Does Bug Prediction Support Human Developers? Findings From a Google Case Study

Chris Lewis[1], Zhongpeng Lin[1], Caitlin Sadowski[2], Xiaoyan Zhu[3], Rong Ou[2], E. James Whitehead Jr.[1]
[1]University of California, Santa Cruz, [2]Google Inc., [3]Xi'an Jiaotong University
Email: {cflewis,linzhp,ejw}@soe.ucsc.edu, {supertri,rongou}@google.com, xyxyzh@gmail.com

***Abstract*—While many bug prediction algorithms have been developed by academia, they're often only tested and verified in the lab using automated means. We do not have a strong idea about whether such algorithms are useful to guide human developers. We deployed a bug prediction algorithm across Google, and found no identifiable change in developer behavior. Using our experience, we provide several characteristics that bug prediction algorithms need to meet in order to be accepted by human developers and truly change how developers evaluate their code.**

I. Introduction

The growth of empirical software engineering techniques has led to increased interest in bug prediction algorithms. These algorithms *predict* areas of software projects that are likely to be *bug-prone*: areas where there tend to be a high incidence of bugs appearing (these are also sometimes called *fault-prone* areas [1]). The overall approach is statistical: bug prediction algorithms are based on aspects of how the code was developed and various metrics that the code exhibits, rather than traditional static or dynamic analysis approaches. One common strategy is to use code metrics [2], [3], while another has focused on extracting features from a source file's change history, relying on the intuition that bugs tend to cluster around difficult pieces of code [4], [5].

Evaluations of these algorithms have confirmed that they can successfully predict bug-prone code areas. Such evaluations have typically involved the use of the information retrieval and machine learning metrics of accuracy, precision, recall, F-measure, and ROC AUC; the underlying rationale has been that high values for these metrics indicate a well-performing algorithm. The follow-on assumption is that a well-performing bug prediction algorithm provides useful information to developers. Since this is a foundational assumption for bug prediction, one would expect that it has been well tested by multiple empirical studies of the use of bug prediction in software development settings. It has not.

There is very little empirical data validating that areas predicted to be bug-prone match the expectations of expert developers, nor is there data showing whether the information provided by bug prediction algorithms leads to modification of developer behavior. In this paper, we focus on predictions at the granularity of files. The goal of this paper is to investigate how developers respond to bug prediction algorithms via the following research questions:

RQ 1 According to expert opinion, given a collection of bug prediction algorithms, how many bug-prone files do they find and which algorithm is preferred?

RQ 2 What are the desirable characteristics a bug prediction algorithm should have?

RQ 3 Using the knowledge gained from the other two questions to design a likely algorithm, do developers modify their behavior when presented with bug prediction results?

To answer these questions, we partnered with Google's Engineering Tools department to evaluate various bug prediction algorithms and develop one for use at Google. We address RQ 1 using formal user studies, RQ 2 through informal discussion with developers, and RQ 3 through quantitative data collected from Google's code review system. We find that developers prefer bug prediction algorithms that expose files with large numbers of closed bugs and present a number of desirable characteristics that a bug prediction algorithm should have. We also find that there was no significant change in developer behavior after our deployment of a bug prediction algorithm.

II. Bug Prediction

A. Algorithm Choice

One of the most popular academic bug prediction algorithms at the time of writing is FixCache [1], which won an ACM SIGSOFT Distinguished Paper Award at the International Conference of Software Engineering 2007. FixCache is an algorithm which uses the idea of bug "locality": when a bug is found, it is likely there will be more in that area of the code base. Using various localities, FixCache creates a "cache" of files that are predicted to be bug-prone at a particular commit. When a file meets one of these locality criterions, it enters the cache, and an old file is replaced via a selectable policy (commonly Least Recently Used (LRU)). FixCache uses three localities: if a file is recently changed/added it is likely to contain faults (churn locality); if a file contains a fault, it is likely to contain more faults (temporal locality); and files that change alongside faulty files are more likely to contain faults (spatial locality).

FixCache's characteristics were extensively surveyed by Rahman et al. [6], investigating how it can aid human inspection. They concluded that the bug density of files in the cache is generally higher than outside the cache, and that the temporal locality feature is what mostly contributes to FixCache's effectiveness. They present findings that, when bug

prediction is used to support human inspection (such as code review), using the number of closed bugs to rank files from most bug-prone to least bug-prone works almost as well as FixCache. We call this the Rahman algorithm.

We decided that the FixCache and Rahman algorithms were ideal for our study. FixCache, as well as performing well in the lab, has been used with success in industry at Ericsson [7], [8]. The simplicity of the Rahman algorithm provides an excellent counterpoint to the more-complex FixCache, and so these two algorithms complement each other well. FixCache can be run with various parameters, which were evaluated in the original paper [1] to find optimal settings, but we found that the cache size metric would not be usable in its current form.

B. FixCache Cache Size

FixCache is found to work best when the cache size is set to store about 10% of the files in the code base. However, after consultation with Google engineers, this number was deemed too large for most Google projects. Mid-size projects can run into order thousands of source files, meaning hundreds of files will be flagged by FixCache as bug-prone. This load is too high for developers to properly process, and would lead to the majority of files being worked on at any given time being flagged. Such exposure to the flag will lessen its impact, and we believed developers would quickly learn to ignore it. To combat this, we settled on showing just the top 20 files.

