EFindBugs: Effective Error Ranking for FindBugs

Haihao Shen[†]*, Jianhong Fang[‡], and Jianjun Zhao[†]
[†]*School of Software*
Shanghai Jiao Tong University, 800 Dongchuan Road, Shanghai 200240, China
{*haihaoshen, zhao-jj*}*@sjtu.edu.cn*
[‡]*Department of Computer Science*
University of California, Los Angeles, Los Angeles, CA 90024, USA
sjtufjh@ucla.edu

***Abstract*—Static analysis tools have been widely used to detect potential defects without executing programs. It helps programmers raise the awareness about subtle correctness issues in the early stage. However, static defect detection tools face the high false positive rate problem. Therefore, programmers have to spend a considerable amount of time on screening out real bugs from a large number of reported warnings, which is time-consuming and inefficient. To alleviate the above problem during the report inspection process, we present EFindBugs to employ an effective two-stage error ranking strategy that suppresses the false positives and ranks the true error reports on top, so that real bugs existing in the programs could be more easily found and fixed by the programmers. In the first stage, EFindBugs initializes the ranking by assigning predefined defect likelihood for each bug pattern and sorting the error reports by the defect likelihood in descending order. In the second stage, EFindbugs optimizes the initial ranking self-adaptively through the feedback from users. This optimization process is executed automatically and based on the correlations among error reports with the same bug pattern. Our experiment on three widely-used Java projects (AspectJ, Tomcat, and Axis) shows that our ranking strategy outperforms the original ranking in FindBugs in terms of precision, recall and F1-score.**

I. Introduction

We strive for a low false positive rate and recommend users reviewing the correctness warnings first. [4]

by FindBugs developers

Static analysis for software defect detection is a promising technique to improve software quality. Recently there has been a surge of interest in using static checking tools [3], [5], [6], [10], [11] to find potential program errors. In addition to finding existing bugs, these tools can also help programmers to prevent future-introduced defects.

Typically, static analysis tools emit error reports about the checking software. However, a challenging problem shared by most static analysis tools, including FindBugs [3], is high false positive rate of defect reports (also called error reports). As described in an empirical evaluation [16], false positive rate can easily reach 30 – 100%. As a result, programmers often spend a great amount of their time on carefully checking error reports one by one in order to screen out real bugs. Moreover, when only few true error reports exist in a large number of error reports, the elimination of false error reports is becoming an arduous, highly manual involved and time-consuming process. However, most static analysis tools employ a simple error ranking scheme to issue error reports, without handling false error reports. For example, error reports in FindBugs are prioritized according to category, pattern, class, or package alphabetically. This ranking method may lead to that some true errors are often residing at the bottom of report list. As reported in previous work [15], these false error reports (also called false positives) can quickly render static detection tools useless by hiding real errors amidst the false. It is also likely to cause users to potentially stop using static analysis tools, especially in detecting large-scale software systems. The usability risk of these tools has motivated the development of novel error ranking methods [15], [16].

In this paper, we present EFindBugs, an improved FindBugs tool with an effective two-stage error ranking strategy. In the first stage, EFindBugs can assign *defect likelihood*[1] for each bug report and automatically rank true reports in the front of false reports initially. To improve the ranking quality of EFindBugs, we first run a sample program – JDK in FindBugs, manually analyze the error reports and count the number of true error reports and false positives to calculate approximate defect likelihood for each bug pattern and *bug kind*[2]. After the initial ranking, whenever the user designates error reports as a different classification such as *not a bug* and *must fix*, EFindBugs can make use of user's designations[3], explore all correlated error reports, recalculate the defect likelihood of those reports and update the ranking to improve the accuracy of ranking. The two-stage error

* Current address: Intel Corporation, APAC R&D, PRC, haihao.shen@intel.com

[1]Defect likelihood denotes the possibility of a reported defect to be a real bug within the range from 0% to 100%.

[2]FindBugs includes seven categories, and each category has many bug kinds. For each bug kind, it contains several bug patterns with similar features (in Figure 1).

[3]In FindBugs, there are seven kinds of designation (described in Section III) and the default is unclassified.

ranking strategy can help users to check those reports that are most likely to be true defects in the first place. Furthermore, users can save time ruling out false positives and have no need to check the reports with a defect likelihood lower than 30% or an expected value according to requirement. We perform an empirical evaluation of EFindBugs on several large open-source projects in Java (like Tomcat [7], AspectJ [1], and Axis [2]). Compared with the original ranking method in FindBugs, EFindBugs improves the precision and recall significantly in the first 60% of error reports, which indicates the effectiveness of our proposed error ranking scheme.

In summary, the main contributions of this paper are:

- An effective error ranking scheme based on defect likelihood has been proposed to improve the existing ranking method in FindBugs. Users can check from the top of the list and easily decide if the rest of bug reports are worth inspecting when defect likelihood is lower than an expected value.
- A self-adaptive improvement procedure has been integrated into EFindBugs through user's designation of error reports. Once the user designates an error report, EFindBugs could search out other correlated bug patterns (or bug kinds) and update the initial ranking. With the feedback from users, defect likelihood for each bug pattern could become more accurate.
- An empirical evaluation is performed on three real-world Java projects. The result indicates EFindBugs demonstrates a great enhancement on the error ranking of FindBugs in terms of precision, recall and F1-score.

The rest of this paper is organized as follows. We first introduce the background of FindBugs and its report structure in Section II. In Section III, we present empirical observations of error reports in FindBugs. Sections IV describes two stages of ranking in EFindBugs. Section V provides experimental setup and results on three open-source projects, AspectJ, Tomcat and Axis. Section VI discusses the possible extensions and threats to validity. Finally, related work and concluding remarks are given in Section VII and VIII, respectively.

II. FindBugs Background

We next briefly introduce the background of FindBugs and the structure of error reports.

FindBugs, one of the most popular static analysis tools, is becoming widely used in Java community. As of July, 2008, it has been downloaded more than 700,000 times. FindBugs implements a set of *bug detectors* for a variety of common *bug patterns* (code idioms that are likely to be errors [12]), and uses them to find a significant number of bugs in real-world applications and libraries [8]. For instance, FindBugs was used to analyze 89 publicly available builds of JDK, and it also plays an essential role in the development of Java programs in Google [8], [19].

