Comparing Static Bug Finders and Statistical Prediction

Foyzur Rahman[†] Sameer Khatri[†] Earl T. Barr[‡] Premkumar Devanbu[†]

[†]Department of Computer Science
University of California Davis
Davis, CA 95616, USA
{mfrahman, sakhatri, ptdevanbu}@ucdavis.edu

[‡]Department of Computer Science
University College London
London, WC1E 6BT, UK
e.barr@ucl.ac.uk

ABSTRACT

The all-important goal of delivering better software at lower cost has led to a vital, enduring quest for ways to find and remove defects efficiently and accurately. To this end, two parallel lines of research have emerged over the last years. *Static analysis* seeks to find defects using algorithms that process well-defined semantic abstractions of code. *Statistical defect prediction* uses historical data to estimate parameters of statistical formulae modeling the phenomena thought to govern defect occurrence and predict where defects are likely to occur. These two approaches have emerged from distinct intellectual traditions and have largely evolved independently, in “splendid isolation”. In this paper, we evaluate these two (largely) disparate approaches on a similar footing. We use historical defect data to apprise the two approaches, compare them, and seek synergies. We find that under some accounting principles, they provide comparable benefits; we also find that in some settings, the performance of certain static bug-finders can be enhanced using information provided by statistical defect prediction.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Code Inspections and Walk-Throughs*

General Terms

Experimentation; Measurement; Reliability; Verification

Keywords

Software Quality; Fault Prediction; Inspection; Empirical Software Engineering; Empirical Research

1. INTRODUCTION

Given the centrality of software in modern life and the fallibility of human programmers, better software quality is a never-ending quest. Software quality-control involves several distinct approaches, including testing, inspection, and formal verification. In this paper, we use historical defect data to comparatively evaluate and search for synergies between two approaches that have, of late, been of tremendous interest both in academia and industry: *static bug-finding* and *statistical defect prediction*

Static Bug-Finding ($\mathcal{SBF}$): These approaches range from simple code pattern-matching techniques to rigorous static analyses that process carefully designed semantic abstractions of code; all $\mathcal{SBF}$ tools find and report likely defect locations in code. These likely locations are reported to programmers (typically) at coding-time. The pattern-matching techniques, such as PMD, are unsound but scale well and have been effectively employed in industry. Hybrid tools like FINDBUGS incorporate both static data-flow analysis, and pattern matching. Tools like ESC-Java [8] and CodeSonar[1] are slower, but their analyzes have desirable *soundness* properties (See Section 2.2). In practice, however, reported warnings are infested with both false positives and/or false negatives. False positives can waste a developer's time, and false negatives can allow defects to escape and cause customer grief. Developers usually cannot determine the truth or falsity of a given warning without examining code.

Defect Prediction ($\mathcal{DP}$): Extensive and widespread data-gathering in modern software processes stimulates this approach. Animated by theories of the human and technical factors that durably influence errors, researchers have channeled this deluge of data to estimate rich statistical (and machine-learning) models that *predict* where defects can occur. Given the relative immutability of human nature, models are expected to be quite stable over relatively long time intervals. Thus, it has been believed statistical methods can predict where defects are likely to occur in the future. Empirical evaluations using fairly sophisticated economic models suggest that these methods are indeed likely to be effective in helping to locate defects [2, 5, 12]

Motivation/Barriers: These approaches have emerged from parallel and disparate traditions of intellectual thought: one driven by algorithm and abstraction over code, and other by statistical methods over large defect datasets. These differences are perhaps analogous to the Chomsky/Norvig debate in computational linguistics (see Section 2).

Although the “holy grail” of $\mathcal{SBF}$ is automatically proving a correctness property of a program, *i.e.* certifying the program free of a certain class of bugs, undecidability means that, in practice, $\mathcal{SBF}$ produces warnings that identify *lines* for developer inspection. For its part, $\mathcal{DP}$ identifies <u>*files*</u> for

[1]`http://www.grammatech.com/codesonar`

inspection. In practice, then, these approaches tackle the same problem: *improving inspection efficiency*, the problem of finding minimal, potentially defective regions in source for careful inspection. However, neither is infallible.

Although, in practice, these two approaches share the same goal, their performance is difficult to compare. We call this the *comparability of bugginess identification* (CBI) problem; This problem comprises two subproblems: *zonation* and *measurement.* The zonation problem concerns the granularity at which a tool reports bugginess: intra-line, line, statement, scope, block, method, file, *etc.*. While most defects are local and contiguous, a single defect can be arbitrarily scattered in the sense that fixing it requires changes to lines throughout a codebase. Defect scatter causes the *measurement* problem: given a particular zonation, how should one measure bug identification? Choices for a binary score include 1) the reported zone must subsume the defect or 2) it must simply intersect the defect; choices for a fractional score include 1) the ratio of the size of intersection to the size of the defect or 2) their Jaccard index.

The IEEE Computer Society defines software engineering as "(1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)." [24]; its goal is to find ways to produce better, cheaper software faster. The CBI problem, therefore, cuts to the core of software engineering: neglecting it violates this imperative because it not only leaves unanswered which approach is better than the other under what circumstances, but it also forgoes the exploration of their synergies in service of this goal. Both approaches have attracted considerable investment, which could be better guided by a good answer to the former question; failure to address the latter has stymied cross-pollination between the two traditions. We take the first steps toward solving it and building a bridge between $\mathcal{SBF}$ and $\mathcal{DP}$.

The two approaches offer very different trade-offs. $\mathcal{DP}$ tools are very easy to implement: once adequate history is available, programming language, build environment, platform evolution, *etc.* are immaterial; indeed, process metadata is very easily extracted from bug repositories and version control has proven to be much more valuable than metrics based on source code content [19]. On the other hand, $\mathcal{DP}$ generally operates at a coarse granularity, typically suggesting entire files for inspection. $\mathcal{SBF}$ tools, including bug-finders, are fine-grained, suggesting individual lines to inspect; but they are language- and platform-specific; they require a variety of compilation, thus carrying the baggage of build procedures that vary with language, platform and even time. $\mathcal{SBF}$ tools can be very difficult to deploy[2]. Indeed, our dataset of warnings from bug-finding tools is very hard-won, and we claim the public availability of our dataset is itself an important stepping-stone for future work on comparing the two and searching for synergies.

In current state-of-the-art $\mathcal{SBF}$ and $\mathcal{DP}$ techniques and tools, the CBI problem first manifests itself in differences in how the two traditions have done their evaluation. In terms of zonation, $\mathcal{DP}$ works at file granularity while $\mathcal{SBF}$ works at line granularity; this zonation mismatch complicates the choice of a bug-identification measure. Solving the CBI problem, and comparing the two approaches is **vital**, given the investments in using them and the high human and economic cost of software defects. Section 3.2 presents the choices we made and the techniques we deployed to overcome these challenges and present a first solution to the CBI problem.

[2]See [4] for vivid testimonials on deployment difficulties.

We empirically study the value of three different static bug-finding tools, FindBugs, Jlint, and pmd, in conjunction with statistical defect prediction, using a very large set of historical defects.

- We formulate and present two cost-effectiveness-based accounting approaches to the *comparability of bugginess identification* problem (CBI).
- We find no significant differences in the cost-effectiveness of the $\mathcal{DP}$ and the two $\mathcal{SBF}$ techniques — FindBugs and pmd— we could evaluate.
- We find $\mathcal{SBF}$ and $\mathcal{DP}$ do sometimes find different defects.
- We find the incorporation of metrics derived from $\mathcal{SBF}$ does not significantly improve the performance of $\mathcal{DP}$ techniques.
- We find that $\mathcal{SBF}$ tools frequently perform better when ordering their warnings using priorities produced by $\mathcal{DP}$ than when using their native priorities.
- Finally, we have (with a great deal of effort) created a multi-release repository of warnings from three tools, Jlint, pmd, and FindBugs, which we shall make publicly available for future research.