Showing the top 20 files is less trivial than it sounds. FixCache has a binary understanding of bug-propensity: either files are bug-prone or they are not. There is no weighting of severity; once files are in the cache, they are not easily re-ordered to rank the most bug-prone. We identified two possible solutions:

1) **Reduce the cache size**: The first option is to reduce the cache size to only store 20 files. With such a small cache size, it is conceivable that the number of files created in a day could fill it entirely with new files. Another issue with making the cache so small is that the cache hit rate decreases dramatically, as it is highly likely that a file will not be in the cache. This is not in and of itself negative, as hit rate, as a metric, is not perfect, but this has been the primary method of evaluation of FixCache in previous literature and could indicate that this solution is destined to perform poorly.
   Reducing the cache size biases the results towards newer files, as older files are pushed out of the cache.
2) **Order the cache by some metric**: The second option is to keep the cache size at 10%, and then order the cache by a reasonable metric to ascertain bug-propensity severity. The chosen metric for this task was the *duration*, which is the measure of how many total commits the file has been in the cache for. Files that leave the cache and then re-enter it do not have their durations reset, but continue where they left off.
   Ordering the cache biases the results towards older files, as they have more time to increase their duration.

Project	Language	Source Lines of Code	Files	Interviewees
A	Java	600 000	7000	10
B	C++	1 000 000	4000	9

TABLE I: Characteristics of the investigated projects. The Source Lines of Code (SLOC) were counted using the `sloccount` tool, which ignores comments and blank lines. Both SLOC and Files have been rounded to one significant figure.

Both options match two possible intuitions for which areas of a code base are more bug-prone: reducing the cache size highlights areas of immaturity, whereas ordering the cache highlights areas which have dogged the project for a long period of time. As neither presented a strong case over the other, we chose to use both in our studies.

III. User studies

A. Overview

To evaluate RQ 1, we decided to perform user studies, in order to utilize developer knowledge in finding out whether files flagged as bug-prone by an algorithm match developer intuition. Developers, those who work on a project daily, are the authoritative source on a code base. While it is possible to produce and use metrics such as bug density, we felt that going directly to developers would help us gain a holistic view of the code base that can sometimes be missing from data mining.

To do this, we pitted the three algorithms – FixCache with a cache set to 20 (which we will refer to as Cache-20), FixCache with its output ranked by duration (Duration Cache) and Rahman – against each other. We performed developer studies with two anonymous Google projects, which we'll refer to as Project A and Project B. The characteristics of the projects can be seen in Table I. Each project has run for a number of years, and these two were chosen based on their differing sizes and language.

In order to track which files are involved in bug fixes, each algorithm requires knowledge of closed bugs to be extracted from bug tracking software. To do this, we extracted data from Google's internal bug tracker. However, Google does not mandate use of its internal tracker, and teams may use any means of bug tracking that they wish. This means projects that were not using the tracker would not be visible to these algorithms. Further, there is evidence that bug trackers are biased, and Google's is likely no different [9]. There was no other discoverable means of finding this data, so we had to rely on this source.

The bug tracker allows developers to classify tickets as bugs, feature requests, or a number of other classifications. We only used tickets which were classified as a bug.

B. Methodology

1) Interviews: We selected two projects, A and B, and set out about recruiting interviewees from each of these projects. Projects A and B were chosen to provide a spread of variables: A is in Java, B is in C++. A has fewer source lines than B, but

more files overall. Both projects are relatively mature. Maturity meant that we had a greater amount of source history to feed each algorithm.

Interviewees were recruited by emailing project leads or those interested in code health, who were then responsible for disseminating news of the study to the other developers on their project. Volunteers would contact us to say they were willing to perform in the study. We recruited 19 interviewees with varied experience levels, ranging from between one to six years of experience on the projects studied.

Each interviewee was met in private, and then shown three lists of twenty files from the project that he or she worked on, each list generated by one of the different algorithms. Each interviewee for a particular project saw the same lists, but the lists were shuffled and handed out during the interview, and the study was performed double-blind: neither the interviewee nor the interviewer knew which list was which. Interviewees were told that the lists were generated by bug prediction algorithms with varying parameters, but were given no further details. Each interview took around 30 minutes, and interviewees were instructed to not discuss the experiment with others.

During the interviews, interviewees were asked to mark each file as either a file that they did consider to be bug-prone, that they *didn't* consider to be bug-prone, that they had no strong feeling either way about (which we will refer to as ambivalence), or that they didn't have enough experience with to make a choice (which we will refer to as unknown).

Interviewees were told that "bug-prone" means that the file may or may not have bugs now, but that it has a certain set of characteristics or history that indicates that the file has a tendency to contain faults. We very specifically avoided using a term that indicated the file had bugs currently, like "buggy", as we felt many developers may feel unable to answer this question if they do not have a very strong current understanding of the file.

As the lists were not exclusive (a file could appear in different lists), interviewees were instructed to ensure they gave consistent answers across lists, although interviewees were allowed to pick whichever method of ensuring this they felt most comfortable with. Most chose to do this by filling out the first list, and then checking the other lists for duplicates. Others would mark a file and then immediately look to see if it appeared elsewhere.

Finally, each interviewee was asked to rank the entire lists themselves, from the list they think most accurately reports the bug-prone areas of the code, to the list that is the least accurate. As there were three lists, this created three classifications that a list could have: most bug-prone, middle bug-prone, and least bug-prone. Interviewees were instructed that they did not have to take into account how they had marked the files on each list, and were informed that they should look at the list holistically. For example, a file that a developer strongly feels should be included may be more important than three or four files that were marked bug-prone, but were less problematic in practice. An example result would be that the developer marked Duration Cache as the file list that is most bug-prone, Cache-20 in the middle, and Rahman as the least bug-prone list.