Bug patterns in FindBugs are mainly divided into seven categories, that is *Bad Practice*, *Correctness*, *Malicious code vulnerability*, *Multithreaded correctness*, *Performance*, *Security*, and *Dodgy*. For those experienced users, they can define new bug patterns according to their needs [12], [17], [20]. According to the report structure of FindBugs, each *bug category* includes many *bug kinds* and each bug kind consists of several bug patterns. Figure 1 shows a sample structure of error reports in FindBugs. In Figure 1, bug kinds, like *BC*[4], *ICAST*[5], *RV*[6], *DLS*[7], and *DMI*[8], belong to correctness category. And the bug kind *BC* (the abbreviation for **b**ad **c**asts of object references) contains four bug patterns. The structure of error reports serves a fundamental basis in our design of ranking strategy.

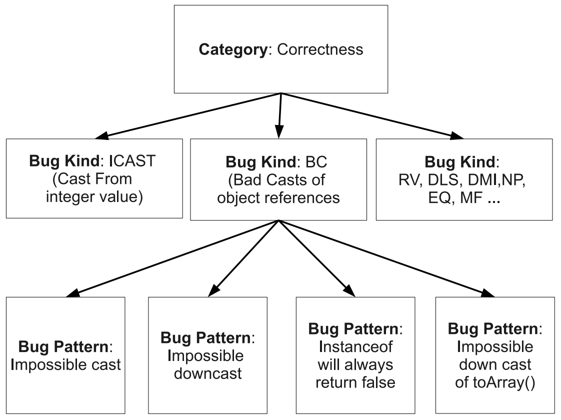
Category: Correctness
Bug Kind: ICAST (Cast From integer value)
Bug Kind: BC (Bad Casts of object references
Bug Kind: RV, DLS, DMI,NP, EQ, MF ...
Bug Pattern: Impossible cast
Bug Pattern: Impossible downcast
Bug Pattern: Instanceof will always return false
Bug Pattern: Impossible down cast of toArray()

Figure 1. Structure of Error Reports in Correctness Category

III. Observations on FindBugs

We next present three main observations based on the implementation of FindBugs (we choose FindBugs 1.3.7 as our target), including error ranking, false positives, and report correlation.

Error Ranking. FindBugs takes three main ranking options into account:

1) Sorting reports in alphabetical order
2) Sorting reports in severity
3) Sorting reports through user's designation

Category, bug pattern, package, class, and version are classified into the first kind of ranking option. However, alphabetical ranking method cannot effectively help users

[4]BC: Bad casts of object references.
[5]ICAST: Cast from integer value.
[6]RV: Bad use of return value from method.
[7]DLS: Dead local store.
[8]DMI: Dubious method invocation

to detect potential defects quickly and easily. In particular, those reports ranked in front are likely to waste much time on identifying real bugs. As for the second option, each error report is assigned a priority value (*High*, *Medium*, or *Low*) in terms of its severity. However, according to our previous work [20], this value is hard coded by the bug detector's developer and remains unchangeable. Moreover, it is also possible that the error reports with high severity are of high false positive rate, since the priority is set according to personal experience of the developers. To successfully eliminate false positives, the third ranking option, namely user's designation, is currently supported to allow users to better arrange the error reports. With this function, users can manually designate each bug report as *need further study*, *not a bug*, *mostly harmless*, *should fix*, *must fix*, *bad analysis* or *unclassified*. Inevitably, they must spend much time on checking error reports one by one in order to make correct designation.

Although several ranking methods have been integrated in FindBugs, users are still facing the high risk of false positives and high cost of the report inspection process when checking large-scale software systems.

False Positives. As described by FindBugs developers, they strive for a low false positive rate for more than 130 bug patterns (nearly one third of all) in correctness category [4]. They recommend that users should review the correctness warnings first to check probable bugs. Empirically, correctness category is mainly used to find real errors in software systems, while other categories are designed to help programmer follow the recommended coding practices and idioms to eliminate confusing codes and improve the performance. Therefore, in current implementation of EFindBugs, we focus on the defects reported in correctness category, and will apply EFindBugs in other categories as our future work.

Report Correlation. Intuitively, we expect error reports of a bug pattern or even a bug kind have the same designation such as *not a bug* and *must fix* so that users can easily handle the error reports in groups. Here, we can make an assumption that error reports from a bug pattern or a bug kind are either false or true. With this assumption, if a user inspects and makes designation for one error report from a certain bug pattern or a bug kind, the rest of error reports belonging to the same pattern or kind will have the same designation. To verify our intuition and assumption, we first investigate the detailed implementation of the bug detectors in FindBugs. Considering our previous experience in XFindBugs [20], we find that each bug detector can detect at least one bug pattern. For instance, the bug detector *VarArgsProblems* detects the bug pattern *VA_PRIMITIVE_ARRAY_PASSED_TO_OBJECT_VARARG* of the bug kind *VA* shown in Table I. In most cases, a detector, such as *FindBadCast2* and *MethodReturnCheck*, can also detect most bug patterns from the same bug kinds. For more examples, please refer to Table I.

Therefore, we can safely assume that error reports of a bug pattern, generated by the same bug detector, are of the same designation. Sometimes, error reports of a bug kind can also have the same designation when the implementation of corresponding detector covers all the bug patterns in this bug kind (e.g., *DuplicateBranches* in Table I). Moreover, it is possible that a detector checks more than one bug kind (e.g., *FindUnrelatedTypesInGenericContainer*). For these two particular cases, we need further study to explore more correlations between different bug kinds. In our current work, we employ the first assumption as our fundamental basis to implement two-stage ranking strategy.

IV. EFindBugs Tool

In this section, we present EFindBugs, an improved FindBugs tool with a two-stage error ranking scheme. The first stage is to make an initial ranking of the generated error reports, and the second stage is to improve the ranking by designating inspected error reports. Before performing the two-stage error ranking, we first define *defect likelihood* as the possibility of a defect to be a real bug within the range from 0% to 100%. Generally, a bug pattern or bug kind with a higher defect likelihood reflects more real error reports in EFindBugs.