These findings are *actionable*; as explained earlier, $\mathcal{DP}$ are independent of language and build procedures, and work very well with process meta data [19]; if they could provide comparable benefits, this is good news for projects that have abundant process data, but have complex, multilingual source bases and build procedures that (see [4]) inhibit the adoption of $\mathcal{SBF}$ tools. It is also interesting that they do find different defects in some cases, so that if budgets allow, it might be worthwhile to use both. Finally, under one measurement regime, $\mathcal{DP}$ appears to be almost always a better way to order $\mathcal{SBF}$ warnings, compared to their native priority-ordering.

2. BACKGROUND

The contrast between $\mathcal{SBF}$ and $\mathcal{DP}$ has parallels in the Chomsky-vs-Norvig debate about statistical *vs.* first principles approaches to natural language processing[3]. Chomsky tends to favor first-principles approaches to NLP, whereas Norvig argues that NLP should be based on the statistics of human linguistic behavior. Defects in software, likewise, arise from the interaction of formal semantics of PL and human behavioural imperfections.

$\mathcal{SBF}$ researchers observe a pattern in defect occurrence, use PL semantics to theorize how such defects could be found, and then engineer abstractions and algorithms to find these defects. $\mathcal{DP}$ researchers observe patterns in human behaviors that cause defects, and (because formal theories of

[3]Norvig-vs-Chomksy and the fight for the future of AI at `http://www.tor.com`, June 2011.

humans do not exist), pursue the ML approach of extracting features and letting learning algorithms do the work of fitting the data. Perhaps because of different intellectual traditions, the two have never been properly compared and synthesized. In this paper, we first attack the problem of comparing the two on an equal footing.

(Prelude) Defect data: Most modern software projects follow software processes that require extensive data gathering. Version control tools such as GIT track every change made to the code. Tools such as BugZilla and JIRA are widely used to record bug reports, track the associated discussion and task allocation, and eventually relate the fix to a specific commit in a specific set of files by a specific individual. It is thus possible to find where (which files and line numbers) defect repairs occurred. Because of the careful tracking of changes by version control systems such as GIT, it is actually possible to precisely track the *provenance* of each line of code; it is thus possible to track, for every line of code that is changed to fix a defect, exactly where that line came from. A large body of research has been animated by the availability of this data, and a steady stream of results in defect prediction and the etiologies of defects has been coming out in recent years. In this paper, we use this data to evaluate static analysis tools and statistical defect prediction, and to compare the two and seek synergies.

2.1 Statistical Defect Prediction

Statistical defect prediction ($\mathcal{DP}$) employs historical data on reported (and repaired) defects to predict the location of previously unknown defects that lurk in the code.

Specifically, let us assume that we are developing the code for release $\mathcal{R}_n$. When working on $\mathcal{R}_n$, we are also repairing defects reported on the previous release $\mathcal{R}_{n-1}$. In the process of working on $\mathcal{R}_n$, we also, invariably, introduce defects, some of which will be discovered and reported after the official release of $\mathcal{R}_n$, and presumably, fixed during the development of $\mathcal{R}_{n+1}$. In modern development, one typically gathers not just the details of all defect repairs during $\mathcal{R}_1 \cdots \mathcal{R}_n$, but also other process and product metrics associated with files (or packages, or directories) such as complexity metrics, number of developers, number of lines changed, number of commits, etc. The task of statistical defect prediction is to learn a per-file prediction function *bugs* of the form

$$\mathit{defectcount}(f) = \mathit{bugs}(m_1(f), m_2(f), \cdots, m_n(f)),$$

where $m_1, m_2, \cdots$ are process or product metrics over the file f, as of release of $\mathcal{R}_n$ and the *predicted defect count* is the number of defects *bugs* predicts will be discovered in the file f. In some settings *bugs* is a binary, indicating only whether *some* defect is predicted to occur in f.

Defect prediction is a fairly mature field of research. Previous research indicates that process metrics [19] and organizational metrics [17] are quite effective in predicting the loci of defects. Defect prediction models have been reported to have been used at Google [13].

Evaluating Defect Prediction: $\mathcal{DP}$ tools are intended to be used to focus quality control efforts. Good prediction performance is thus vital, to ensure that limited money and time are spent effectively. Several different approaches to evaluating $\mathcal{DP}$ tool performance have been reported. IR methods such as precision, recall and accuracy are easily calculated, but are difficult to interpret in the highly class-imbalanced settings of defect prediction; a simple guesser that predicts all files to be defect-free achieves high-levels of accuracy. In response, the non-parametric AUROC (area under the receiver-operating characteristic curve, also known as AUC) has become increasingly popular. However, as noted by Arisholm & Briand [2], measures of $\mathcal{DP}$ performance should be sensitive to the *cost* of inspecting files; in particular larger files are often more costly to inspect. They proposed AUCEC (area under the cost-effectiveness curve), a "lift-chart" measure, non-parametric (like AUC), but different in that it is sensitive to the cost of inspection, essentially based on the number of lines inspected. It is by now *de rigueur* for papers on $\mathcal{DP}$ tools to report a variety of parametric, non-parametric, and cost-sensitive measures of performance. The scholarly literature on well-evaluated $\mathcal{DP}$ tools is extensive, as a simple search will reveal; we have just presented a few highlights above, for lack of space.

2.2 Static Bug Finders

Static bug finding ($\mathcal{SBF}$) arguably begins with type checking built into the compiler. We focus here on supplementary tools, which are typically targeted at specific kinds of coding errors, such as buffer overflow, race conditions, SQL injection vulnerabilities, or cross-site scripting attacks [11, 7, 28, 9]. Static analysis has typically developed opportunistically, as researchers discover new classes of defects, and seek to invent abstraction techniques and algorithms that can detect these classes of defects efficiently. Analysis tools that can detect various classes of defects, including memory allocation errors, race conditions, buffer overflows, and taint-related errors, have been reported in the literature. The verification imperative is to never falsely certify a program to be free of some class of bugs. Since bug-freeness is undecidable in general, verification techniques over-approximate program behavior to include infeasible behavior so long as no feasible behavior is lost. Bug-freeness is then proven with respect to the over-approximation. A tool is sound, *i.e.* meets the verification imperative, when this proof goes through and the over-approximation is proven not to lose any feasible behavior. Imprecision is the degree to which an over-approximation admits infeasible behavior. Sound static analysis generates false positives when it warns about infeasible behavior introduced by over-approximation. Undecidability of nontrivial program properties thus forces a Hobson's choice: sound tools have false positives, whereas unsound tools can have false negatives. And so, without foreknowledge of which warnings are false and which are true, developers must examine all the lines of code flagged in the warnings and hope to find some actual defects.

Evaluating Static Analysis Tools: The evaluation criteria here have parallels with those discussed above for defect prediction. $\mathcal{SBF}$ tools pay off when warnings lead to true positives, *viz.*, actual defects, and waste effort when warnings are false positives; they incur potential subsequent costs for missed defects that "leak" out as field defects. Formerly, papers on static analysis tools reported successful experiences with finding actual defects with their tools. When open-source developers agreed that reported warnings were actually bugs, and agreed to fix them, that was considered a success. Often the sample of test subjects is chosen by the $\mathcal{SBF}$ developers to illustrate the power of the tool in finding the specific coding errors targeted by the $\mathcal{SBF}$ tool, rather than improving the overall quality of the test subject.