2) Common questions: Interviewees raised two common questions:

1) **"What is a bug?"**: The most common question asked by interviewees was what "bug" actually meant, with a significant minority requesting guidance on this issue. The interviewer responded "I want you to define the term for yourself." Self-definition more specifically reflects the operation of the bug prediction algorithms we chose: they do not give reasoning behind what a bug is or isn't, instead using closed bug tickets as a way of inducing that a bug was present in the code beforehand. Some felt that certain classes of files, such as templates or configuration files, could not even contain bugs by their own definition (although [10], a well-regarded taxonomy of bugs, does include a classification for "configuration faults").
2) **"Can I mark files I don't work on?"**: The second most common question asked by interviewees was whether they could mark files based on feelings alone, even if they had not worked on (or even seen) the given file. Many of these interviewees indicated that they had gathered from colleagues that a certain set of files was particularly difficult to work with. As previously mentioned, some classes of files were deemed unable to contain bugs at all, and some interviewees felt they were able to confidently mark not bug-prone on these files. When asked if a file can be marked without direct interaction, the interviewer responded that the interviewee should mark a file however he/she felt comfortable to do so. For the most part, the interviewee would then continue to mark the file as bug-prone or not bug-prone, rather than leave it as an unknown. This led us to theorize that many developers on a large code base gain opinions of it via developer folklore and gut instinct rather than direct interaction, and that they feel confident that these opinions are valid.

3) Classification of responses: We aimed to specifically analyze the bug-prone and not bug-prone responses. Ambivalence is treated as a non-answer: the interviewee could have marked bug-prone or not bug-prone, so it is essentially a split decision between both. We chose not to factor this answer into our results. We use the unknown classification to differentiate between knowledge of the file but no answer (ambivalence), and having no knowledge of the file at all (unknown). File classification was performed as follows:

1) For a bug-prone classification, at least 3 respondents must have indicated the file is bug-prone, and the bug-prone responses must be at least double that of the not bug-prone responses (e.g. if 4 respondents mark a file as bug-prone, at most 2 respondents can answer not bug-prone).
2) For a not bug-prone classification, the same method as bug-prone is used in reverse.

3) If the file is neither classified bug-prone nor not bug-prone, and if at least 5 respondents indicated they did not know the file, the file is classified as unknown.
4) Otherwise, the file is marked as being disagreed upon.

While this represents only one of a multitude of classification possibilities, we found this most accurately represented our instinctual interpretation of the raw survey data. A simpler classifier, such as majority wins, will select more files as unknown as few developers have a strong understanding across the entire code base. This would nullify signals given by the developers that are actually experienced enough with the files to give a bug-prone or not bug-prone answer.

C. Results

1) List quality: Figure 1 display the results of each file classified in Projects A and B.

2) Preferred list: The preferred lists in Figure 2 show how developers ranked each list against each other, from most bug-prone to least bug-prone.

3) Notes on Project A: Interviewee comments from Project A indicated that one possible reason for the Duration Cache list unknown responses is because a number of files in the list that were old enough that most team members had never worked on them. However, developers did comment that they were aware of the files' existence, they just did not need to work with them.

Two of the files marked as not bug-prone in the Rahman list (one of which appeared in the Cache-20 list) were files containing constants. These files would need to be changed alongside various feature releases, so would be included in bug-fixing changes, causing them to be flagged. The developers we interviewed felt that these files could not be buggy by definition, and marked them not bug-prone. For example, one constants file resulted in a report of 1 developer marking it bug-prone, 7 marking it not bug-prone, and 2 having no feeling either way.

Four of the files in the Cache-20 list did not garner a single response about whether the file was or was not bug-prone, only recording a small number of ambivalent responses (1–2), with the rest being given an unknown response. This caused one developer to comment that the Cache-20 list "looked random." Two of these files were pinpointed as prototype code. When developers create small prototypes to perform experiments, they are encouraged to commit these prototypes to source control, so that any useful ideas generated are not lost. However, our algorithms had no means of identifying and filtering these files, so they were pulled into the cache.

When participants discussed the different Project A lists, they would mention that the Rahman algorithm seemed to be pulling out older files, to which developers assigned terms such as "monolithic", "has many dependents" and "hard to maintain".

Project A had one file that appeared in two lists, and one file that appeared in all three lists.

4) Notes on Project B: Project B responses were largely similar to those of Project A, and Project B interviewees commented on the age of the files in the Rahman list just as Project A developers did: "these are old files that have gained a lot of cruft", "lots of people have changed these over the years", "we have to work carefully with these because of their dependents".

Again, as with Project A, neither FixCache list seemed to perform much better than the other, but they both shared a large number of unknown responses. The unknown files in the Duration Cache may have been marked as such for the same reason that Project A identified: the files were so old that developers simply didn't work with them anymore: "[The Duration Cache is] a list of old files that represents [Project A] five years ago that is not relevant to today" and "it's like [the Rahman list], just doing a worse job".

It is worth noting that we did not identify any prototype code in any list for Project B.

Project B had more files that appeared in more than one list than Project A. Eight files appeared in two lists, and three files appeared in all three lists.

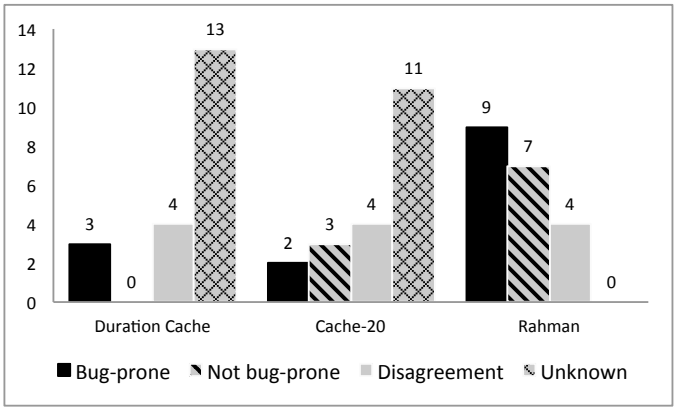
5) Interpretation: The Rahman algorithm performed significantly better than the FixCache algorithms at identifying bug-prone files. Both the Duration Cache and Cache-20 lists mostly contained unknown files. Only the Rahman algorithm resulted in a list where interviewees both knew the files on the list, and felt those files to be bug-prone. This result does not necessarily mean that the files identified in the Duration Cache and Cache-20 lists are not bug-prone, but that developers did not have enough experience with those files to make any comment.

