historical data.

We denote the three strategies as **A-a**, **ABC-a**, and **ABC-d** (with d denoting different projects), respectively. We show the comparison results for these three strategies in the subsequent subsections. Note that we constructed the training sets for our prioritization using all the four tools and conducted the comparison of warning prioritization for revision m/2 of each program ***a*** under analysis.

1.1.1 A-a vs. ABC-a

For each project under analysis, we constructed different training sets using two strategies: "A-a" (using its own history) and "ABC-a" (using five projects' historical data: Lucene, Spring, JPF, Log4j, and ANT). Based on different training sets, we trained different predictors to prioritize warnings.

The results of all projects are similar, but due to space limit we show only part of them. Figures 12-14 show the results for Lucene, ANT, and Spring, respectively. In Figures 12-14, Ours(ABC-a) and Ours(A-a) denote our prioritization based on the strategies "ABC-a" and "A-a", respectively. From Figures 12-14, we have the following observations:

(1) **Both the "ABC-a" and "A-a" based prioritization outperform each tool's built-in prioritization effectively.** For example, for Lucene, the precision of top 100 warnings of our approach is always more than 26%. In contrast, the precision of each tool is always less than 2%. Only one warning among top 100 warnings of FindBugs is actionable, whereas 26 are actionable among top 100 warnings of our approach and 20 of them are prioritized on the top of the warning list by our "ABC-a" based prioritization. For ANT, only 3 of top 100 warnings of FindBugs are actionable, while 22 and 20 warnings are actionable among top 100 warnings of ours "ABC-a" and "A-a" based prioritization, respectively.

(2) **The "ABC-a" based prioritization is superior to the "A-a" based prioritization for the three projects.** Through analysis, we find that the results shown in Figures 12-14 are reasonable. Our projects under analysis are all open-source and their developers generally share the same guidelines or customs to fix bugs. Only one project's history is limited and cannot supply enough "knowledge" for a predictor to learn. A larger training set from more projects can help a predictor learn more. Therefore, it is reasonable that the "ABC-a" based prioritization performs better than the "A-a" based prioritization due to the sufficiency of training data.