Clearly, not all coding errors lead to defects that actually get exposed and reported; this issue is typically not of concern in these evaluations. In terms of CBI, their zonation is line, and their measure, developer confirmation of a warning. In addition, these papers typically describe one specific tool, and evaluate it, rather than comparing the overall power of several different tools in finding defects.

Our evaluation here is *comparative, retrospective* and centers on *reported defects*: we are asking,

> *"If developers had actually carefully inspected the lines warned by $\mathcal{SBF}$ tools, how many of the defects subsequently reported (and fixed) in the system could potentially have been discovered?".*

We also retrospectively evaluate $\mathcal{SBF}$ and $\mathcal{DP}$ on an equal footing, and explore synergies. We focus exclusively on pattern-matching static bug-finding tools for three reasons: 1) commercial static analysis tools are encumbered by licenses that prohibit their study; 2) unencumbered static analysis tools are prototypes that usually suffer from bit-rot and are difficult to acquire, build, and run; 3) in particular, running these tools retrospectively on systems with long histories is especially difficult; and 4) the fact that static analysis tools do not scale well to large systems, which would have hampered a fair comparison with prediction models, which do scale, and, in fact, are hungry for masses of data. In particular, our work focuses on JLINT, FINDBUGS and PMD.

Closely Related Work: We now briefly review prior work in evaluating $\mathcal{SBF}$ tools. Kim & Ernst [10] analyze history to determine which warnings programmers tend to fix, to help prioritize future warnings; they focused on warnings, not associated defects (if any). Wagner *et al* [27] evaluate FindBugs, PMD and QJ Pro with 4 small Java projects (3K-58K NCSL) by manually inspecting warnings to determine how many of them were true positives. Our interest is in much larger systems; we also focus on field defects, and false negatives with respect to field defects. Rutar *et al* [23] evaluate the overlap between different Java bug finding tools, without considering their relationship to reported defects. Thung *et al* [26] actually do retrospectively evaluate with field bugs with 3 widely used open-source systems; they use a case study methodology, manually examining the code to evaluate the precision and recall of static analysis tools. Nanda *et al* [18] also do a retrospective evaluation with respect to actual defects, but their retrospective evaluation is concerned primarily with null pointer defects; we consider all kinds of reported field defects. Zheng [30] evaluated the effectiveness of $\mathcal{SBF}$ tools in "live" use in an industrial testing, evaluating what kinds of errors the tools detected. Marchenko & Abrahamsson [14] report that warning counts are sometimes correlated (and sometimes anti-correlated!) with defects. Nagappan & Ball [16] report that static analysis warning density is well-correlated with *pre*release defect density; we are interested to use more typical process-metric-based prediction models, rather than static analysis tools *per se*. Ayewah [3] *et al* report the results of a survey at Google on how users and projects use FindBugs. Bessy [4] offers engaging anecdotes on the experience of running Coverity^{TM} at scale. The defect-finding rate is not reported. None of the above approaches specifically compare or look for synergies between $\mathcal{DP}$ and $\mathcal{SBF}$.

Our central experimental conceit is evaluation based on actual, reported and fixed defects; we take the position that these defects (which after all, are the ones that developers actually chose to fix) are the most important ones. While other defects may lurk undetected in the system for years, we argue that the defects that were actually reported and fixed are the ones most likely to have influenced the perceived quality of the system. So we assess the potential value of $\mathcal{DP}$ and $\mathcal{SBF}$ based on their potential to guide developers towards locating these defects as early as possible.

An immediate consequence of this conceit is that our analysis is *post facto*; as such, we have to retrospectively predicate how the $\mathcal{DP}$ and $\mathcal{SBF}$ tools *should* have been used, to locate and remove these defects *before* they were released into the wild and base our analysis on this predication. We chose a simple approach. We stipulate that:

1. The tools (either $\mathcal{DP}$ or $\mathcal{SBF}$) are run very close to release date;
2. Lines indicated with warnings by $\mathcal{SBF}$ and files predicted as defective by $\mathcal{DP}$ are carefully inspected by competent personnel; and
3. *All defects* associated with those lines are discovered during inspection.

While these simplifications are not entirely realistic, we argue that they are justifiable, for a retrospective experiment. First (step 1 above), many processes do indicate that code inspections occur fairly close to release candidate status; for simplicity, we run the analysis tools and defect prediction on the code of a released version. Second (step 2 above), the cost of inspecting the entire system make it quite reasonable to target inspections at those portions of the system that are considered highest risk. Finally (step 3), as a first step toward solving the CBI measurement problem, we assume that all defects in the inspected code are discovered, even ones unrelated to $\mathcal{SBF}$ warnings.

This *efficacy* assumption is made in published literature on the evaluation of $\mathcal{DP}$ tools, since $\mathcal{DP}$ tools indicate the likely locations of defects based on previous patterns, rather than on specific etiologies. However, efficacy is typically *not* assumed in the evaluation of $\mathcal{SBF}$ tools: a line of code that flagged as containing a race condition may not, in fact, contain a race condition, but contain instead an unrelated error. By convention, $\mathcal{SBF}$ tool researchers typically give the $\mathcal{SBF}$ tool credit only if the actual error corresponds to the reported error. To measure the two approaches on equal footing, we must adopt either $\mathcal{DP}$'s efficacy assumption or $\mathcal{SBF}$'s more stringent measure. We lack an oracle for matching warnings to bugs and are operating at a scale where manually checking warning to bug correspondence is infeasible, so we make $\mathcal{DP}$'s efficacy assumption. In other words, we assume that a careful code inspector would in fact detect any and all errors in the warned lines.

In conclusion, we make a fairness assumption that applies equally to $\mathcal{SBF}$ and $\mathcal{DP}$: if some code is flagged for inspection, regardless of the reason why, we assume that a competent and conscientious developer would detect *any* defect in that region of code.

2.3 Research Questions

Armed with our solution to CBI, which we explicate in Section 3.2, we are now ready to compare $\mathcal{SBF}$ to $\mathcal{DP}$. First, we ask how well do the two approaches perform in terms of effectively identifying code for inspection.

Research Question 1: How do static bug-finding tools compare with statistical defect prediction with partial and full credit?

Here, partial and full credit refers to our solution to the CBI measurement problem, again as described in Section 3.2. To our knowledge, we are the first to effect this comparison.

Having established a means for comparing the two approaches we now turn our attention to the search for synergies between the two approaches. First we ask;

Research Question 2: Can statistical defect prediction improve the performance of static bug-finding tools?

then, "vice versa", we ask:

Research Question 3: Can static bug-finding tools improve the performance of statistical defect prediction?

3. EXPERIMENTAL METHODOLOGY

We study five open-source projects from the Apache Software Foundation: Lucene, Derby, Wicket, OpenJPA, and Qpid^{TM}, as shown in Table 1. They range in size from 68-630K NCSL. All are Java projects. There are varying numbers of releases for each project. We studied the occurrence of bugs, and the performance $\mathcal{DP}$ and and $\mathcal{SBF}$ approaches at release level intervals. Specifically, as described above, we assess how well inspections just prior to a release $\mathcal{R}_n$, guided by $\mathcal{DP}$ and/or $\mathcal{SBF}$, would have helped developers avoid defects discovered after $\mathcal{R}_n$ is made available to users.

