Crowd Debugging

Fuxiang Chen and Sunghun Kim

Department of Computer Science and Engineering
The Hong Kong University of Science and Technology
Hong Kong, China
{fchenaa,hunkim}@cse.ust.hk

ABSTRACT
Research shows that, in general, many people turn to QA sites to solicit answers to their problems. We observe in Stack Overflow a huge number of recurring questions, 1,632,590, despite mechanisms having been put into place to prevent these recurring questions. Recurring questions imply developers are facing similar issues in their source code. However, limitations exist in the QA sites. Developers need to visit them frequently and/or should be familiar with all the content to take advantage of the crowd’s knowledge. Due to the large and rapid growth of QA data, it is difficult, if not impossible for developers to catch up.

To address these limitations, we propose mining the QA site, Stack Overflow, to leverage the huge mass of crowd knowledge to help developers debug their code. Our approach reveals 189 warnings and 171 (90.5%) of them are confirmed by developers from eight high-quality and well-maintained projects. Developers appreciate these findings because the crowd provides solutions and comprehensive explanations to the issues. We compared the confirmed bugs with three popular static analysis tools (FindBugs, JLint and PMD). Of the 171 bugs identified by our approach, only FindBugs detected six of them whereas JLint and PMD detected none.

Categories and Subject Descriptors
D.2.5 [Testing and Debugging]: Debugging aids

General Terms
Human Factors, Verification

Keywords
Crowd Debugging, Crowd Sourcing, Debugging

1. INTRODUCTION
Research shows that, in general, many people turn to Question & Answer (QA) sites to solicit answers to their problems. In this paper, we are interested only in QA sites that discuss source code problems, hence QA sites refer only to those sites that are source code related, unless stated otherwise.

Previous research work has shown that users ask recurring questions in QA sites such as Yahoo! Answers and Naver. Allamanis et al. reported that there are recurring questions about code idioms in the developer’s QA site, Stack Overflow (SO). We also observed many recurring questions appearing in SO.

SO is actively used to ask questions for various reasons such as fast response in receiving answers and having many experts in the community. Treude et al. reported that within the first two years of SO’s establishment, there were already three million questions with answers. We observe that many questions and answers contain code fragments where the former display the code in issue and the latter provide the rectified version. Owing to this, we use SO in our study.

Discovering questions that frequently recur strongly implies that many developers are facing similar, if not the same issues in their source code.

However, there are limitations in the QA sites. Developers need to visit them frequently and/or should be familiar with all the content in order to take advantage of the crowd’s knowledge. Due to the large volume and rapid growth of the QA data, it is difficult, if not impossible for developers to catch up.

Regardless of the mechanisms that have already put into place to deter recurring questions in SO, it is still observed that there is a huge number (1,632,590) of recurring questions. Therefore, we are motivated to mine QA sites to leverage the huge mass of crowd knowledge to help developers detect defective code fragments in their source code. With our technique, we propose to find defective code fragments by first detecting code clones before making use of them to triangulate source code anomalies. The defective code fragments (with the crowd’s explanation) are then coupled with the crowd’s suggested solution (with the crowd’s explanation as well) and reported to developers for their concurrence.

Although there are existing similar debugging tools such as FindBugs, PMD and JLint, they are based on manually identified rules and patterns, rather than patterns mined from the crowd. Furthermore, these tools are only detectors as they lack the solutions to the detected issues.

Our approach has the advantage of leveraging the big data of crowd knowledge to check for source code issues, and to
provide solutions and an explanation which are absent from existing static analysis tools.

Our technique is able to generate a reasonable number (189) of warnings in eight high-quality and well-maintained projects, and produce a high percentage (90.5%) of confirmed bugs. Developers show appreciation of our findings as we are able to provide them with solutions and an explanation from the crowd. We also compared our results with existing static analysis tools (FindBugs, JLint and PMD). Six of our confirmed bugs are detected by FindBugs whereas JLint and PMD detected none. Despite being able to detect six of the same bugs, these three popular static analysis tools missed 165 (96.6%) bugs.