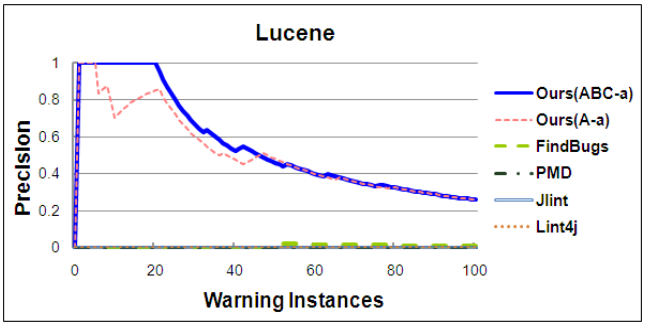

Figure 12. "A-a" vs. "ABC-a" based prioritization for Lucene

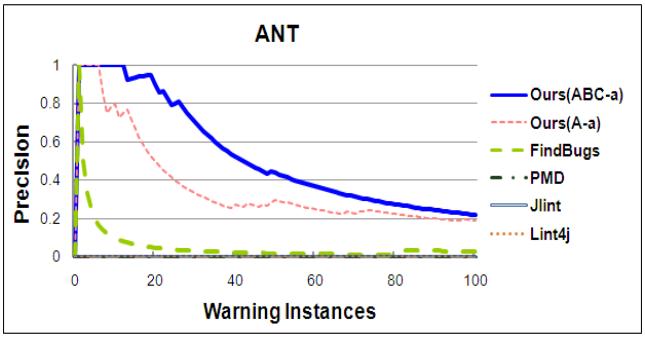

Figure 13. "A-a" vs. "ABC-a" based prioritization for ANT

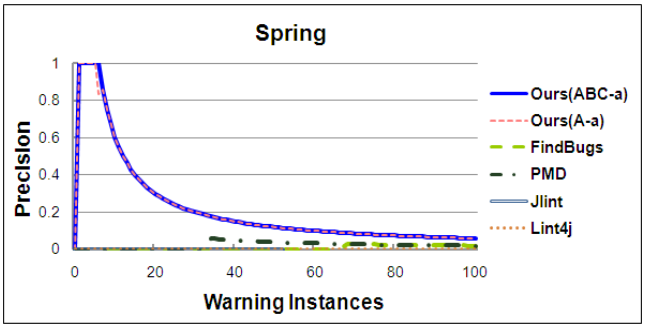

Figure 14. "A-a" vs. "ABC-a" based prioritization for Spring

1.1.2 ABC-a vs. ABC-d

We used the same training set (constructed using history from Lucene, Spring, JPF, Log4j, and ANT) to prioritize warnings for Lucene and Tapestry. Therefore, for Lucene, the prioritization is

based on the "ABC-a" strategy, whereas for Tapestry, the prioritization is based on the "ABC-d" strategy. Figures 15-16 show the results. From Figures 15-16, we observe that (1) both the "ABC-a" based prioritization and the "ABC-d" based prioritization outperform each tool's built-in prioritization effectively; (2) the "ABC-a" based prioritization performs better than the "ABC-d" based prioritization.

In summary, the "ABC-a" strategy can help construct a more effective training set than the "ABC-d" strategy.

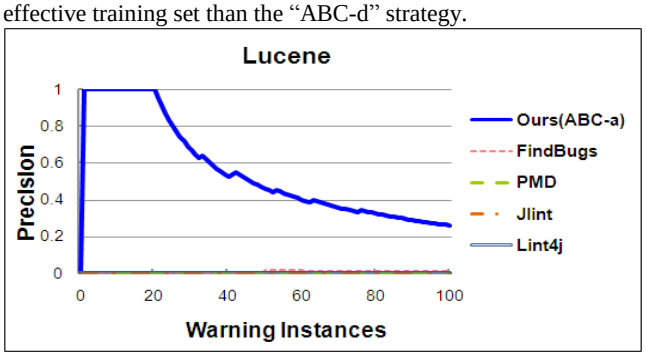

Figure 15. "ABC-a" based prioritization for Lucene

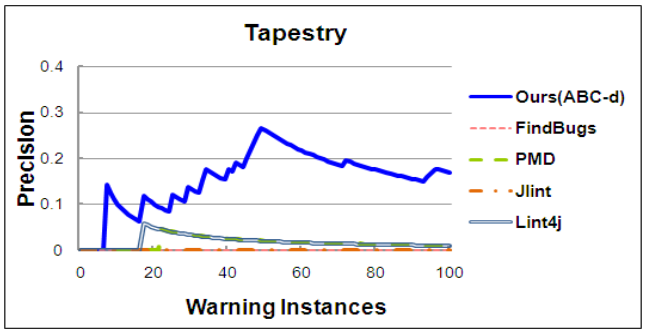

Figure 16. "ABC-d" based prioritization for Tapestry

5.4 Threats to validity

The main threat to external validity includes the representativeness of the subject projects that we selected in the evaluation. Therefore, the results of our evaluation may be specific only to these projects. To reduce this threat, we chose projects from different open source communities and their types are different from each another. The threat could also be reduced by more evaluations on more subjects in future work. The main threat to internal validity is human factors when manually verifying the generic-bug-fix revisions automatically identified by our approach and the actionable warnings in our constructed training set. To reduce the threat, the first three authors of this paper conducted the verifications with the principle of "majority voting" and great carefulness.

6. RELATED WORK

To the best of our knowledge, we are the first to propose an automatic approach to construct an effective training set for warning prioritization and point out the importance of "generic-bug-related lines" for the problem. There are several threads of related work; however, they use different techniques to prioritize warnings.

Kremenek and Engler [21] developed a ranking algorithm (z-ranking) to prioritize warnings for a single warning type (i.e., a single "checker"). They observed that clusters of warnings were usually either all false positives or all true bugs. A generalization of this work produced an adaptive ranking scheme called Feedback Rank [20]. This work used code locality to identify clusters of false positives and true bugs among warnings. Their algorithm updated warning priorities as nearby warnings were classified. In their approach, based on their observation, they selected "code locality" (an impact factor) to directly prioritize warnings. However, their observation is not generic for all warnings.