3.1 Data Gathered

For every project, we gathered version-control information from GIT; all projects also use the JIRA issue tracking system. From the JIRA data, we identified commits that were bug fixing. We considered any file associated with a bug fix to be buggy. We used GIT `blame` to identify the provenance of the lines where defects occurred with options for detecting changes and moves, and whitespace insensitivity. We used the SZZ [25] approach with this data to identify the source of the buggy lines. For each release of each project, we collect the warnings reported by FindBugs, PMD, and JLint. $\mathcal{SBF}$ tools have different settings that could potentially produce different sets of warnings. In practice, we found that the warnings produced FindBugs and JLint showed no variation with different settings. For PMD we used all the Java language rulesets, except for rules having to do with comments, coupling & design (we felt these were higher level than we wanted to target) and API rules that were not relevant to the systems under evaluation. While PMD, being source-based, did not require a huge effort to run, JLint and FindBugs required builds of multiple versions of our large subject systems, which required six person-months of effort to compile all the systems, run the tools over the resulting class files, and gather warnings.

Perhaps because of the need to map warnings from class files to source files, we found that JLint warnings had erroneous line numbers. Upon inspection of a sizable randomly chosen sample, we found too many warnings pointed into

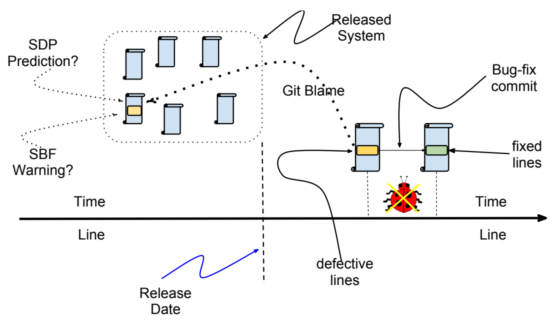

Figure 1: The timeline shows a system version release date (dashed vertical line) with released versions of all files in a dashed box. For every bug that is fixed, post-release, we use `git blame` to identify the lines <u>in the released version</u> of the file; we then examine whether $\mathcal{DP}$ predictions or $\mathcal{SBF}$ warnings would have indicated, <u>at release time</u>, that those lines should be inspected.

commented regions of code, specifically in the license disclosure region in the first several lines. Because of poor data quality issues, and because our analysis is at the line-level, we *discarded* JLint *from further consideration.*

We gathered a wide range of process and product metrics and used them all for $\mathcal{DP}$. As reported earlier [2] the precise learning algorithm is not vital; for simplicity, we just used logistic regression with this collection of metrics to predict defect occurrence likelihood. Since the objective here is to maximize prediction performance, we were not concerned with issues, such as multi-collinearity, and just used all available process and product metrics at the prediction problem. For process metrics, we used a large collection, including number of committers, number of changes, number of minor committers, experience of owners, *etc.*; for product metrics, we used a large collection of complexity, size, and object-oriented metrics. We have described and used this set of metrics in earlier work [20, 19, 21].

To classify files as defective or clean, we use a logistic regression classifier with these metrics as its input. The logistic regression classifier uses both code metrics and process metrics as gathered from the JIRA issue tracking system. To consider a file truly defective, there must be some bug-fixing commit that includes that file. We train our logistic regression-based prediction models on the k-th release of a project, and we test the model on any successive release.

Test Data: We study five open-source projects from the Apache Software Foundation: Lucene, Derby, Wicket, OpenJPA, and Qpid^{TM}. For each release of each project, we collect the warnings reported by FindBugs, PMD, and JLint.

All three tools produce warnings on the functionality of code. JLint is the only one not to report on style. However, FindBugs and PMD warn on style. For example, FindBugs has a warning category specifically for style, and PMD warns on empty code. While PMD and JLint report warnings at a line-level granularity, FindBugs operates differently. FindBugs reports warnings using a combination of line-level, method-level, and class-level granularity. For this reason, FindBugs reports significantly more lines, but far fewer warnings, as compared to PMD and JLint.

Defect Data: For our test to scale, we need to map defects to lines so we can then ask when $\mathcal{SBF}$ warned lines or files

Table 1: Summary data on projects, ranges of values are shown in columns when applicable, from smallest to largest. Extremal values may occur in different releases for different covariates.

Project	Releases	Files	NCSL	FindBugs Warnings	PMD Warnings	Defects	FindBugs		PMD	
							Rec.	Prec.	Rec.	Prec.
Lucene	7	0.5–1.4K	68–178K	137–300	12–31K	24–83	0.036	0.022	0.17	0.015
Qpid	7	2.3–3.3K	212–342K	32–66	69–80K	74–127	0.00061	0.0017	0.10	0.0056
Wicket	5	2.1–2.7K	138–178K	45–86	23–30K	47–194	0.0070	0.017	0.10	0.0080
Derby	7	2–2.9K	420–630K	1527–1688	140–192K	89–147	0.067	0.0043	0.17	0.0041
OpenJPA	8	1.2–4.7K	152–454K	51–340	62–171K	36–104	0.0034	0.0038	0.26	0.0024

identified by $\mathcal{DP}$ intersect those lines. Figure 1 shows our solution to this problem. First, we identify (using JIRA) all the big-fixing commits that occurred subsequent to the release date (note commit marked with X-ed out bug). The lines that were changed in these commits are considered defective lines. We use `git blame` to identify the provenance of the defective lines. Now any blamed lines that were present in the released system, shown in the Figure 1, could potentially have been identified for inspection by $\mathcal{SBF}$ warnings, or by an $\mathcal{DP}$ prediction. Blamed lines that are actually identified with $\mathcal{SBF}$ warnings or $\mathcal{DP}$ predictions are considered true positives (or "hits", to use IR terminology).

3.2 Solving the CBI Problem

In general, we measure the value of both methods on an equal footing, using area under the cost-effectiveness curve, AUCEC [2]. This measurement is the area under a lift-chart, x-axis being proportion of SLOC, and y-axis the proportion of defects. This is a cost-sensitive and, unlike precision and recall, non-parametric measure. For these reasons, it has become quite popular recently. However, several complications arise when applying these measures in our setting, which we discuss below, along with our solution to the CBI problem.

Zonation: The fact that $\mathcal{DP}$ predictions are file-granular, while $\mathcal{SBF}$ warns at a line level makes their direct comparison challenging. A fair comparison dictates that we compare the performance of the two approaches on the same number of lines. To solve this problem, we introduce some terminology. Each warning a $\mathcal{SBF}$ tool emits warns a possibly empty set of lines. For each project, we sum all of the unique, warned lines a $\mathcal{SBF}$ tool emits. This sum is our "inspection budget" for that project. To measure the "hit" rate of the warned lines, we inspect the emitting tool's native priority order (high priority warnings in smaller files first, in line order within files) and calculate the AUCEC value. We call this value AUCECL (AUCEC for warning **L**ines).

We then train logistic regression normally, using all available data but restrict the files for which we measure its success at defect prediction to be only over a set of files whose line sum is within a small error tolerance of the inspection budget. We then ask: "What is logistic regression's pay-off given this inspection budget?". Now the question becomes choosing on which subset of a project's files to "spend" this inspection budget, based on $\mathcal{DP}$ predictions. First, we order all the files by defect likelihood (as predicted by $\mathcal{DP}$), then select as many files as we can given the inspection budget. Most of the time, we waste some of the inspection budget because it is insufficient to allow us to "buy" the next file; sometimes, this means we return no file. The AUCEC measure over this collection of files can then be compared with the AUCECL above on an equal footing.

One could argue that this approach is *unfair to* $\mathcal{DP}$; sometimes $\mathcal{SBF}$ tools warn on as little as 0.4% of the SLOC in a project; with such a limited line budget, a candidate $\mathcal{DP}$ might only be allowed to recommend a handful of files. On the other hand, one could argue that, when inspecting $\mathcal{SBF}$ warnings, developers *rarely* look at *just* the warned lines and examine surrounding code, with the result that $\mathcal{SBF}$'s budget is unfairly low. We acknowledge these arguments, but claim our procedure defines a reasonable baseline for a long-overdue comparison; we hope that our dataset will enable other types of future analysis of the CBI problem.