A. Initial Ranking

When running a new project, FindBugs will generate a large number of error reports which are designated as *unclassified*, and they are sorted by category and bug pattern alphabetically. Initial ranking is regarded as a very important procedure, because users will lose confidence in static analysis tools and stop checking when encountering too many false positives at the very beginning of the list [15]. In order to perform a good initial ranking, it is necessary to figure out the defect likelihood of each bug pattern and place bug patterns with the lowest false positive rate in the first place.

Initial ranking is composed of three steps. **The first step** is to choose a suitable sample project to approximately measure each bug pattern in correctness category. After testing several projects, we decide to use the most widely-used project – JDK as our sample project. The main reason why we choose JDK as our training project is that JDK serves as a core, fundamental development toolkit to construct upper-level Java applications. More importantly, JDK contains many different kinds of components, such as GUI, network, io, and utility components. Thus, error reports generated in FindBugs could cover as many bug patterns and bug kinds as possible so that we can calculate the initial defect likelihood for most bug patterns and bug kinds in correctness category to improve the ranking quality. Otherwise, in the worst case, EFindBugs, applied in other different programs, may generate all the bug patterns or bug kinds without initial defect likelihood, which will make the

Table I
DETECTORS AND BUG PATTERNS

Detector	Bug Patterns detected by the Detector	Brief Description of Bug Pattern
FindBadCast2	BC_BAD_CAST_TO_CONCRETE_COLLECTION	Questionable cast to concrete collection
FindBadCast2	BC_BAD_CAST_TO_ABSTRACT_COLLECTION	Questionable cast to abstract collection
FindBadCast2	BC_IMPOSSIBLE_INSTANCEOF	instanceof will always return false
FindBadCast2	BC_VACUOUS_INSTANCEOF	instanceof will always return true
FindBadCast2	BC_UNCONFIRMED_CAST	Unchecked/unconfirmed cast
FindBadCast2	BC_IMPOSSIBLE_CAST	Impossible cast
DuplicateBranches	DB_DUPLICATE_BRANCHES	Method uses the same code for two branches
DuplicateBranches	DB_DUPLICATE_SWITCH_CLAUSES	Method uses the same code for two switch clauses
BadSyntaxForRegularExpression	RE_BAD_SYNTAX_FOR_REGULAR_EXPRESSION	Invalid syntax for regular expression
BadSyntaxForRegularExpression	RE_POSSIBLE_UNINTENDED_PATTERN	"." used for regular expression
BadSyntaxForRegularExpression	RE_CANT_USE_FILE_SEPARATOR_REGULAR_EXPRESSION	File.separator used for regular expression
MethodReturnCheck	RV_RETURN_VALUE_IGNORED	Method ignores return value
MethodReturnCheck	RV_RETURN_VALUE_IGNORED_BAD_PRACTICE	Method ignores exceptional return value
MethodReturnCheck	RV_EXCEPTION_NOT_THROWN	Exception created and dropped rather than thrown
VarArgsProblems	VA_PRIMITIVE_ARRAY_PASSED_TO_OBJECT_VARARG	Primitive array passed to function expecting a variable number of object arguments
FindUnrelatedTypesInGenericContainer	GC_UNRELATED_TYPES	No relationship between generic parameter and method argument
FindUnrelatedTypesInGenericContainer	GC_UNCHECKED_TYPE_IN_GENERIC_CALL	Unchecked type in generic call
FindUnrelatedTypesInGenericContainer	DMI_COLLECTIONS_SHOULD_NOT_CONTAIN_THEMSELVES	Collections should not contain themselves
FindUnrelatedTypesInGenericContainer	DMI_USING_REMOVEALL_TO_CLEAR_COLLECTION	Don't use removeAll to clear a collection
FindUnrelatedTypesInGenericContainer	DMI_VACUOUS_SELF_COLLECTION_CALL	Vacuous call to collections

ranking the same as FindBugs. Furthermore, JDK is well-written and includes detailed specifications so that it reduces the difficulty in manually inspecting error reports.

After choosing the sample project, we run JDK in FindBugs and manually check the error reports to count the true error reports and false positives for each bug pattern. There are totally four students involved in the manual classification, and all the students are majored in software engineering with at least three years' experience in Java programming. To ensure the accuracy of the initial ranking, we classify four students into two groups to cross-validate the error reports and then make final decisions. Additionally, the important criteria of classifying FindBugs error reports is that error reports, which contains potential defects, redundant codes and useless codes, will always be considered as true positives, while error reports related to programming habits and functional behaviors are mostly regarded as false positives. For those error reports which are ambiguous and difficult to be classified, we will take a conservative method and mark them as false positives rather than true positives. In this way, we are more confident in the accuracy of the true positives, which are the top concern for users.

Table II shows the part results of error reports for each bug kind and bug pattern in correctness category. During the report inspection process, we find out some features of false positives:

- FindBugs is not able to deal with the methods invocation among different files, which is also the limitation of static analysis tools.
- Some programmers write special codes on purpose, which may violate some bug patterns in FindBugs.
- Repeated codes or similar codes, which are common in many large projects, may result in false positives explosion [15].