The overall rankings show that most interviewees preferred the Rahman algorithm. This is unsurprising given the quality of the results we saw in its ability to match developer intuition at predicting bug-prone files. Comments from interviewees did not provide any clear insight into why the Duration Cache list was the middle preference in Project A, and why the Cache-20 list was the middle preference in Project B.

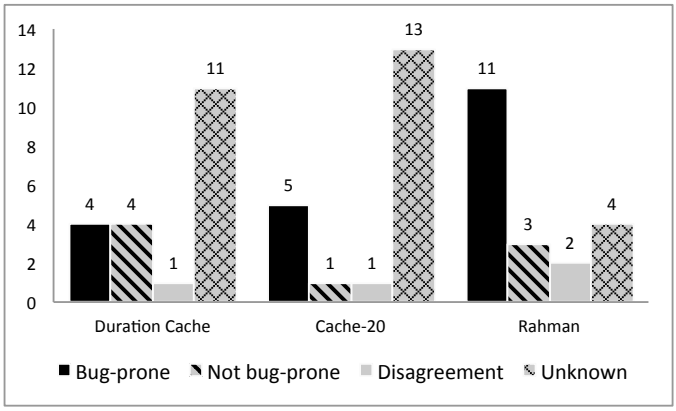
IV. Desirable Algorithm Characteristics

A. Overview

In addition to our formal user study, we had informal discussions with many developers about what they would like to see from a bug prediction algorithm, to answer RQ 2. These discussions were typical "watercooler" conversations that arose naturally in places such as on mailing lists, an internal Google+ deployment, in corridors, at lunch and on the bus. While these conversations were not formalized or coded, they are typical of the Google working environment and represent a common means of ascertaining developer feedback. Using input from these discussions, we identified three main algorithm characteristics that a bug prediction algorithm should have in order to be useful when aiding developers during code review. These three algorithm characteristics were mentioned in a majority of developer discussions, and are supported by prior research on static analysis adoption and error messages.

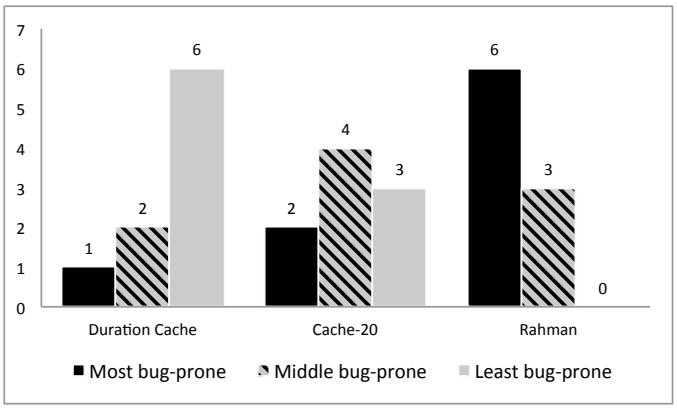


(a) Project A

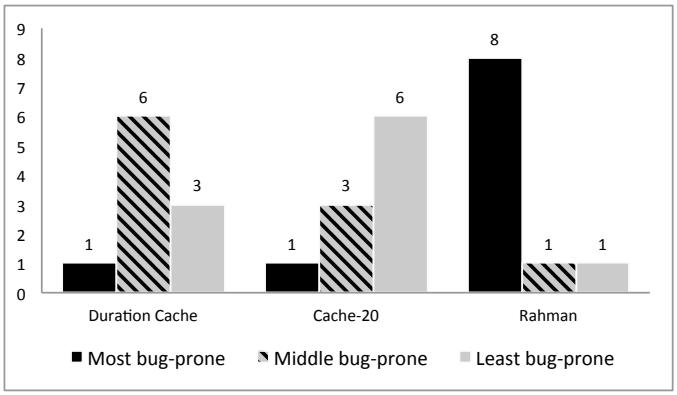


(b) Project B

Fig. 1: Charts showing the number of files of a certain classification from Project A and Project B. Files can be classified as bug-prone or not, as a disagreement between whether the file was or was not bug-prone, or the file was not known to interviewees. A higher bug-prone rating, and lower not bug-prone or disagreement ratings, is better.



(a) Project A



(b) Project B

Fig. 2: Charts showing the rankings of the lists from participants in Project A and Project B. Each bar shows the number of times the list was given a certain rank, so in Project A, one developer chose the Duration Cache as the most bug-prone, two developers chose it as the middle, and six developers chose it as the least bug-prone list. A higher most bug-prone rating and a lower least bug-prone rating is better.

We also identified two required algorithm scaling characteristics from our experiences deploying bug prediction at Google. To our knowledge, there have been no articles on bug prediction enumerating these five characteristics, although they are essential to ensure developer adoption.

B. Algorithm Characteristics

1) Actionable messages: Given the information that a code area is bug-prone, it may not be clear how to improve the software. In other words, there is a potential *gulf of execution* [11], [12] between bug prediction results and improving software artifacts based on those results. The importance of suggesting solutions to problems or errors has also long been emphasized for usability [13]. Prior research has also found that the description of a fault is important in deciding whether to fix it [14].

By far the most desired characteristic is that the output from a bug prediction algorithm is actionable. Once a bug-prone area has been identified, the development team should be able to take clear steps that will result in the area no longer being flagged. This matches developer intuition about how programming support tools – such as type-checkers or static analysis tools like FindBugs [15] – should be integrated into a workflow. For example, FindBugs is already in use at Google, and most categories of warnings cause over 90% of users to investigate high priority warnings [16].

Unfortunately, FixCache's reliance on source history means there is nothing that can be done by a team to immediately unflag a file; they must try to improve the health of the file, then wait weeks or months for it to no longer collect bug reports. This is particularly frustrating if the warning appears frequently, which is possible if a bug-prone piece of code is changed multiple times a week.