IMPORTANT AUCEC and AUCECL are the same measure, calculated the same way (proportional pay off in defects for proportion of lines inspected). We use the different spellings to remind the reader of the zonation issue at play here, between the file-level for $\mathcal{DP}$ and the line-level for $\mathcal{SBF}$.

CBI Measurement: Defective code sometimes is a few consecutive lines in a file; sometimes defective code is widely scattered. What we have access to in the process metadata is the bug-fixing commit and the lines changed to repair the defect. Current best-practice in the mining community is the celebrated "SZZ" [25] approach, which flags the lines changed in the bug-fix commit as defective lines. We adopt this approach, while acknowledging its imperfections, and consider this code to be defective in our analysis.

The *overlap*, intersection, of defective code with either a file flagged by $\mathcal{DP}$, or lines warned by $\mathcal{SBF}$ may be partial or complete. Consider a null-pointer warning by an $\mathcal{SBF}$ tool, say FindBugs. Assume there are m lines warned for this null-pointer a file f in release r. Let's assume that this same file f after the same release r has a bug fix, (that changes n lines in file f) between release r and $r+1$. Assume further that l lines overlap between the m warning lines and the n defective lines. The CBI measurement problem arises here: "How much credit should be given to FindBugs?"

An optimistic view, the "optimistic credit" or $Credit_{\mathcal{F}}$ view, is that, if even a *single* warning line overlaps with a defective line associated with a bug fix (*i.e.* there is a nonempty intersection), there is a strong possibility that bug would have been noticed during inspection and caught before release. This view suggests that with a single line of overlap, the warning tool should be given full credit for the bug. A less optimistic view, "scaled, or partial, credit", ($Credit_{\mathcal{P}}$) is that, if $x\%$ of the defective lines associated with a bug overlap a warning tool, then the warning tool gets credit equivalent to $\frac{x}{100}$ for that bug.

These measures are not perfect. For example, one can argue that $Credit_{\mathcal{F}}$ will lower the AUCEC scores for $\mathcal{DP}$, since defective lines occur together in files; $\mathcal{DP}$ would get credit for only full defects, whereas $\mathcal{SBF}$, which can give scattered warnings at different locations in files, has a greater chance of "hitting" defective lines. One could also criticize $Credit_{\mathcal{P}}$ as lowering the scores for $\mathcal{SBF}$: as when a single warning

Table 2: Recall between various methods of finding defects. O/L refers to number of defects found in that category that overlap defects found by logistic.

Project	FindBugs		PMD		Logistic	
	Total	O/L	Total	O/L	FB	PMD
Lucene	41	11	147	87	36	129
Qpid	5	0	218	113	5	167
Wicket	10	0	160	40	0	82
Derby	171	78	461	319	251	476
OpenJPA	8	0	321	264	14	383

on one line (*e.g.*, a failure to check a return value) generates 10 bug-fix lines, thus giving the warning only 0.1 credit for that bug. This seems unfairly low, if the warning on that one line was sufficient to incite the programmer to fix that bug. In this work, we compare $\mathcal{DP}$ and $\mathcal{SBF}$ using both the optimistic and scaled approaches, to give two different perspectives on their relative effectiveness.

Surely, other measures could be defined, with other attendant compromises. In this work, we have highlighted the importance of the CBI problem, explicated the difficulties of devising an experiment to solve it, and present the results of our solution. We have spent a great deal of effort constructing the dataset; we hope that follow-on work will leverage our dataset to more readily tackle CBI, bringing new measures to bear.

4. RESULTS

We now present the results of our comparative study of two $\mathcal{SBF}$ tools, PMD and FINDBUGS, and $\mathcal{DP}$ based on a logistic model.

Table 1 summarizes our 5 projects. The number of releases vary in each project, from a low of 5 for Wicket to 8 for OpenJPA. We generally discarded the last (current during our analysis) release, because defect data is incomplete, and subject to right-censorship. The systems are of moderate size, ranging from 68K lines to 630K lines for Derby. The file count varies from The defect counts per release range from 24 (Lucene) to almost 200 (Wicket).

Once we gathered the warnings, we removed comment lines from our partial- and full-credit calculations. The remaining lines that are warned in each release are shown Table 1. The difference between FINDBUGS and PMD is in some cases a couple of orders of magnitude. The sparsity of warnings from FINDBUGS is particularly noteworthy in Qpid and Wicket, and is in fact troublesome for our purposes — with such a limited number of warned lines, our approach of selecting files from $\mathcal{DP}$ to meet this budget is severely constrained, and thus we might reasonably expect very poor performance, as we shall see.

The recall and precision numbers are shown in the last column. These numbers are cumulative over all releases; we simply count the total number of defects and see how many are indicated in the warning lines using $\mathit{Credit}_{\mathcal{F}}$. While the recall numbers and precision numbers are very low, this is not unexpected. In particular, note that the recall numbers are as a proportion of actual field defects: it is worthwhile to avoid even a single field defect, so finding any at all is a good thing. The low precision numbers are more troublesome — this indicates that the vast majority of the warnings (as many as 99.5 in some cases) are perhaps false positives that have no bearing on defects reported after release. As pointed out by Zhang & Cheung [29], however, sometimes avoinding the high cost of defects might compensate for the cost of inspecting a large number of false positives.

In Table 2, we show the defect counts actually found by FINDBUGS (*e.g.*, 41 total across all releases in Lucene) and the defects found by Logistic regression $\mathcal{DP}$ using the number of lines warned by FINDBUGS (a total of 36) and number of lines warned by PMD (total of 129). This table also shows the overlap between FINDBUGS defects and those found by logistic $\mathcal{DP}$ (11 of 41 defects) and the overlap between PMD-found defects and logistic $\mathcal{DP}$ defects (86 defects). It's noteworthy that the overlap with $\mathcal{DP}$ is generally higher for FINDBUGS than for PMD, thus suggesting that FINDBUGS and logistic $\mathcal{DP}$ are more complementary than PMD and logistic $\mathcal{DP}$.

Certainly, $\mathcal{DP}$ also produce a lot of false alarms, suggesting files for inspection that have no defects. We compare the two next.

4.1 Inspection Cost Comparison

Figure 2 compares the performance of the FINDBUGS tool against performance of logistic $\mathcal{DP}$, for all 5 of our test projects, over all releases. Each plot point represents the performance of a given approach, for a given project for a given release. The lines are not really meaningful or indicative of any real trends, and are just shown for clarity and aesthetics. The performance here is measured using AUCEC (for logic $\mathcal{DP}$) and AUCECL (for FINDBUGS). As explained in Section 3.2, the measures are fair: they allow the two approaches an equal inspection budget, and measure their effectiveness in capturing field defects. We use the different names to emphasize that they arise from different tool granularities. This plot is made using full credit, $\mathit{Credit}_{\mathcal{F}}$ as discussed in Section 3.2: for defects that require multiple line changes, even a single line overlapping with an inspected line or file is considered a "hit". The random is the AUCECL value (in expectation) of choosing lines uniformly at random, and assume that defects are uniformly at random associated with the lines. In the case of Derby, $\mathcal{DP}$ outperforms FINDBUGS, while in all the other, FINDBUGS generally does better, except for a couple of early releases in Lucene, release 5 in Qpid, and release 2 in OpenJPA. All in all, in 21 (out of 29) releases across 5 projects, FINDBUGS outperforms logistic $\mathcal{DP}$. In the case of PMD, the situation is almost reversed (Figure 3: logistic $\mathcal{DP}$ dominates uniformly in Qpid, Derby, and OpenJPA, in half the releases in Lucene, and uniformly loses in Wicket. All in all, logistic $\mathcal{DP}$ dominates in 19 out of 29 releases.