The second step is to calculate the defect likelihood for bug patterns and bug kinds in correctness category. We denote *C* as bug category, *K* as bug kind and *P* as bug pattern. For each bug category *C*, it contains bug kinds K_1, K_2, K_3, …, K_m. And for any bug kind K_i, it contains bug patterns P_{i1}, P_{i2}, …, P_{in}. It is clear that for any bug pattern P_{ij} in K_i, $j = 1 \ldots n$, we have $P_{ij} = P_{ij} \cdot_F + P_{ij} \cdot_S$, where $P_{ij} \cdot_F$ is the number of false error reports for bug pattern P_{ij}, and $P_{ij} \cdot_S$ is the number of true error reports. We can easily calculate the defect likelihood of each bug pattern $D(P_{ij})$ in the following equation and use $D(P_{ij})$ to rank the error reports roughly.

$$D(P_{ij}) = \frac{P_{ij} \cdot_S}{P_{ij} \cdot_F + P_{ij} \cdot_S} \tag{1}$$

In order to avoid the inequity of calculating defect likelihood (due to different population size of bug patterns), we consider the variance $V(P_{ij})$ as an additional indicator for bug pattern P_{ij}.

$$V(P_{ij}) = \frac{D(P_{ij}) \times (1 - D(P_{ij}))}{n} \tag{2}$$

Suppose that two bug patterns have the same defect likelihood, the one with larger population will have smaller value of variance, which means the change degree of this bug pattern population is lower and the corresponding error reports should be examined first. Once we have the defect likelihood of each bug pattern $D(P_{ij})$, we can continue to calculate the defect likelihood for each bug kind $D(K_i)$.

$$D(K_i) = \frac{P_{i1} \times D(P_{i1}) + P_{i2} \times D(P_{i2}) + \cdots + P_{in} \times D(P_{in})}{P_{i1} + P_{i2} + \cdots + P_{in}} \tag{3}$$

Table II
DEFECT LIKELIHOOD FOR BUG KIND AND BUG PATTERN

Type	Name	Real Defects	False Positives	Defect Likelihood
Bug Kind	Bad use of return value from method	2	3	40%
Bug Pattern	RV_EXCEPTION_NOT_THROWN	2	0	100%
Bug Pattern	RV_RETURN_VALUE_IGNORED2	0	3	0%
Bug Kind	Casting from integer values	0	3	0%
Bug Pattern	ICAST_INT_CAST_TO_DOUBLE_PASSED_TO_CEIL	0	2	0%
Bug Pattern	ICAST_INT_CAST_TO_FLOAT_PASSED_TO_ROUND	0	1	0%
Bug Kind	Confusing method name	3	1	75%
Bug Pattern	NM_LCASE_HASHCODE	0	1	0%
Bug Pattern	NM_METHOD_CONSTRUCTOR_CONFUSION	2	0	100%
Bug Pattern	NM_WRONG_PACKAGE	1	0	100%
Bug Kind	Masked Field	12	7	63%
Bug Pattern	MF_CLASS_MASKS_FIELD	12	7	63%
Bug Kind	Repeated conditional test	7	2	78%
Bug Pattern	RpC_REPEATED_CONDITIONAL_TEST	7	2	78%
Bug Kind	Test for floating point equality	6	0	100%
Bug Pattern	FE_TEST_IF_EQUAL_TO_NOT_A_NUMBER	6	0	100%
Bug Kind	Uninitialized read of field in constructor	0	5	0%
Bug Pattern	UR_UNINIT_READ	0	5	0%
Bug Kind	Unwritten field	2	12	14%
Bug Kind	UWF_NULL_FIELD	0	5	0%
Bug Kind	UWF_UNWRITTEN_FIELD	2	7	22%

In Equation (3), we ignore the population size of error reports in bug kind. For example, suppose that there are two bug kinds K_1 and K_2, which both contain two bug patterns. In K_1, each bug pattern has 50 false positives and 50 true error reports so that the defect likelihood for K_1 is 50%. In K_2, one bug pattern has 100 true error reports and 0 false positive, while the other bug pattern has 0 true error report and 100 false positives. As a result, the defect likelihood for K_2 is also 50%, and these two bug kinds have the same defect likelihood. However, considering the discrete degree, it is better to examine the bug kind K_1 first because of its centralized distribution. The following equation is used to calculate the degree of discretization for bug kind:

$$S^2 = \frac{1}{n-1}\sum(D(P_{ij}) - D(K_i))^2 \tag{4}$$

The final step is to assign the value of defect likelihood and variance to bug patterns and bug kinds in correctness category. With that, EFindBugs is able to initialize the ranking for other projects automatically, according to the built-in ranking. If error reports are sorted by defect likelihood of bug patterns in EFindBugs, we can get a best ranking output. On the other hand, with the sacrifice of precision, it is easier for users to inspect error reports sorted by defect likelihood of bug kinds, because bug patterns in one bug kind focus on one type of defects, which are similar to each other. At the same time, we can find that the defect likelihood for approximately 80% of bug patterns is close to 100% or 0% rather than 50% (partly shown in Table II), which renders us that error reports of the same bug pattern are all false positives or true positives. This statistical observation can also verify the correctness of the assumption in Section III that error reports of a bug pattern are of the same designation.

B. Ranking Optimization

Ideally speaking, we hope that the initial ranking can get close to optimal ranking that all the true error reports are on the top of the list [16]. However, the initial ranking may not be accurate due to the reasons as follows.

- The new checking project is of different type from the sample project (JDK).
- The sample project does not cover enough bug patterns and bug kinds.
- Users may focus on specific kinds of error reports to meet their requirements rather than all error reports.

Consequently, it is necessary to optimize the ranking automatically through the users' feedback. The optimization is divided into two parts. **The first part** is to reset defect likelihood for new bug patterns and bug kinds, which are not covered by our sample project. If EFindBugs generates a new bug pattern in an existing bug kind, then we can directly assign the defect likelihood of the bug kind to the new bug pattern; if there is a new bug kind, then we simply set 50% to be the defect likelihood of the bug kind and its bug patterns. However, it is not clear yet whether the ranking can be improved by changing the defect likelihood to some other value rather than 50%. If new bug pattern or bug kind contains a great many error reports (such as false positives explosion [15]), it is reasonable to assign a value below 50%. One the contrary, bug patterns or bug kinds containing only few error reports could be assigned a value above 50% [16]. How to select the optimal defect likelihood for a new bug kind will be one of our future works.

The second part is to dynamically rank the error reports by collecting the user's designation to optimize the ranking. This optimization is based on the observation that error reports of the same bug pattern frequently have the same designation (described in Section III). Without loss of generality, this optimization is also suitable for other categories in FindBugs. When a user designates a bug pattern, EFindBugs can find all error reports from this bug pattern and make the same designation. Then, the defect likelihood and variance of the bug pattern and its corresponding bug kind will be recalculated. In result, EFindBugs will sort error reports again to suppress the false positives and prioritize bug patterns of low false positive rate. Since some bug kinds share the same designation, if EFindBugs sets the same designation for each bug kind, the user may save much time on inspecting error reports, however the accuracy will decrease correspondingly.