Neither Rahman nor FixCache has actionable messages,

but a related problem is that both will *always* flag a certain percentage of files, so the code can never be given a clean bill of health. This means that these algorithms never provide developers with any reward, just a revolving door of problems, which is likely very disheartening.

2) Obvious reasoning: When an area is flagged as bug-prone, there must be a strong, visible and obvious reason why the flagging took place, allaying any fear that the flag is a false positive that will only waste developer time. If a developer is convinced that a tool outputs false positives, he or she will ignore it from that point on. The burden of proof is always on the tool to precisely elucidate *why* it has come to the conclusion it has. This largely goes hand-in-hand with actionable messages: a message is only actionable if there is a clear and obvious reason why the message appeared at all. Prior research also has emphasized that programmers must develop trust for analysis tools [14], [17].

Rahman's reasoning is clear and obvious; any developer can quickly believe (and verify) the number of closed bugs on any given file. In contract, FixCache does not easily offer its reasoning; it is often opaque as to why FixCache has pulled in a file, even with a strong understanding of the algorithm and the input it has been given. Inspecting log messages will eventually allow a human to build a mental model of what happened, and presumably this indicates that a tool could be written to explain why a file was in the cache, but such a tool does not currently exist.

Obvious reasoning does not necessarily require the developer to completely agree with the assessment, only that he or she would at least be able to understand the tool's reasoning.

3) Bias towards the new: Although in the user test users noted that they believe older files to be more bug-prone, a number of developers mentioned that they aren't actually worried about the files they know have technical debt. Instead, developers were more concerned about files that are currently causing problems (which may or may not include those older files). If an algorithm doesn't have actionable messages, it should probably bias itself towards newer issues, rather than taking into account older ones (which could be a number of years old, and not indicative of the current state of the code). Prior research has identified the importance of this characteristic when using static analysis to find bugs [18]. FixCache's Least Recently Used replacement policy creates a bias towards the new, and it is a helpful coincidence that this replacement policy was also found to be the most optimal replacement policy in the original work.

C. Scaling

As a pre-requisite for deploying a bug prediction algorithm at Google, we had to ensure that the bug prediction algorithm would scale to Google-level infrastructure. We ran into two scaling issues which blocked deployment of FixCache.

1) Parallelizable: The modern architectures of large technology companies like Google, Yahoo or Amazon, now revolve around parallelized computations on large computer clusters. Ideally, we want response from a bug prediction algorithm to happen as quickly as possible. In order to meet such a goal with the large source repositories that these large companies have, the algorithm must also be able to be run in a parallelized fashion; a single CPU and data store is too slow to process an entire repository efficiently.

Rahman is easily parallelized and could run on the entire code base quickly. FixCache is less amenable to parallelization as it is not clear how one could reconcile each FixCache worker's individual cache into a single cache on the reduction step. If one was interested in just a per-project output, FixCache does run quite quickly on this smaller set of data, so multiple FixCache workers could run on separate CPU cores and complete within a number of hours.

2) Effectiveness scaling: As mentioned previously, FixCache's assumption of 10% of files is not feasible for human consumption in almost all cases. In a modest 2000 file project, 200 files will be flagged. Worse, it is likely that figure is the superset of all files being actively worked on in the project, as they are the ones that are currently being created and changed. FixCache likely flags too much in this case, and almost every file a developer submits for review will have the FixCache warning attached, decreasing its impact dramatically. However, FixCache doesn't perform as well (at least, with the currently agreed upon metrics by peer review) when the cache size is reduced. Any good bug prediction algorithm for humans must be effective whether it is to flag one file or one hundred.

In contrast, Rahman can scale to any effectiveness level and still operate similarly.

V. Time-Weighted Risk algorithm

A. Description

Given what we learnt from our derived desirable characteristics, we set about modifying the Rahman algorithm to achieve our goals. Rahman already embodies the obvious reasoning, parallelizable and effectiveness scaling traits. We modified it to bias towards the new, resulting in a new algorithm for scoring the bug-propensity of files, which we call Time-Weighted Risk (TWR). It is defined as follows:

$$\sum_{i=0}^{n} \frac{1}{1 + e^{-12t_i + \omega}} \tag{1}$$

Where i is a bug-fixing commit, n is the number of bug-fixing commits, and t_i is the normalized time of the current bug-fixing commit. t_i runs between 0 and 1, with 0 being the earliest commit, and 1 being the last submitted commit (in practice, this value is just the time when the script is run, as something is committed to the Google code base every day). ω defines how strong the decay should be (it shifts the scoring curve along the x-axis).

This equation sums up all the bug-fixing commits of a file, weighting the commits based on how old they are, with older commits tending to zero. Note that TWR only iterates through bug-fixing commits: commits that do not have any bug-fixes are scored zero, and so are not considered. Mathematically, this has no distinction, but computationally being able to quickly

filter these inconsequential commits lowers memory usage and improves runtime. The weighting function is a modified logistic curve, where the x-axis is the normalized time, and the y-axis is the score. The score is not normalized as it is designed to only be used as a comparator between commits.

Of note is that the score for a file changes every day, as the time normalization will alter the weighting of a commit. This is by design, as older files that go unchanged should begin to decay.

We experimented with various x-axis placements of the curve, but found that a very strong decay seemed to be of most use, based on our domain knowledge of Projects A and B. Currently, the strength of decay in the algorithm, represented by ω, is hard-coded to 12. Ideally, this shifting should be dynamically calculated over time to keep the decay window about the same; as time goes on, the window expands. At the time of writing, the window counts commits for about 6 to 8 months before they start to become inconsequential. This levels the field between old files and new files, which are scored based on their most recent issues, rather than taking into account problems in the past that may have been fixed. The value of ω is deployment-dependent, and will vary depending on what decay window makes intuitive sense for the given environment. 12 happened to be the value that worked for our purposes at Google.