In the case of Partial Credit accounting (not shown for reasons of space) for PMD, Logistic $\mathcal{DP}$ almost uniformly dominates PMD in 27 out of the 29 releases, except for the first and last release in Qpid. Turning to FINDBUGS for Partial Credit accounting, logistic $\mathcal{DP}$ again dominates for Derby, and dominates FINDBUGS for two releases each in OpenJPA and Lucene, and one release in Qpid, overall doing better in only 11 releases out of 29.

For 3 of the projects, Wicket, Qpid, and OpenJPA, only a very small portion of the system is selected for inspection by the warning tools (and thus the same line count by $\mathcal{DP}$) so the performance of both is very low. In some cases a few defects are "hit" by both tools, but there is no noticeable consistent difference between the approaches. The core observation here is that there is no significant discernible differ-

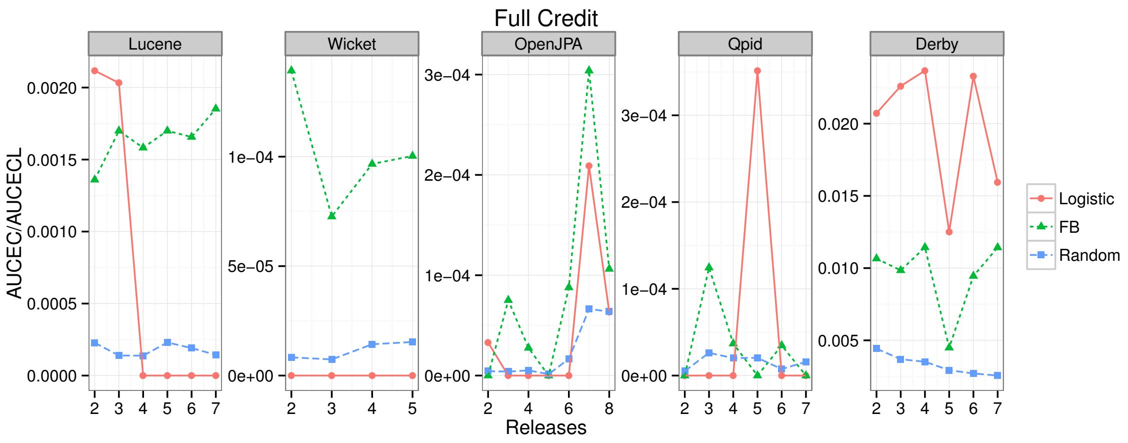

Figure 2: Comparing $\mathcal{SBF}$ (FindBugs) and $\mathcal{DP}$ (logistic regression) prediction using AUCEC as a performance measure for $\mathcal{DP}$ and AUCECL for warnings. In general the performance is not significantly different in our dataset. The two measures are calculated in a commensurate manner, using $Credit_{\mathcal{F}}$ (See Section 3.2).

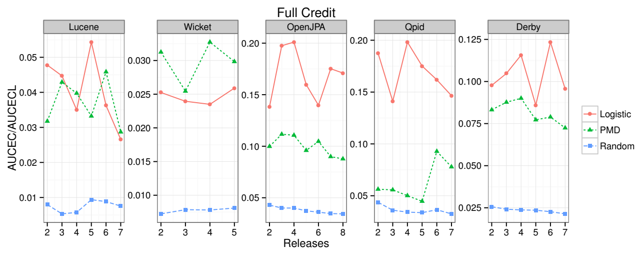

Figure 3: Isomorphic to Figure 2, but for PMD.

ence in our dataset between the performance of FindBugs and logistic $\mathcal{DP}$

Figure 3 shows the same data as Figure 2, but for PMD, again, we can observe that there is no discernible consistent difference either way. The above plots are calculated using the full credit accounting methods, $Credit_{\mathcal{F}}$. We have also generated plots using partial credit method, $Credit_{\mathcal{P}}$; however, we see the same lack of a clear, consistent difference between the two approaches.

This is a rather unexpected finding. As noted above, $\mathcal{SBF}$ tools such as FindBugs and PMD operate at a line-level granularity, and only require specific lines that are warned to be inspected. Specially given partial credit accounting, one might reasonably expect that $\mathcal{DP}$ tools would have the benefit, and thus show a clear advantage over $\mathcal{SBF}$. Subject to the warnings given in Section 5, this finding emphasizes the importance of comparing these two important, disparate approaches to software inspecting targeting on an equal footing. It also suggests licensing regimes imposed by certain very successful commercial vendors of $\mathcal{SBF}$ tools that inhibit the publication of such evaluation results are an unjustifiable barrier to the development of more effective software inspection practices; certainly, Vendors such as GrammaTech, who actively support such evaluations, are to be commended.

If this result holds up on replication, it's rather cheering. *$\mathcal{DP}$ tools work well with meta data, and do not require build integration!* With $\mathcal{DP}$ tools, we do not need to get $\mathcal{SBF}$ tools for each programming language, and integrate with build procedures. The barriers to large-scale use of $\mathcal{SBF}$ tools are documented in Bessey *et al* [4]; however $\mathcal{SBF}$ tools that analyze byte codes do not require separate build integration, and can be easier to use. In fact, many organizations, because of process maturity imperatives, have engaged in substantial metrics-gathering, and have good databases of bug reports and version control repositories; in such settings, $\mathcal{DP}$ can be readily utilized, without need for expensive software licenses or patent royalties: in fact, we just used the open-source R system in our work.

4.2 Enhancing $\mathcal{DP}$ with $\mathcal{SBF}$

We now turn to Research Question 2: can the performance of $\mathcal{DP}$ be improved by using static analysis results? (See Figure 4.) There is a simple rationale for this pursuit: Prior work on $\mathcal{DP}$ predictions has indicated that process metrics work really well [19], and in general beat measure of properties of the product. Generally speaking product metrics, which measure various properties of code such as coupling, inheritance etc. are strongly correlated with size (the larger a module, the more it is coupled, *etc.*), have been found by others [6] and us [19]. In a sense, one can view static analysis warnings as assessing a novel kind of property of source code, perhaps one more strongly allied with defects.

All $\mathcal{SBF}$ tools are intentionally designed to locate regions of code that appear to have properties that prior experience (or theory) *directly* indicate the presence of defects. This is different than a property like coupling, which is thought to cause defects *indirectly*, perhaps mediated by HCI phenomena such as the difficulty of code comprehension. Indeed prior work by Nagappan & Ball [16] suggests that there is a direct, positive correlation between static analysis warning density and defect density (although Marchenko & Abrahamsson [14] found a *negative* correlation in some cases). In conclusion, motivated by prior research and experience, and our available data, we sought to improve the performance of $\mathcal{DP}$, using metrics based static analysis warnings.