Overall, this paper makes the following contributions:

- **An empirical evaluation of recurring questions in QA sites**: We evaluate the number of recurring questions in SO which serves as the main motivation for this research work.
- **A novel debugging technique that leverages the crowds’ knowledge**: We propose a novel technique in leveraging the crowds’ knowledge to detect defective code fragments in software projects and to provide the crowds’ suggestions and explanations to aid in debugging.
- **An empirical evaluation of crowd debugging which includes developers’ feedback and comparison with existing static analysis tools**: We evaluate our technique by reporting the warnings for developers’ concurrence and comparing our results with existing static analysis tools, FindBugs, PMD and JLint.

The remainder of this paper is organized as follows. Section 2 presents the motivation in our design of the crowd knowledge based debugging and Section 3 provides our Crowd Debugging approach using an illustrated factual example. We describe our evaluation settings in Section 4 and portray the results in Section 5. Related work is surveyed and shown in Section 6 while we further analyse the various limitations and threats in Section 7. We conclude with directions for future research in Section 8.

## 2. RECURRING QUESTIONS

Research has been conducted to show that questions are recurring in a myriad of QA sites such as Yahoo! Answers and Naver. Wang et al. found 10,255 recurring questions within a short span of four months (February to June 2008) in Yahoo! Answers. Jee et al. also found 1,557 recurring QA pairs in a collection of 68,000 questions in the “Computer Notice” category in Naver.

In retrospect to Software Engineering, previous research has studied the common questions asked by developers at the project and organizational levels, as well as in mailing lists and discussion forums. Fritz et al. and Sillitto et al. identified developers’ questions that relate to a lack of support in projects and 44 common developers’ questions that relate to software evolution tasks. Hen et al. extracted recurring questions from mailing lists and discussion forums to provide software development related documents to help developers in the software implementation process.

In SO, the QA site tailored specially for developers, recurring questions are however discouraged, and mechanisms have been put into place to detect and deter recurring questions. Similar questions are also shown when a developer posts a new question.

Recurring questions in SO are generally categorized into three groups: exact word-by-word copying and pasting, partial use of keywords in original questions and having subtle semantic differences to the original questions that do not belong to the previous two groups. If there is a recurring question in SO, developers can vote to include it with comments and links to the original question. This will then result in the question being modified to reflect it as recurring. Repeating questions can also be merged by notifying moderators to perform a merge operation on the recurring questions.

### Table 1: Number of Recurring Questions in SO since its establishment.

1,632,590 questions are recurring despite mechanisms put into place to prevent them. SO users can also detect duplicates after posting their questions to further eliminate them. Therefore, we believe this 8.2% (1,632,590) is a high number for recurring questions.

<table>
<thead>
<tr>
<th>Period</th>
<th># of Questions</th>
<th># of Recurring Questions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aug '08 - May '14</td>
<td>19,881,018</td>
<td>1,632,590 (8.2%)</td>
</tr>
</tbody>
</table>

We have observed recurring questions in our use of SO on several occasions and this led us to investigate the number of recurring questions in SO. Table 1 shows the total number of questions in SO from August 2008 till May 2014; the number of identified recurring questions and its ratio with respect to the total number of questions.

Despite having mechanisms put into place to prevent recurring questions from taking place, and SO users can detect duplicates after questions are posed to further eliminate them, we observed many recurring questions in SO from our dataset. Although the ratio of recurring questions to the entire question set is around 8%, it is still a huge number (1,632,590).

We therefore hypothesize that programming issues faced by a developer may also be faced by others, and questions asked in SO can be leveraged to help other developers in similar situations. This further motivates us to design crowd knowledge based debugging by tapping into the existing pool of knowledge from the QA database. We have used all the question data from August 2008 till May 2014 and we do not limit ourselves to only the recurring question data so as to increase the chances of a higher detection rate.

## 3. APPROACH

This section describes our approach to leverage the crowd’s QA knowledge to detect defective code fragments.