B. Step-by-step example

To illustrate how TWR works in practice, consider an imaginary file. The first commit places the file in a brand-new repository, so this is the earliest commit for the normalization calculation. The example will use three differences from a practical deployment to make things clearer. Each commit added will move time forward one unit. In reality, commits are very unlikely to be equally spaced. Also recall that the algorithm TWR only iterates through bug-fixing commits, as commits that don't add any bug-fixes are scored 0 and not worth considering. Normal commits will be included here to better illustrate how the TWR calculation changes over time. Finally, ω will be set to 6, as the 12 used in the Google deployment decays TWR too rapidly.

Commit	1
Bug-fixing?	No
TWR	0.00

The developer finds a mistake, and makes a second commit that fixes the first one.

Commit	2
Bug-fixing?	Yes
TWR	1.00

At this point, Commit 2's normalized time, t_i, is 1.00, and the initial commit's t_i is 0.00. TWR calculates to 1.00 (at three significant figures; at a greater level of accuracy we would see that a positive value of ω causes a slight amount of decay even to new bug-fixing changes). Everything is now fine with the file, and the developer makes another change without issue.

Commit	3
Bug-fixing?	No
TWR	0.500 (Commit 2 with $t_i = 1/2$)

The value of TWR is now recalculated. The bug-fixing commit in Commit 2 is weighted to the normalized time. As there are three commits, Commit 1's time remains 0.00, but now Commit 2 normalizes to a t_i of 0.50, and Commit 3's time is 1.00. With this smaller normalized time, the bug-fixing commit has decayed to be worth much less, and TWR calculates to 0.50. The developer adds another non-bug-fixing commit.

Commit	4
Bug-fixing?	No
TWR	0.119 (Commit 2 with $t_i = 1/3$)

Commits 4 to 9 are also non-bug-fixing. Commit 10 is the next bug-fixing change.

Commit	10
Bug-fixing?	Yes
TWR	1.00 (Commit 10 with $t_i = 1$) + 0.00932 (Commit 2 with $t_i = 1/9$) = 1.00932

Note how the decay function causes Commit 2 to be almost inconsequential in the calculation of TWR at Commit 10. This is intentional. New commits are given much stronger favor over old commits. In this example, even with the reduced value for ω, the decay appears to be very fast. As mentioned previously, in our deployment, the decay window was about 6 to 8 months, so commits don't tail off as quickly as one might infer here.

VI. Bug prediction deployment

A. Overview

Having settled on TWR, we set out to find a suitable place in the developer workflow for the results to be displayed, where we could reasonably expect to adjust developer behavior and encourage developers to show more attention to files which are flagged as bug-prone. Embedding the results in an IDE proved impractical, as Google does not centrally mandate any particular development environment, so a number of plugins would need to be developed. However, all developers that commit to Google's main source repository must have their code peer-reviewed, and the majority of reviews are performed through Google's code review software, Mondrian.

When a developer wishes to commit code, she must first submit that code for review, and nominate reviewers to perform the review. Those reviewers should be knowledgeable about the code that is being changed, and will be able to spot any bugs that may have evaded the original developer. Comments can be added at each line by reviewers and the submitter, leading to brief conversations about bugs, possible refactorings or missing unit tests. The submitter can then submit new code with the requested changes. This cycle can happen any number of times. Once the review team is happy, the submission is marked "Looks Good To Me" and is committed to the source repository.

To help aid reviewers, Mondrian offers a report of which unit tests passed and failed, and runs a lint checker to help

ensure that code meets the company standard. The lint checker runs as a process which attaches comments line by line, using the same mechanism that human reviewers do. The various lint warning levels are attached to the comments, and reviewers can change the warning levels to show or hide the comments.

As most developers use Mondrian, and are already used to seeing code health being reported in the interface, we decided that the best insertion point for bug prediction results would be to attach a comment to line 1 of any file that was flagged as bug-prone. The exact message used was "This file has been flagged as bug-prone by [TWR], based on the number of changelists it has been in with a 'bug' attached. For this warning, 'Please Fix' means 'Please review carefully'." The substituted word is the project codename that we call TWR here. Changelists refers to Perforce changelists, which are commits in other source-code vocabularies. "Please Fix" is a reference to a shortcut that Mondrian offers on lint warnings to allow reviewers to indicate to others that the lint warning should be dealt with. As "Please Fix" was hard-coded into Mondrian as a possible response to lint warnings, we needed to show that the link had a different meaning in this case.

TWR was run as a nightly job across the entire code base, and was set to flag the top 0.5% of the Google code base. Note that TWR was not run at a project level, but using Google's entire source history. This decision was made in order to flag only what TWR considers the very worst files at the company, which we hoped would filter false positives from the list. This meant that some projects had more files flagged than others. Running the algorithm across the entire code base did not affect the output at a UI level: a code reviewer would see a file was flagged in Mondrian, but was not told where it was ranked. Only those who have experience with the portion of the code base in question should be used as code reviewers, so the flag was expected to be meaningful to those who were presented with it.

B. Methodology

To evaluate RQ 3, we deployed our solution in September 2011. Bug-prone files began to have comments attached in Mondrian, and all developers could see it during code review. After four weeks, the comments were given the same priority as a high-level lint warning: developers could see it if warnings were enabled, but otherwise the comment was not visible. This was largely done to bring the project into a consistent state with other warnings, but reduced visibility of our comments.

Three months later, we investigated whether there was any quantitative change in developer behavior. Three months should have been a sufficient time for developers to have either taken to the tool, or chosen to ignore it. We were looking for measurable means to identify whether flagging a file resulted in greater developer engagement with the review; something that indicates they are thinking about a review more deeply than they otherwise would have. We identified two metrics for study:

1) The average time a review containing a bug-prone file takes from submission to approval.
2) The average number of comments on a review that contains a bug-prone file.