Our metrics are quite simple: we simply followed the successful experience of [16], and added warning counts from both PMD and FindBugs as an additional metrics into logistic $\mathcal{DP}$. Whether the correlation or earnings with defects was positive, or negative (!) logistic regression would discover the relationship, and train a model that can exploit this information. The results can be seen in Figure 4, which uses the same general scheme as the earlier figures; so it is compacted to save space. The y-axis shows the performance measure using the AUCEC measure. The per-

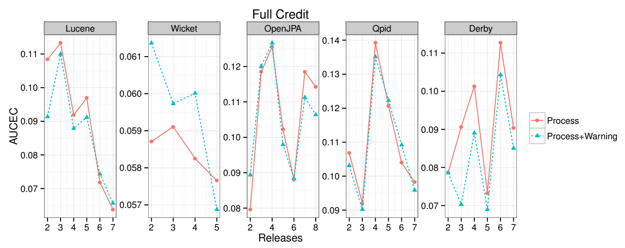

Figure 4: The effect of including $\mathcal{SBF}$ warnings as predictive metrics for $\mathcal{DP}$ for 20% NCSL inspection budget. There appears to be no discernible trend. This figure is isomorphic to Figure 2, hence its compact form here.

formance of "baseline" process-metrics based $\mathcal{DP}$ (logistic regression in this case) are shown in red. The blue line shows an "enhanced" $\mathcal{DP}$ exploiting a combination of the traditional process metrics and the warnings counts from PMD and FindBugs as an additional metric. The specific plot here is computed using the Optimistic Credit accounting method, $Credit_{\mathcal{F}}$. However, accounting under scaled credit gives similar results. The basic conclusion that can be drawn here is that there is no clear evidence that warning counts improve the performance of logistic $\mathcal{DP}$. Sometimes the enhanced $\mathcal{DP}$ performs better, sometimes the baseline performs better; this phenomenon does not change under $Credit_{\mathcal{P}}$ accounting.

Other approaches to synergizing $\mathcal{DP}$ and $\mathcal{SBF}$ may provide better performance. For example traditional product metrics thus far have only taken limited advantage of semantic properties of code, such as control flow (McCabe metrics) or data flow (certain cohesion metrics). We believe that more complex properties, such as the cardinality of points-to sets or a count of the number of times widening was applied during an abstract interpretation, that relate semantic properties to potential difficulties in human comprehension or maintainability might yield better results.

4.3 Enhancing $\mathcal{SBF}$ with $\mathcal{DP}$

Our final research question concerns whether $\mathcal{DP}$ can improve the performance of $\mathcal{SBF}$ tools. The intuition here arises from the abundant evidence that human/process factors (such as ownership, organizational structure, and geographical distribution [15, 17, 22]) influence quality. Thus, it is entirely possible that static bug finders could benefit from paying attention to such factors when prioritizing warnings for developers. As an example, warnings on code recently produced by an inexperienced programmer who is changing code that he is unfamiliar with are probably likely to be associated with defects. By contrast, warnings produced on mature code written by one of the initial creators of a system, that has remained unchanged for several releases, are unlikely to be of any great concern. These phenomena are very well accounted for in the typical process metrics used in the $\mathcal{DP}$ community in recent years to predict defective *files*, with very good reported results. There is a zonation problem here: $\mathcal{DP}$ prioritizes *files* for inspection; how is to be used for prioritizing *line-level* warnings?

Our approach to enhancing $\mathcal{SBF}$ with $\mathcal{DP}$, was *to order the warnings by the $\mathcal{DP}$-predicted probability of the files within which the warned lines occur.* This gives us a particular ordering of the warned lines. Using both our optimistic and scaled accounting, we can scan these lines in this order, assigning credit for any defective lines that are encountered. This gives us an AUCECL score for this enhanced $\mathcal{SBF}$ warning order. Now what should we compare this with? Clearly, one good candidate the the AUCECL score of a standard, priority-based ordering that is produced by FindBugs, that we use in the first research question above. However, there are many possible based orderings of defects; in fact, programmers may have different preferred orders on how they choose to inspect the warnings. It would be interesting to get a *robust estimate* of how well our ordering enhanced by $\mathcal{DP}$, and the "native" ordering produced by the built-in prioritization of the tool compares with the entire population of such orderings.

For this purpose, we generate 100 random re-orderings of the warnings produced by PMD and FindBugs, and for each one, we calculate the AUCECL. This gives an empirical distribution over a large sample of orderings, which then allows us to estimate how well the $\mathcal{DP}$-enhanced ordering compares with the overall population of orderings. Figure 5 shows the results under partial credit accounting. There are separate panels for each projects, and each is broken up by release. For each release, we show a box plot of the AUCECL values for 100 random orderings, a blue triangle for the native priority warning, and a red dot for the logistic (p-val, or predicted probability of defects value) based ordering. The lines are shown for visual clarity, and have no semantics. Upper row is FindBugs, and lower row is PMD. Under this accounting, (after correcting the p-values for false discovery using Benjami-Hochberg procedure) the logistic (p-val) ordering dominates ($p < 0.01$) in 25 of the 29 releases for PMD (over all different projects), and 19 of the 29 releases for FindBugs. It also dominates the native priority ordering most cases. In fact, logistic-based ordering dominates in virtually every release of virtually every project except for Wicket, just because wicket has *very few* warning lines, thus constraining the line-budget for $\mathcal{DP}$. Thus yielding a simple lesson: under partial accounting, *b*est to order the warnings using logistic $\mathcal{DP}$ predicted probability of defects!

Under full accounting (which we omit for space reasons), p-val based ordering doesn't fare well at all. It significantly beats the random orderings ($p < 0.01$) after correction only in 5 releases (2 in OpenJPA and 4 in Qpid). As explained earlier, this is not surprising—when bugs span multiple lines, all lines tend to occur together in a file, and scattershot $\mathcal{SBF}$ warnings that happen to hit even a single one of those lines will get full credit for that bug; whereas $\mathcal{DP}$ will select full files and get only a single credit for a multi-line bug only after the entire set of lines for that file is accounted for.

Finally, there is a very interesting observation: *the logistic based ordering frequently outperforms the native priority tool-based ordering, across project and release, for both tools.* In the case of PMD only for 2 out of the 29 releases under Full Credit (1 for the partial) the native ordering is better than logistic ordering; for FindBugs, the native ordering dominates only 8 cases under partial credit (7 full). A two-sample Wilcoxon test rejects the null hypothesis (with alternative hypothesis set to *logistic* > *native*) after correction ($p < 0.01$) in all cases, except for partial credit under FindBugs ($p = 0.02$). This suggests that one could prefer logistic-based ordering to the native ordering. However, one should interpret the p-values prudently; an abundance of caution suggests that releases are not necessarily indepen-

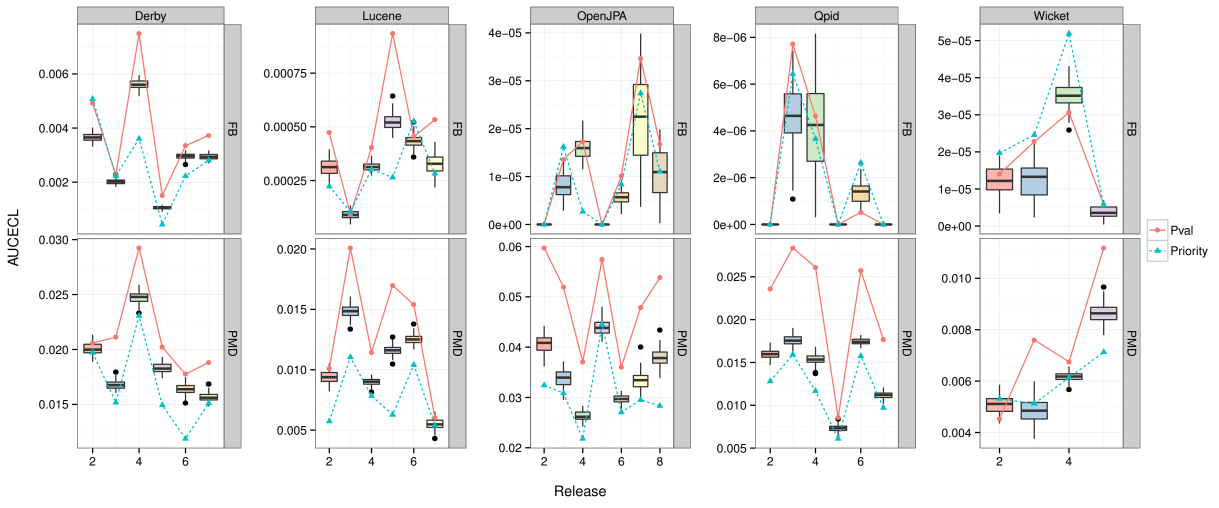

Figure 5: The effect of ordering *SBF* warnings using *DP* predictions, compared to 100 random orderings of *SBF* warnings. The boxplot shows the 100 orderings. The red line (with circles) is warnings ordered by logistic *DP* prediction of defect probability (p-val) and the blue triangles are the native priority ordering. *Partial credit* scoring is used.

dent samples.