Generally, the user needs to check only one error report for each bug pattern and then EFindBugs automatically places all the true error reports on the top of the list to get close to the optimal ranking. The ranking optimization is of great significance because it changes traditional one-time error ranking in static analysis tools and makes use of users' feedback to self-adaptively improve the ranking as they need.

V. Empirical Evaluation

This section presents the results produced by EFindBugs on three Java open-source projects, Tomcat 6.0.18, AspectJ 1.6.4 and Axis 1.4. We next describe the objective, the subjects, experimental procedures and results.

A. Objective

The objective of this experiment is to consider the following questions:

- Does EFindBugs improve the ranking of error reports compared to FindBugs ?
- Can users find potential bugs more quickly and easily in EFindBugs ?
- Can EFindBugs be applied to large-scale Java applications ?

B. Subject Programs

In our experiment, we use three open-source Java applications as our subject programs. Apache Tomcat 6.0.18 (Tomcat for short) [7] is a widely used web application server all over the world. AspectJ 1.6.4 (AspectJ for short) [1] is a seamless aspect-oriented extension to Java programming language and integrates new features, like pointcut, advice and intertype declaration. Axis 1.4 (Axis for short) [2] is an implementation of the SOAP (Simple Object Access Protocol) submission to W3C.

We select different types of applications as subject programs to cover as many bug patterns and bug kinds as possible. We run FindBugs on subject programs and count the error reports in Correct category. In addition, we manually inspect the error reports to figure out true error reports and false positives. Table III shows the name of subject program (Name), number of lines of code in total (LOC), number of error reports (#ER) in Correctness category, number of false positives (#FP), and number of true error reports (#TE). to use these data to measure the precision and recall value.

Table III
Inspection Results for Subject Programs

Name	LOC	#ER	#FP	#TE
Tomcat	171449	86	47	39
AspectJ	391369	52	34	18
Axis	86373	36	23	13

C. Comparison Indicators

We next compare the ranking results of EFindBugs with those of FindBugs to show the effectiveness of our error ranking strategy. To simplify our experiment, we mainly concentrate on the comparison of the initial ranking of error reports (with less ranking optimization). We employ three indicators, *precision*, *recall*, and *F1-score* to quantitatively reflect the comparison result. Precision and recall are two widely-used measures in evaluating the exactness and completeness in a statistical classification task, and F1-score is another test accuracy indicator to measure the harmonic mean of precision and recall.

By ranking error reports, the total number of false positives cannot be reduced. If we take all error reports into consideration, the precision and recall will remain the same value. Our approach is to compute the precision, recall, and F1-score and compare EFindBugs with FindBugs from the first of 10%, 20% to 100% of error reports. Therefore, the user only needs to check the error report from the top of the list and find real bugs rather than manually eliminate false positives one by one. The precision, recall and F1-score can be computed by the following equations (5), (6), and (7) respectively.

$$Precision = \frac{\#True\ Positives}{\#True\ Positives\ +\ \#False\ Positives} \tag{5}$$

$$Recall = \frac{\#True\ Positives}{\#True\ Positives\ +\ \#False\ Negatives} \tag{6}$$

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{7}$$

In Equations (5) and (6), false positives refer to false error reports, true positives are corresponding to true error reports, and false negatives are true errors not reported. In our experiment, a perfect precision score of 1.0 means that each inspected error report is a true error report whereas a perfect recall score of 1.0 means that all the true error reports are included in the inspected error reports. Ideally speaking, our self-adaptive ranking strategy will make both the precision and recall close to the optimal gradually and therefore F1-score will also reach the optimal.

D. Results and Analysis

In our experiment, we rank the error reports in EFindBugs by the defect likelihood of bug patterns, while error reports in FindBugs are sorted alphabetically by bug patterns, which is the default ranking strategy in FindBugs. The percentage of error reports is used as abscissa and precision, recall as ordinate in the following figures (Figures 2 - 7).

1) Tomcat: Figure 2 shows the comparison of ranking in precision and recall between EFindBugs and FindBugs. We could see from Figure 2 that EFindBugs performs a good ranking in the first 20% of error reports in terms of precision (precision value reaches up to 100%). In FindBugs, the precision of the first 10% error reports only gets to nearly 25%, which may make users stop checking and discard the static analysis tool. Furthermore, the recall of EFindBugs is much better than FindBugs. In EFindBugs, users can find out nearly 85% of real bugs after checking the first 50% of error reports. However, in FindBugs, users have to check 80% of error reports to find out only 70% of real bugs. Based on precision and recall, Figure 3 shows better F1-score in EFindBugs due to the benefit from precision and recall. This comparison result indicates that EFindBugs can effectively control false positives and save time on checking enormous error reports. On the other hand, EFindBugs can improve the ranking gradually by collecting users' designations when inspecting error reports, whereas the ranking in FindBugs is static and unchangeable.

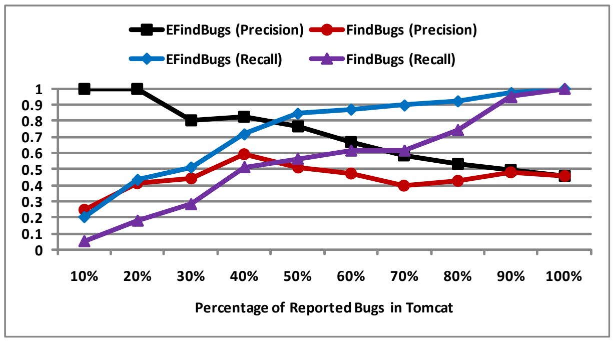

Figure 2. Precision & Recall on Tomcat

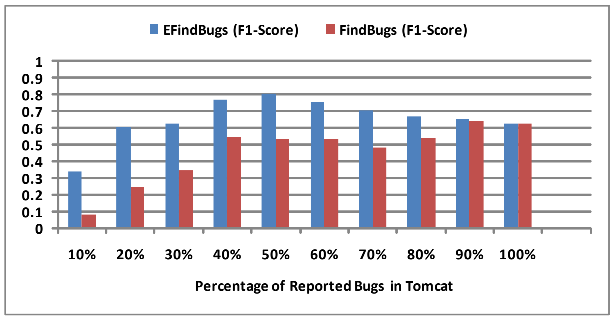

Figure 3. F1-Score on Tomcat

2) AspectJ: In the case of AspectJ, EFindBugs generates 52 error reports in correctness category; 18 out of which are true error reports. From Figure 4, we can observe that the ranking in EFindBugs is better than that in FindBugs both in precision and recall. Correspondingly, the overall F1-score in EFindBugs is also higher than that in FindBugs, which is shown in Figure 5.