To measure this, we generated a list of all the files that had been flagged as bug-prone at any point since launch. We then took reviews from three months either side of the launch date that included these files. We analyzed the time a code review took from submission to approval and the number of comments that the code review gained.

C. Results

1) Developer behavior: We first looked at how the overall mean averages changed before and after launch, as displayed in Table II. For this test, outliers that were lower than the 5th percentile and higher than the 95th percentile of each data set were filtered to prevent extreme outliers from skewing the results.

We then looked at whether the means for each individual file increased or decreased, as displayed in Table III.

Student's t-tests show that neither change was significant, supporting a null hypothesis that the bug prediction deployment had no effect on developers.

2) Developer Feedback: Initial developer feedback was mixed, but leaning towards the negative. We monitored feature requests sent directly to us as bug tickets, and also internal mailing lists where the project had been mentioned. Common complaints included:

- No opt-out function
- Confusion from both submitters and reviewers about how to "fix" the file before the submission would be approved (as the algorithm is non-actionable, there was no means to do this)
- Technical debt files would be flagged again and again, with developers feeling they were in no position to make any positive change
- Auto-generated files were not filtered, and would often be changed, so would end up attached to bug-fixing changes and get flagged
- Teams that were using Google's bug tracking software felt unfairly flagged versus teams that used a different bug tracking method

There was some indication that developers may have tried to sanitize their tracked bug tickets by correctly flagging them with the bug or feature request flag (only bugs, not feature requests, are tracked by the project) but our investigation of the data did not find any significant change in the ratio of bugs versus feature requests being reported, as shown in Figure 3. If there was any change in reporting, so that only bugs are characterized as so, we would expect the number of bugs reported to go down, and the number of feature requests to go up. As the ratio didn't change, we found no evidence that there was any significant change in behavior.

Some developers did see value in the project, and requested means to run the tool only for their own workspaces. However, by and large, qualitative feedback did not have the positive responses that we had hoped for.

Metric	Before deployment	After deployment	Change
Mean time to review (days)	2.01	1.94	-0.08
Mean number of comments	5.97	6.38	0.41

TABLE II: A table showing how the mean time to review and mean number of comments changed before and after the deployment of TWR.

Metric	Increase	Decrease / No Change	Increase Improvement
Mean time to review	464	527	-63
Mean number of comments	524	467	57

TABLE III: A table showing the number of files that saw an increase or decrease in their mean time to review and their number of comments since deployment of the TWR.

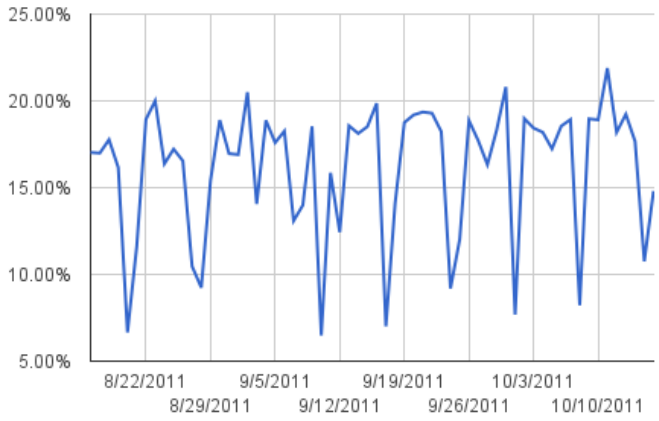


Fig. 3: A chart showing the ratio of bug tickets classified as bugs vs bug tickets classified as feature requests submitted on a given day. The project launched on September 16th, and this shows one month of data either side of that date. There was no significant change in the ratio.

3) Interpretation: We interpret these results not as a failure of developer-focused bug prediction as a whole, but largely as a failure of TWR. The feedback from developers was clear: unless there was an actionable means of removing the flag and "fixing" the file, developers did not find value in the bug prediction, and ignored it. In hindsight, this is not surprising, given the characteristics that developers requested during our informal discussions. We do not believe FixCache would have fared any better.

We were at least expecting to see a small, but statistically significant increase in the number of comments left on a file, as we anticipated reviewers asking questions such as "Are we sure we should be editing this file at all?" While we see a small increase in the number of comments, the increase is not significant.

VII. Threats to validity

A. Assumption of expert primacy

In our evaluation of FixCache and Rahman, we test how well the bug prediction matches the intuition of expert developers. We were interested in whether we could use these algorithms to help convey implicit knowledge about bug-prone files to new developers on a project. However, bug prediction may also help help identify hotspots that may have been missed by every developer, not just those new to the project.

One possible experiment design to take this into account would be to deploy each algorithm to two groups, and have a signal back from developers which indicated both the developer's initial assumption about whether the algorithm was correct or not, and then how the developer felt about the file after investigating the warning. We could then measure which algorithm performed the best after developer investigation, and also see whether the algorithm helped to shift developer views about certain files.

B. Deployment

By conducting this research exclusively at Google, the results may not be generalizable to other companies with other development processes.

Within Google, we identified Mondrian as the best place for deployment, but there were limitations. Ideally, there would have been a better injection point for showing flagged files than an annotation at Line 1 on a file. One quirk of the annotations is that developers were used to only seeing them when there was some action that should be taken at that line, which likely created confusion, as TWR doesn't provide actionable messages.

One developer suggested that instead of being used for code review, bug prediction could have been used as a tool to help guide iteration planning to select cleanup work. In this way, the lack of directly actionable messages is lessened.

Other deployment methodologies were discussed after the initial launch of TWR. One possibility is for results to be part of the IDE, perhaps as part of a system like the Code Orb [19]. Moving the interaction with TWR to the beginning of the code change workflow, where a developer may have more leeway to modify their approach, could yield significantly different results to those presented here.