Boogerd and Moonen [3] used execution likelihood analysis to prioritize warnings. For each warning location, they computed the execution likelihood (an impact factor). If the location was very likely to be executed, the warning was assigned a high priority. This technique might help developers to focus on warnings at locations that had high execution likelihood. However, warnings at locations with low execution likelihood could be also important. In fact, severe bugs in lines with low execution likelihood are more difficult to detect. Heckman [17] proposed an adaptive model that used feedback from developers to rank warnings. The developers' feedback resulted in updating weights for warnings. Her approach was effective in favorably ranking and identifying false-positive warnings. Her approach tried to dynamically collect "bug-fix information" (a training set) from developers' feedback to improve warning prioritization. However, developers are usually too busy to provide feedback when using static analysis tools.

Williams and Hollingsworth [32] used software change histories to improve existing static analysis tools. 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 static analysis tool warns all locations that use unchecked return values. To reduce the false positives, they used the software history to find which kinds of function return values must be checked. However, they focused on only a small set of bug patterns, while our approach is general to all warnings. Ruthruff et al. [25] used logistic regression models to predict the categories of warnings from information in the warnings and the related code. Their work was aimed to classify warnings rather than ranking all warnings. The training data for their model was manually labeled by Google's developers.

Kim and Ernst [18, 19] estimated the importance of warning categories by mining bug-fix information. They used all bug-related lines to identify whether a warning is a true positive or not. In our evaluation, we found that more than 60% of bug-related lines computed by their approach were about project-specific bugs (which static analysis tools cannot detect or report). Those project-specific-bug-related lines should be eliminated when evaluating static analysis warnings; otherwise high inaccuracy could be introduced. In our work, we pointed out the importance of "generic-bug-related lines" for evaluating static analysis warnings, proposed an automatic approach to compute them, and used them to construct an effective training set for warning prioritization.

7. CONCLUSION

In this paper, we highlighted the importance of an effective training set for warning prioritization and proposed an automatic approach to construct an effective training set by mining generic-bug-fix history. In our training set, we summarized a set of impact factors for warning prioritization as its input attributes and labeled it with the help of "generic-bug-related lines". We pointed out the importance of "generic-bug-related lines" and proposed a new technique based on natural language processing (NLP) to assist in identifying "generic-bug-fix revisions" effectively. We also proposed a new algorithm to effectively identify accurate "generic-bug-related lines" and conducted evaluations for our approach.

Although our training set is effective, there exist some limitations. First, it is possible that some important bugs never got noticed or fixed, even when a tool would report them. Second, the

warning locations and fix locations for a bug may have no overlap with each other. Sometimes 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. Third, our current identification technique for generic-bug-fix revisions has not achieved sufficiently high accuracy. Some human intervention is still needed during this phase. Forth, we do not conduct code-movement detection when computing bug-related lines. If some lines are moved from the original location to a new location, the source-code repository records only that the moved lines are deleted at the original location and some "new" lines are added at the new location. Without code-movement detection, there could be some lines falsely determined as bug-related lines. In our future work, we plan to explore better heuristics for identifying actionable warnings, and more automatic and accurate techniques to identify generic-bug-fix information.

8. ACKNOWLEDGMENT

The authors from Peking University are sponsored by the National Basic Research Program of China (973) grant 2009CB320703 and the Science Fund for Creative Research Groups of China grant 60821003, and the National Science Foundation of China grant 60773160. Tao Xie's work is supported in part by NSF grants CCF-0725190, CCF-0845272, CNS-0958235, ARO grant W911NF-08-1-0443, and ARO grant W911NF-08-1-0105 managed by NCSU SOSI.

9. REFERENCES

[1] C. Artho. Jlint - Find Bugs in Java Programs. http://Jlint.sourceforge.net/.

[2] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, vol. 25, no. 5, pages 22-29, 2008.

[3] C. Boogerd and L. Moonen. Prioritizing software inspection results using static profiling. In *Proc. SCAM*, pages 149-160, 2006.