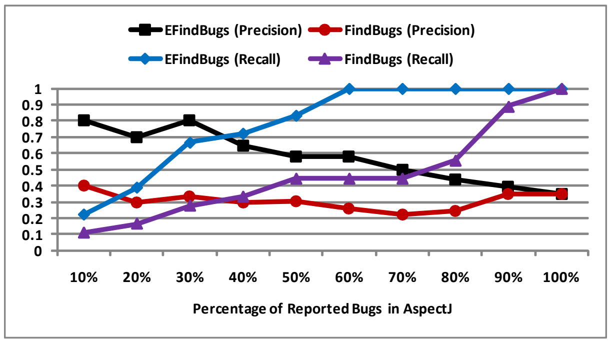

Figure 4. Precision & Recall on AspectJ

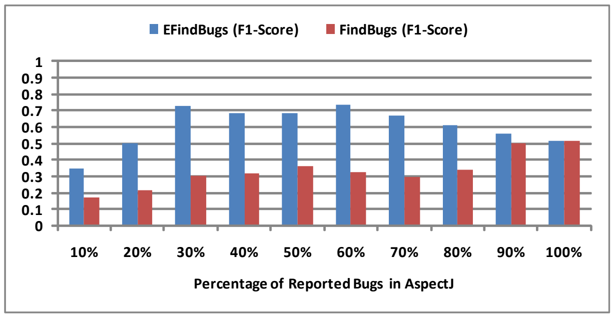

Figure 5. F1-Score on AspectJ

For FindBugs, the precision is lower than 40% from 10% to 100%, and it is down to 20% for the first 70% of error reports, which means users waste much time on checking the false positives. For EFindBugs, the best precision can get to 80% for the first 10% and 30% of error reports. Though the precision decreases slightly, the precision can still remain nearly 57% for the first 60% of error reports. Considering recall indicator (right part in Figure 4), it is easy to find that the recall value reaches up to 100% quickly at the point of 60% of error reports. On the contrary, the users using FindBugs have to check all the error reports to discover all potential bugs. Through the combination of these two graphs (left and right parts in Figure 4), we can find that users can detect all potential bugs in EFindBugs after inspecting 60% of error reports with 57% precision value, which undoubtedly performs better than FindBugs.

3) Axis: As for Axis, FindBugs totally generates 36 error reports in correctness category and 13 out them are true error reports. We also compare the use of EFindBugs and FindBugs in error reports ranking, which is shown in Figure 6 and Figure 7.

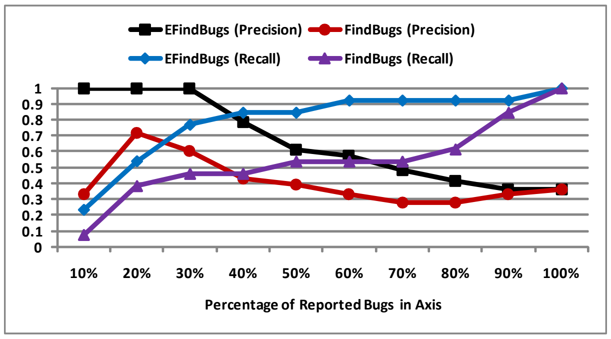

Figure 6. Precision & Recall on Axis

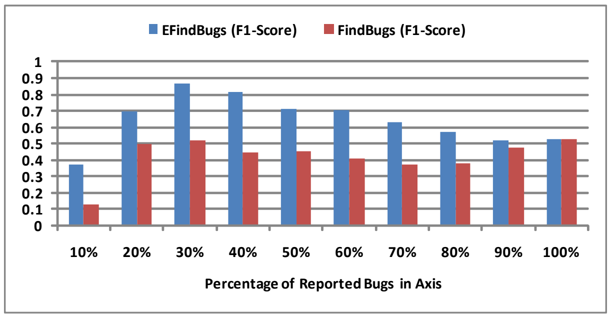

Figure 7. F1-Score on Axis

At the first glance, the ranking in FindBugs seems good in the first 20% of error reports with a precision of 72% and a recall of 45%. However, the precision drops down quickly to 39% and the recall is only 52% after inspecting 70% of error reports. It is not surprising that FindBugs may have a good ranking in some cases if a bug pattern of low false positive rate is listed on top alphabetically and the checking project happens to have lots of defects associated with this bug pattern. Regardless of defect likelihood of bug pattern or bug kind, FindBugs ranks the categories and bug patterns in alphabetical order. As a result, the ranking in FindBugs cannot always be efficient in various projects.

As for EFindBugs, it still performs a better ranking in Axis, the accuracy reaches to 100% and recall is up to 75% in the first 30% of error reports. Unfortunately, the precision drop down by nearly 20% at the point of 40% and 50% of error reports. The reason for that is the target project is quite different from the sample project and some bug patterns turn to be those of a higher false positive rate in the new project. However, this problem can be solved by ranking optimization in EFindBugs if the user makes designation to false error reports.

E. Comparison with Z-ranking

To show the effectiveness of EFindBugs, we also compare our ranking strategy with the existing error ranking technique Z-ranking [16]. Z-ranking employed *z-test* statistical model and introduced two hypotheses for error ranking:

- Weak hypothesis: true error reports come with many successful checks.
- Strong hypothesis: false error reports come with many failed checks (many other error reports).