Figure 1 shows the overview of our approach. SO question-answer code-pairs have been populated in a database and we perform code clone detection between the target source code and the question code blocks in SO to identify similar fragments in the target code. The detected pairs of code clones are the shaded regions in Figure 1. The shaded regions in the first figure depict two matched clone fragments (M1 and M2) from the target source code while the shaded regions in the center figure show the matched clones (QF1 and QF2) from the SO question code blocks.
Figure 1: Overview of approach. The target source file is compared against the SO question code blocks for code clones. The code-like terms of the matched clone in the question are further compared with those in the answer code blocks. A matched code-like term denotes that the clone fragment in the target source file is potentially defective. Post filters are then applied to reduce the potential false positives.

The code-like terms from the detected clones in SO question code blocks are then compared with the code-like terms in the corresponding answer code blocks. Code-like terms are sequences of characters that resemble code elements. This step is needed as we observed questions and answers containing code fragments usually have some identical code terms. The questions usually contain the problematic code while the answers provide the rectified version. If there is a matched code-like term, it will be marked as a potential defective code fragment.

We further applied post filtering process to identify and remove potential false positives and the output will be treated as a defective code fragment. Together with the SO answer with explanation and its URL, these will be reported to the developers.

In Figure 1, M1 is a defective code fragment as there are matched code-like terms between its matched clone QF1 and the answer code-blocks. M2, however, is not a defective code fragment as there is no matched code-like term between QF2 and the answer code-blocks.

Our approach can be seen in five consecutive phases, namely, Code-Pairs Database Building, Clone Detection, Element and Title Matching, Post Filtering, and Reporting Defective Code Fragments. The following subsections explain each phase with a concrete example from JFreeChart.

3.1 Code-Pairs Database Building

Our goal in this phase is to create and populate a database that contains question-answer code-pairs from SO.

Specifically, we are interested in questions that have the highest scored answers (at least a 1 up-voted score) and contain code elements within questions and answers. If there are multiple answers with the same high score to a question, we will choose the most recent answer even though it might not have been explicitly marked as accepted. We observed that better answers can be posted after a previously given answer has been marked as accepted. An answer to a question in SO is voted upwards manually to give a score to the answer by the community of developers if the solution presented solves the issue highlighted in the question.

The highest scored answers would mean that the answers have a strong consensus among a larger group of developers. Code elements are important in our approach as we need them for detecting faults in the target source code.

Most code elements are formatted in code blocks inside the question and answer sections of a SO post. The code elements are either embedded inside the <code> tag for inline code or the <pre> <code> tags for blocks of code. We define code blocks here as the entire content inside either of the previous mentioned encompassing tags.

The detected question-answer pair is denoted as <Q, A> where Q refers to the code blocks from the SO question body and A refers to the corresponding answer code blocks.

In this paper, we consider only Java questions. We use the public SO data from the Stack Exchange Data Dump website for convenience purposes as all the data we needed are structured in the XML form Posts.xml for easy extraction. In particular, we harvest Java tagged questions from Posts.xml which contains all SO questions and answers since its establishment from August 2008 to May 2014 (~2 GB in file size). A total of 309,453 code-pairs are mined and used for detecting defective code fragments. Future addition of new SO data can be harvested quickly by querying the Stack Exchange Data Explorer by the questions’ creation time. However, our approach is not limited to Java. This is because SO contains questions for different types of programming languages. Extracting code fragments of another programming language can be replicated similarly by mining SO questions with the tagged keyword of that programming language (e.g., C++) instead of Java. Furthermore, the clone detector (CCFinderX) used in the next phase (Section 3.2) is able to identify code clones using textually similar tokens regardless of the type of programming languages.

3.2 Clone Detection

Our goal here is to compare the target source code from the software projects with Q to detect similar code in the target.