[4] D. Binkley. Source code analysis: a road map. In *Proc. FOSE*, pages 104-119, 2007.

[5] J. Bevan, E. J. Whitehead, Jr., S. Kim, and M. Godfrey. Identifying changed source code lines from revision repositories. In *Proc. ESEC/FSE*, pages 177-186, 2005.

[6] B. Chess and J. West. Secure programming with static analysis. *Aaison Wesley*, 2007.

[7] D. Cubranic and G. C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proc. ICSE*, pages 408-418, 2003.

[8] K. Chen, S. R. Schach, L. Yu, J. Offutt, and G. Z. Heller. Open-source change logs. *Empirical Software Engineering*, vol. 9, no. 3, pages 197-210, 2004.

[9] D. Engler, B. Chelf, A. Chou, and S. Hallem. Bugs as deviate behavior: A general approach to inferring errors in system code. In *Proc. SOSP*, pages 57-72, 2001.

[10] D. Engler and M. Musuvathi. Static analysis versus software model checking for bug finding. In *Proc. VMCAI*, pages 191-210, 2004.

[11] M. Fischer, M. Pinzger, and H. Gall. Populating a release history database from revision control and bug tracking systems. In *Proc. ICSM*, pages 23-32, 2003.

[12] FindBugs, available at *http://findbugs.sourceforge.net/*.

[13] Fortify, available at *http://www.fortify.net/intro.html*.

[14] K. Hornik, M. Stinchcombe and H. White. Multilayer feedforward networks are universal approximators. *Neural Networks*, vol. 2, pages 359-366, 1989.

[15] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proc. OOPSLA*, pages 132–136, 2004.

[16] S. Heckman and L. Williams. On establishing a benchmark for evaluating static analysis alert prioritization and classification techniques. In *Pro. ESEM*, pages 41-50, 2008.

[17] S. S. Heckman. Adaptively ranking alerts generated from automated static analysis. *ACM Crossroads*, 14(1), pages 1-11, 2007.

[18] S. Kim and M. D. Ernst. Which warnings should I fix first? In *Proc. ESEC/FSE*, pages 45-54, 2007.

[19] S. Kim and M. D. Ernst. Prioritizing warning categories by analyzing software history. In *Proc. MSR*, pages 27-30, 2007.

[20] T. Kremenek, K. Ashcraft, J. Yang and D. Engler. Correlation exploitation in error ranking. In *Proc. FSE*, pages 83-93, 2004.

[21] T. Kremenek and D. R. Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. In *Proc. SAS*, pages 295-315, 2003.

[22] Lint4j, available at *http://www.jutils.com/*.

[23] A. Mockus and L. G. Votta. Identifying reasons for software changes using historic databases. In *Proc. ICSM*, pages 120-130, 2000.

[24] PMD, available at *http://pmd.sourceforge.net/*.

[25] J. R. Ruthruff, J. Penix, J. D. Morgenthaler, S. Elbaum, and G. Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *Proc. ICSE*, pages 341-350, 2008.

[26] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for Java. In *Proc. ISSRE*, pages 245-256, 2004.

[27] G. Salton, A. Wong, and C. S. Yang. A vector space model for automatic indexing. *Communications of the ACM*, vol.18, no.11, pages 613-620, 1975.

[28] S. E. Sim, S. Easterbrook, and R. C. Holt. Using benchmarking to advance research: a challenge to software engineering, In *Proc. ICSE*, pages 74-83, 2003.

[29] J. Spacco, D. Hovemeyer, and W. Pugh. Tracking defect warnings across revisions. In *Proc. MSR*, pages 133-136, 2006.

[30] J. Sliwerski, T. Zimmermann and A. Zeller. When do changes induce fixes? In *Proc. MSR 2005*, pages 1-5, 2005.

[31] Weka, available at *http://www.cs.waikato.ac.nz/~ml/weka/*

[32] C. C. Williams and J. K. Hollingsworth. Automatic mining of source code repositories to improve static analysis techniques. *IEEE Trans. Software Engineering*, vol. 31, no. 6, pages 466-480, 2005.