As a general ranking technique, Z-ranking can be adapted to sort the error reports generated from three open-source projects in Java. Based on the above hypotheses, we compare the result of Z-Ranking with that of our ranking strategy. We can see from Table IV that our ranking strategy can be very effective at least for the subjective programs we investigated. For the first half of error reports, our ranking result is more precise (almost 2X) than Z-ranking in detecting real bugs. Besides, users have to inspect more error reports (1.6X) to find out 50% of real bugs if they use Z-ranking rather than our ranking strategy. Based on precision and recall, the overall F1-score for EFindBugs is 40% higher than that for Z-ranking, which could be observed in Figure 8. The main reason for this is that we manually investigated the error reports in sample projects and provided more accurate information for each detector. As mentioned in Kremenek et al's paper [16], prior information of source codes or detectors, collected by hand, statistics or machine learning techniques, can improve the ranking quality. On the other hand, with the ranking optimization in EFindBugs, our ranking will become far better than Z-ranking.

F. Experimental Conclusion

In our experiment, EFindBugs shows its effectiveness in ranking of error reports on three various large Java projects: Tomcat, AspectJ and Axis. The ranking result of EFindBugs is better than that of FindBugs both in precision and recall, especially in the first 60% of error reports. Furthermore, the comparison is based on the initial ranking of EFindbugs and Findbugs, and if the user makes designation for error reports, the ranking of EFindbugs will be further improved. The above empirical evaluation demonstrates that our two-stage ranking scheme is of great help to suppress false positives and detect real bugs more quickly and easily. And we recommend users to employ EFindBugs as a supplementary tool for FindBugs to detect potential defects in large Java applications.

VI. Discussion

In this section, we discuss the possible extensions to EFindBugs and some threats to validity in our work.

A. Possible Extensions

In our work, EFindBugs is only available for bug patterns in the correctness category. Without any particularity in our ranking scheme, we believe that we can also apply the ranking method to other categories, such as *Multi-threaded correctness*, *Security*, *Malicious code vulnerability*, etc. Moreover, other static analysis tools for Java programs, including JLint and PMD, can also employ this ranking

Table IV
PRECISION AND RECALL COMPARISON WITH Z-RANKING

Reported Bugs (%)	Axis				AspectJ				Tomcat			
	E_Pre	Z_Pre	E_Rec	Z_Rec	E_Pre	Z_Pre	E_Rec	Z_Rec	E_Pre	Z_Pre	E_Rec	Z_Rec
10%	1	0.33333	0.23077	0.07692	0.8	0.6	0.22222	0.16667	1	0.25	0.20513	0.05128
20%	1	0.71429	0.53846	0.38462	0.7	0.4	0.38889	0.22222	1	0.29412	0.4359	0.12821
30%	1	0.8	0.76923	0.61538	0.8	0.4	0.66667	0.33333	0.8	0.48	0.51282	0.30769
40%	0.78571	0.64286	0.84615	0.69231	0.65	0.35	0.72222	0.38889	0.82353	0.44118	0.71795	0.38462
50%	0.61111	0.5	0.84615	0.69231	0.57692	0.34615	0.83333	0.5	0.76744	0.48837	0.84615	0.53846
60%	0.57143	0.42857	0.92308	0.69231	0.58065	0.29032	1	0.5	0.66667	0.52941	0.87179	0.69231
70%	0.48	0.4	0.92308	0.76923	0.5	0.25	1	0.5	0.58333	0.5	0.89744	0.76923
80%	0.41379	0.41379	0.92308	0.92308	0.43902	0.34146	1	0.77778	0.52941	0.44118	0.92308	0.76923
90%	0.36364	0.36364	0.92308	0.92308	0.3913	0.34783	1	0.88889	0.49351	0.49351	0.97436	0.97436
100%	0.36111	0.36111	1	1	0.34615	0.34615	1	1	0.45349	0.45349	1	1

E_Pre: Precision in EFindBugs, Z_Pre: Precision in Z-ranking, E_Rec: Recall in EFindBugs, and Z_Rec: Recall in Z-ranking

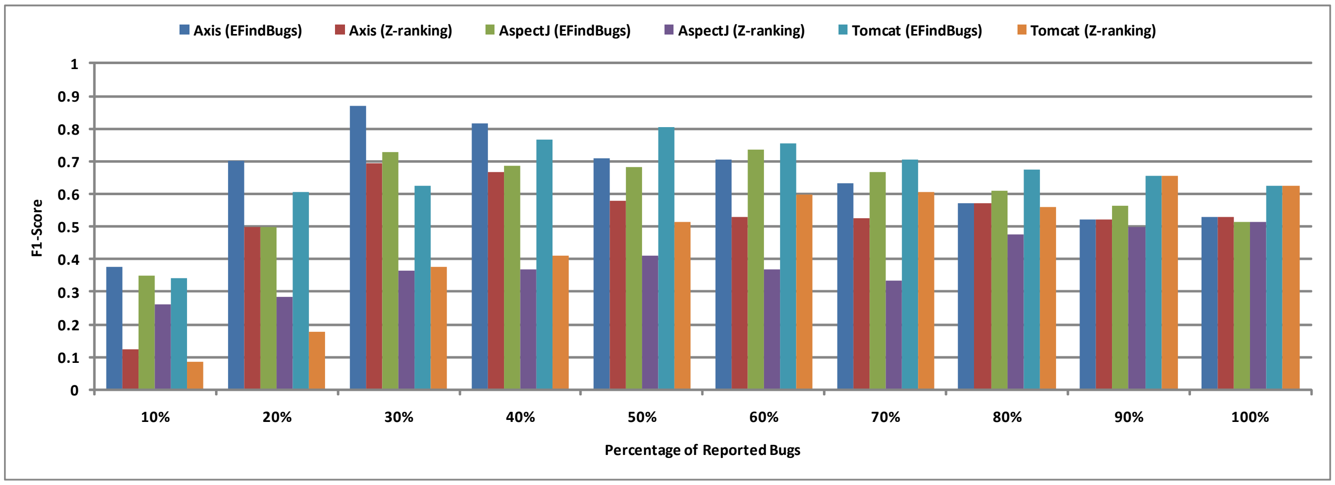

Figure 8. F1-Score Comparison with Z-ranking

method. For different static analysis tool, we first need to investigate the different implementation and explore implicit correlations between error reports. Second, it is necessary to inspect one or more sample projects to calculate the defect likelihood for each type of error reports as the initial ranking. Finally, we can make use of the correlation to update the ranking based on user's feedback.