We experimented several state-of-the-art code detection techniques such as Graph-based (GrouMiner), Tree-based (Deckard) and Function-based (SimCad). These techniques depend on existing Java Language compilers such as the Eclipse Java Compiler. These compilers can handle some errors but cannot handle partial code fragments very...
Many code fragments in SO are structured coarsely but incomplete, thus making them non-compilable. Despite wrapping with enclosing classes/methods in mitigation to the code with no class name or method signature, we were still faced with many compilation errors which were due to missing types. For example, SimCad is unable to parse 267,174 (86%) of the SO code fragments. Existing state-of-the-art Partial Program Analysis (PPA) which attempts to recover missing types also depends on Java Language Compiler and has shown ineffective in processing code from SO. In light of these limitations, we turned to token-based code detection approach which is independent of any language compiler.

To detect more precise and accurate code clones, we restrict the code clone detection to only Type I and II clones. Type III clones have several constraints such as producing many false positives, not being representative of all subjects, and require individual subject experiments in determining the lines of gap for omission.

We used CCFinderX, a textual token-based code clone technique to achieve this detection goal. CCFinderX is chosen for several reasons. Besides being able to detect Type I and II clones efficiently, it also has the ability to detect a wider range of code fragments in different languages.

As we do not want our detection to be so fine-grained that it covers every single statement it may detect, we limit the minimum number of detected tokens to 30 (default is 50) in CCFinderX. This number is chosen because it symbolises a concrete short method with a method name, the return type and a single statement body. We want our clone detector to identify as many useful clones as possible.

The use of a minimum number of tokens means that several detected code clone pairs can come from the same pair of source-target code-pair. As long as it meets the constrained minimum of 30 tokens, CCFinderX will extract the clone set that covers the largest consecutive code lines that contain the tokens.

The detected code-pair is denoted as \( < QF_n, M_m > \) where \( QF_n \) refers to the \( n^{th} \) detected code fragment in \( Q \) and \( M_m \) refers to the \( m^{th} \) matched code fragment from the software projects.

As an example, CCFinderX detected a code clone pair between AxisEntity.java in JFreeChart and the SO question ID: 7132649. Since there is only one detection between AxisEntity.java from JFreeChart and the SO question ID: 7132649, the detected code-pair is \( < QF_1, M_1 > \) where \( QF_1 \) represents the code fragment in (b) SO Question Block (Figure 1) while \( M_1 \) represents the code fragment in (a) Target Source code Figure 1.

Then, \( < QF_1, M_1 > \) is traced back to the corresponding \( < Q, A > \) to form the clone fragment-answer code-pair \( < QF_1, A > \) for the next phase processing.

### 3.3 Element and Title Matching

Our goal in this phase is to identify the potential defective code fragments from the suite of \( M_m \) identified in the earlier phase (Section 3.2). The collection of \( M_m \) in Section 3.2 is the set of candidate code fragments where they might be defective.

We discard a detected code-pair if there are no matching code-like terms between the SO title and \( QF_n \), the SO title and \( M_m \), and \( M_m \) and \( A \). We observed many of them are false positives and thus are not essential to our detection. The matching will involve stemming the natural words/code-like terms using Porter’s Stemming Algorithm before comparing them.

Also, we observed \( QF_n \) containing too many or too few lines of code are false positives. In addition, SO does not advocate pasting large chunk of code. We experimented and found \( QF_n \) whose code lines are greater than 15 or less than or equal to three are false positives. We removed such warnings.

We also compare the code-like terms for similarity between the \( < QF_n, A > \) code-pairs from Section 3.2 to determine if the corresponding \( M_m \) is defective. If at least one code-like term is found in both \( QF_n \) and \( A \), then the corresponding code fragments \( M_m \) from the software project are considered to be potentially defective. It is because we observed that in many answers, there are references of code elements from the questions that are having issues. The crowd then usually highlights the problematic code elements from the question and gives a rectified solution in the answer.