C. Interviewee pool

We surveyed 10 developers from Project A and 9 from Project B. Although we selected these two projects as examples of 'typical' Google projects, our results could be biased by any unique characteristics of these specific projects.

Another potential bias is in the interviewee pool. While this pool had a good mix of experience levels, we would have

preferred to have more interviewees. In particular, we would have liked to reduce the number of unknown responses for the Duration Cache and Cache-20. Were it not for the performance of the Rahman algorithm, it is possible that our request for developers to comment across the project was unreasonable, and should have perhaps been focused on smaller subsystems. However, the strong performance of the Rahman algorithm shows that developers did seem to have a working knowledge of at least the portion of the code base Rahman presented.

As our interviewees were volunteers, there is the possibility of selection bias in our population. Further study with only developers with exceptionally high experience may yield more accurate results.

D. Metric choice for deployment evaluation

The two chosen metrics, time to review and number of comments, were metrics that were both available and descriptive of the problem at hand. However, it is possible that more suitable metrics could exist, such as checking the number of bugs fixed against a file before and after deployment, or code churn before or after deployment.

Further research should be performed into methods of identifying reception to a tool. In particular, we found many developers talking about the algorithm on mailing lists; a sentiment analysis approach using such discussions may yield interesting insights into tool reception. We also considered modifying the comment in Mondrian to include questions such as "Was this useful to you?" or "Do you agree with this assessment?" with Yes/No buttons, so we could more directly get feedback from developers.

VIII. Conclusion

In this paper, we addressed three different research questions. We found that developers preferred the Rahman algorithm over the FixCache algorithms, although there were still some files developers thought were not bug-prone. We enumerated various characteristics that a bug prediction algorithm should ideally have. We then used this knowledge to alter Rahman and create a new algorithm, but did not observe any change in developer behavior.

We believe our findings do not point to bug prediction being a failure at helping developers. We found developers to be excited about adding a new tool to help aid them in the never-ending code health battle, but these findings provide an illustration that what has currently been developed is not yet useful to them, and hopefully provide a better insight into what might be required for bug prediction to be helpful. We hope that future work will be able to incorporate the missing pieces – particularly actionable messages – and lead to widespread developer adoption.

We would like to caution that our work only pertains to an analysis of human factors when relating to bug prediction, and our study does not analyze the suitability of any algorithm for automated uses, such as test-case optimization. It is also possible that front-line developers are the wrong audience for bug prediction. Instead, software quality personnel might be a better target, as they could use the results of bug prediction to focus quality-improving resources on bug-prone areas.

IX. Acknowledgments

The authors would like to acknowledge the contribution of Googlers Vasily Koroslev, Jeanie Light, Jennifer Bevan, Jorg Brown, Adam Haberlach and John Penix.

References

[1] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, "Predicting Faults from Cached History," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 489–498, IEEE, May 2007.

[2] V. R. Basili, L. C. Briand, and W. L. Melo, "A validation of object-oriented design metrics as quality indicators," *IEEE Transactions on Software Engineering*, vol. 22, no. 10, pp. 751–761, 1996.

[3] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," *Proceeding of the 28th international conference on Software engineering ICSE 06*, vol. 20, p. 452, 2006.

[4] A. E. Hassan and R. C. Holt, "The Top Ten List: Dynamic Fault Prediction," *21st IEEE International Conference on Software Maintenance ICSM05*, pp. 263–272, 2005.

[5] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[6] F. Rahman, D. Posnett, A. Hindle, E. Barr, and P. Devanbu, "BugCache for inspections," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering - SIGSOFT/FSE '11*, (New York, New York, USA), p. 322, ACM Press, Sept. 2011.

[7] E. Engström, P. Runeson, and G. Wikstrand, "An Empirical Evaluation of Regression Testing Based on Fix-Cache Recommendations," in *2010 Third International Conference on Software Testing, Verification and Validation*, pp. 75–78, IEEE, Apr. 2010.

[8] G. Wikstrand, R. Feldt, B. Inst, J. K. Gorantla, W. Zhe, and E. Ab, "Dynamic Regression Test Selection Based on a File Cache - An Industrial Evaluation," *Technology*, pp. 299–302, 2009.

[9] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu, "Fair and balanced?: bias in bug-fix datasets," in *Proceedings of Foundations of Software Engineering*, vol. 139 of *ESEC/FSE '09*, pp. 121–130, ACM, 2009.

[10] A. Avižienis, J. C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, Jan. 2004.

[11] D. A. Norman, *The design of everyday things*. Basic Books, 2002.

[12] B. Schneiderman, C. Plaisant, M. Cohen, and S. Jacobs, *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Pearson Addison-Wesley, 2009.

[13] J. Nielsen, *Usability Inspection Methods*. Wiley, 1994.

[14] L. Layman, L. Williams, and R. Amant, "Toward reducing fault fix time: Understanding developer behavior for the design of automated fault detection tools," in *Empirical Software Engineering and Measurement, 2007. ESEM 2007. First International Symposium on*, pp. 176–185, IEEE, 2007.

[15] D. Hovemeyer and W. Pugh, "Finding bugs is easy," *ACM SIGPLAN Notices*, vol. 39, p. 92, Dec. 2004.

[16] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, pp. 22–29, Sept. 2008.

[17] N. Ayewah and W. Pugh, "A report on a survey and study of static analysis users," in *Proceedings of the 2008 workshop on Defects in large software systems*, pp. 1–5, ACM, 2008.

[18] N. Ayewah, W. Pugh, J. Morgenthaler, J. Penix, and Y. Zhou, "Evaluating static analysis defect warnings on production software," in *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering (PASTE)*, pp. 1–8, ACM, 2007.

[19] N. Lopez and A. van der Hoek, "The Code Orb," in *Proceedings of the 33rd International Conference on Software Engineering - ICSE '11*, (New York, New York, USA), p. 824, ACM Press, May 2011.