B. Threats to Validity

Like any empirical evaluation, there are some threats to validity which must be taken into consideration in our experiment. Though we use JDK as our sample project, it only covers 39 out of 131 bug patterns in correctness category. It is better to choose several various large-scale projects as a sample set to cover as many bug patterns as possible to improve the initial ranking. Moreover, manually inspecting error reports in JDK, Tomcat, AspectJ and Axis may be not accurate enough due to lack of domain knowledge and implementation details. To alleviate this problem, we try to find out the false positives and true error reports through differences among versions [3], [14]. Another alternative solution is to collect feedback from the developers of the sample project because their designations of error reports are accurate and of great use for initial ranking.

VII. RELATED WORK

We next discuss some related work in the area of error ranking for static analysis tools and warning predication for software systems.

Error Ranking. Several research work focused on error ranking of static checkers for C and C++ programs. Kremenek and Engler [16] proposed *z-ranking* algorithm to rank errors, based on the observation that true error reports tend to issue few failed checks while false positives always generate lots of failed checks. Our ranking strategy in EFindBugs is different from theirs in that we first choose sample projects and analyze the results to figure out the approximate defect possibility for each bug pattern and bug kind, and then assign initial ranking for new projects. The most similar work to ours was *Feedback-Ranking* presented by Kremenek *et al.* [15]. It also included an initial prioritization and adaptive learning from inspected error reports. However, Feedback-Ranking depended on the strong clustering of false positives and made use of code locality to find correlated warning for ranking optimization, which was totally different from EFindBugs. In EFindBugs, we use statistics of sample project to initialize the error reports and self-adaptively improve the ranking through designation of error report.

Kim and Ernst [13] presented a warning prioritization by analyzing the software change history. They computed the lifetime of each warning category and sorted warning categories based on its observed lifetime. Then, they applied their warning ranking method on three static analysis tools for Java programs, namely FindBugs, JLint, and PMD. The underlying intuition employed in their approach was that if warnings from a category are resolved quickly by developers, the warnings in the category are important. However, they did not consider warning correlation in software transaction change.

Warning Predication. Experimental program analysis [18] provided an effective approach for software engineers to deduce and infer characteristics of software systems. Ruthruff *et al.* [19] carried out an experimental approach to predict accurate and actionable static analysis warnings. They initially ran FindBugs on the code base in Google, and then manually validated each error report to classify true and false error reports. Based on the training data, they employed logistic regression model to determinate the warning probability of reports.

Boogerd and Moonen [9] considered the likelihood that program execution reaches locations for which warnings are generated by a static software checking tool. They discussed a novel demand-driven algorithm for computing execution likelihood based on the system dependence graph and assessed the quality of predictions by comparing them to actual values obtained by dynamic profiling.

VIII. Concluding Remarks

In this paper, we presented EFindBugs with two-stage ranking scheme to improve the ranking quality in FindBugs. EFindBugs first employed statistics of error reports generated in JDK to calculate the defect likelihood for each bug pattern and bug kind and then dynamically updated ranking by making use of user's designation. We performed empirical evaluation of EFindBugs on three open-source projects in Java and compared the ranking result with that of FindBugs. The comparison results illustrated that EFindBugs is an effective tool for error ranking in large Java applications.

In the future, we would like to investigate more projects as the sample set to improve the initial ranking of error reports. We also plan to explore more correlations between different bug patterns and bug kinds to refine the ranking optimization. We would also like to adapt the ranking strategy in EFindBugs to make effective ranking of error reports available for other static analysis tools.

To encourage evaluation and further research in these and other directions, the source code of EFindBugs and experiment data and results is available at

http://stap.sjtu.edu.cn/~shen/EFindBugs/

Acknowledgements. We would like to thank the anonymous reviewers for their useful comments and suggestions. This work was supported in part by National Natural Science Foundation of China (NSFC) (Grant No. 60673120 and 60970009).

References

[1] AspectJ. http://eclipse.org/aspectj/.

[2] Axis. http://jlint.sourceforge.net/.

[3] FindBugs. http://findbugs.sourceforge.net/.

[4] FindBugs Demo. http://findbugs.sourceforge.net/demo.html.

[5] JLint. http://jlint.sourceforge.net/.

[6] PMD. http://pmd.sourceforge.net/.

[7] Tomcat. http://tomcat.apache.org/.

[8] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou. Evaluating static analysis defect warnings on production software. *Proc. PASTE*, pages 1–8, 2007.

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

[10] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proc. PLDI*, pages 234–245, 2002.

[11] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *Tenth International Workshop on Model Checking of Software (SPIN), pages 235–239*, 2003.

[12] D. Hovemeyer and W. Pugh. Finding bugs is easy. In *Proc. OOPSLA*, pages 92–106, 2004.

[13] S. Kim and M. Ernst. Prioritizing warning categories by analyzing software history. In *Proc. MSR*, 2007.

[14] S. Kim and M. Ernst. Which warnings should I fix first? In *Proc. FSE*, pages 45–54, 2007.

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

[16] 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.

[17] N. Rutar, C. Almazan, and J. Foster. A Comparison of Bug Finding Tools for Java. In *Proc. ISSRE*, pages 245–256, 2004.

[18] J. Ruthruff, S. Elbaum, and G. Rothermel. Experimental program analysis: a new program analysis paradigm. In *Proc. ISSTA*, pages 49–60, 2006.

[19] 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.

[20] H. Shen, S. Zhang, J. Zhao, J. Fang, and S. Yao. XFindBugs: eXtended FindBugs for AspectJ. In *Proc. PASTE*, pages 70–76, 2008.