5. THREATS TO VALIDITY

If developers were in fact using *SBF* tools during development, then it is possible that warnings (and perhaps associated bugs) were fixed before release, and thus fewer post-release defects would be associated with warnings. In Lucene and Wicket, we found no evidence of systematic *SBF* use. In the case of Derby, a developer list message reported that contributors from Oracle may have been using FindBugs. In the case of Qpid, we found that a FindBugs task was added Sept 17, 2010, and in OpenJPA, on Jun 14, 2010. However, when we examined the history of warning counts in Derby, Wicket, and OpenJPA, we found no significant evidence of warning repair; nor did we find any significant preferential reduction in high-priority warnings, or ones more ominous-sounding (*e.g.,* MALICIOUS_CODE or CORRECTNESS being). In addition, we found no evidence in the email archives of any these projects suggesting a systematic adoption of *SBF* tools. These observations mitigate this particular threat to our findings.

Our work may *not be generalizable.* We have chosen 5 distinct projects of different sizes and application domains. We However, all are Java-based. Our two tools, PMD and FINDBUGS are also therefore Java based. Both are heuristic bug-finders, although FINDBUGS does employ some static analysis. FINDBUGS requires compiled Java code, and thus entailed tremendous effort, to compile our 5 systems; older versions presented special challenges, such as finding older Java SDK versions. PMD was relatively easier, as it works on source code. Ideally, this experiment should be repeated for more projects in more different languages, with other tools. In this case, since we making the defect data publicly available, it may help other researchers try other Java-based *SBF* tools. Although we use one *DP* method (logistic regression) using primarily process metrics, prior reports [1, 19] suggest that this approach a) would be difficult to beat and b) easy to repeat in a new setting. Our *measures may be mis-targeted.* As explained, we report performance under the full credit and partial credit assumptions. As discussed in Section 3.2, there are arguments against these measures, and others could be defined. In addition, AUCEC *per se* has been criticized for ignoring the cost of false negatives (missed defects [29]). We hope that our making this hard-won warnings & defect data available will encourage repetition of this trial with other measures.

6. CONCLUSION AND FUTURE WORK

Defect prediction and bug finders target the same problem: selecting a subset of source code on which to expend limited quality control budgets. We are the first to compare their performance. We address the *comparability of bugginess identification* problem, whose tackling cuts to the core of software engineering: knowing when one outperforms the other optimizes resource allocation and promises to guide the search for useful synergies in service of software engineering's core aim to produce better cheaper software faster.

Our comparison is based on the AUCEC cost-benefit metric. We find that statistical defect prediction appears to do better than PMD, a widely used tool, under both partial and full credit accounting in most cases. However, *DP* does not fare as well against FINDBUGS, generally doing worse until full credit accounting, and not as badly under partial credit accounting. Second, we find that using *SBF* warnings as an additional metric does not significantly improve statistical prediction. Last but not least, we find that ordering *SBF* warnings based on *DP* appears to improve upon the native *SBF* priority levels in a majority of cases. While this result appears significant on a two-sample statistical test, we urge caution, since releases are not necessarily independent samples. Comparisons such as these are key to promoting engineering discipline in the selection of quality control techniques, and we invite others use our dataset for further experiments. This material is based upon work supported by the National Science Fondation under Grant No. 0974703.

References

[1] E. Arisholm, L. C. Briand, and M. Fuglerud. Data mining techniques for building fault-proneness models

in telecom java software. In *ISSRE*, pages 215–224. IEEE Computer Society, 2007.

[2] E. Arisholm, L. C. Briand, and E. B. Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *JSS*, 83(1):2–17, 2010.

[3] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.

[4] A. Bessey, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, S. McPeak, and D. Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010.

[5] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[6] K. El Emam, S. Benlarbi, N. Goel, and S. Rai. The confounding effect of class size on the validity of object-oriented metrics. *TSE*, 27(7):630–650, 2001.

[7] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. In *SOSP*, volume 37, pages 237–252. ACM, 2003.

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for java. In *ACM Sigplan Notices*, volume 37, pages 234–245. ACM, 2002.

[9] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *SP*, pages 6–pp. IEEE, 2006.

[10] S. Kim and M. D. Ernst. Which warnings should i fix first? In *FSE*, pages 45–54. ACM, 2007.

[11] D. Larochelle and D. Evans. Statically detecting likely buffer overflow vulnerabilities. In *USENIX Security Symposium*, pages 177–190. Washington DC, 2001.

[12] S. Lessmann, B. Baesens, C. Mues, and S. Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *TSE*, 34(4):485–496, July 2008.

[13] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *ICSE*, pages 372–381. IEEE Press, 2013.

[14] A. Marchenko and P. Abrahamsson. Predicting software defect density: a case study on automated static code analysis. In *Agile Processes in Software Engineering and Extreme Programming*, pages 137–140. Springer, 2007.

[15] A. Meneely and L. A. Williams. Secure open source collaboration: an empirical study of linus' law. In E. Al-Shaer, S. Jha, and A. D. Keromytis, editors, *CCS*, pages 453–462. ACM, 2009.

[16] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *ICSE*, pages 580–586. ACM, 2005.

[17] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *ICSE*, pages 521–530. ACM, 2008.

[18] M. G. Nanda, M. Gupta, S. Sinha, S. Chandra, D. Schmidt, and P. Balachandran. Making defect-finding tools work for you. In *ICSE*, pages 99–108. ACM, 2010.

[19] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *ICSE*, pages 432–441. IEEE Press, 2013.

[20] F. Rahman, D. Posnett, and P. Devanbu. Recalling the "imprecision" of cross-project defect prediction. In *FSE*. ACM, 2012.

[21] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *FSE*, 2013.

[22] N. Ramasubbu, M. Cataldo, R. K. Balan, and J. D. Herbsleb. Configuring global software teams: a multi-company analysis of project productivity, quality, and profits. In *ICSE*, pages 261–270. ACM, 2011.

[23] N. Rutar, C. B. Almazan, and J. S. Foster. A comparison of bug finding tools for java. In *ISSRE*, pages 245–256. IEEE, 2004.

[24] A. September. IEEE standard glossary of software engineering terminology, 1990.

[25] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? In *ACM sigsoft software engineering notes*, volume 30, pages 1–5. ACM, 2005.

[26] F. Thung, D. Lo, L. Jiang, F. Rahman, P. T. Devanbu, et al. To what extent could we detect field defects? an empirical study of false negatives in static bug finding tools. In *ASE*, pages 50–59. ACM, 2012.

[27] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, pages 40–55. Springer, 2005.

[28] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[29] H. Zhang and S. Cheung. A cost-effectiveness criterion for applying software defect prediction models. In *FSE*, pages 643–646. ACM, 2013.

[30] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *TSE*, 32(4):240–253, 2006.