Specifically, we are interested in the following mentioned types of Java code-like terms as they generally make up partial or complete code structures that are essential in our detection. Moreover, many other languages have similar structure and can be replicated across: classes/objects (includes inherited, inner and static classes), variables (includes class and local variables), methods (includes method definition, single method calls and method chains), parameters, operands (e.g. in the logical comparison \( foo == bar \) and \( foo \) and \( bar \) are both named operands) and constructors. For parameters, we considered only named parameters to be important as the code fragments in SO may not explicitly include all variable definition and the parameters might be the issue in discussion. Similarly, only named operands are being extracted. If qualified terms (e.g. \( foo.variable \) or \( foo.method() \)) are present, they are segregated into classes/objects, methods or variables.

We used the Java Language Specification as a guide to help us create an island parser. An island parser is a parser that extracts only the interesting constructs.

In our study, the interesting constructs are the code-like terms and the island parser is based on island grammar which is a set of multiple regular expressions where they have demonstrated to be more effective than lightweight regular expressions and traditional IR techniques.

For the example of AxisEntity.java in JFreeChart and the SO question ID: 7132649, we extracted all code-like terms from \( QF_1 \) (Detected Code Fragment in SO Question Block) and \( A \) (SO Answer Block). The output of using the island parser on \( QF_1 \) and \( A \) are two sets of code-like terms from \( QF_1 \) and \( A \) respectively. The set of code-like terms in \( QF_1 \) are \( a, Book \) and other while the set of code-like terms in \( A \) are \( a, equals, b, Book, ComicBook, hashCode, this, getClass \) and \( a \). Subsequently, we matched the code-like terms from the SO question block to the SO answer block. There are two matched terms, \( Book \) and \( a \), linked from \( QF_1 \) to \( A \). They are then traced back to \( M_1 \) and this sketches out the potential defective code fragment.

Next, we apply post filters to reduce the false positives and identify the defective code fragments. We explain the filtering process in Section 3.4.
3.4 Post Filtering

To eliminate potential false positives, we develop several filters. Figure 4 on the right shows the filters that we have applied.

Table 2: The list of non-essential statements warnings from 8 subjects, 21,333 warnings.

<table>
<thead>
<tr>
<th>Filter Code Pattern</th>
<th># Filtered</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primitive Parsing</td>
<td>373</td>
</tr>
<tr>
<td>Object Creation</td>
<td>4,293</td>
</tr>
<tr>
<td>Appending Collection Items</td>
<td>324</td>
</tr>
<tr>
<td>Trivial Small Code Construct</td>
<td>554</td>
</tr>
<tr>
<td>Overridden Methods</td>
<td>242</td>
</tr>
<tr>
<td>Single Statement Block</td>
<td>205</td>
</tr>
</tbody>
</table>

3.4.1 Non-essential Statement Filters

Kawrykow et al. [31] studied non-essential code changes in software archives and reported that these non-essential code changes can cause inaccurate representation of software development effort. Similarly, we observed that there are many non-essential statements in our detection. Non-essential statements are code statements that are very common but usually not very relevant to program correctness. They exhibit several code patterns and are listed in Table 2. To eliminate these non-essential statements, we develop the following filters.

- **Primitive Parsing, Object Creation, Appending Collection Items**: For QFn that contains primitive parsing code (e.g. `Integer.parseInt`), object creation code (e.g. `new AnObject()`) and appending of collection item code (e.g. `obj.put("item")`), we observed that they are very common but not relevant to errors. We experimented and observed that if the line number ratio between these code patterns and QFn is greater than 0.4, we removed them. A total of \((373 + 4,293 + 324 = 4,990)\) warnings were removed from the 8 subjects.

- **Trivial Small Code Construct**: Many QFn having loop (e.g. `for/while`), condition (e.g. `if`) and exception (e.g. `try/catch/finally`) constructs are usually structurally and semantically correct. If the QFn has total line numbers less than or equal to four lines of these constructs (each construct having four lines of code), then they are removed. A total of 554 warnings were removed from the 8 subjects.

- **Overridden Methods**: Overridden methods (`toString()`, `run()`, `compare()`, `hashCode()` and `equals()`) are also observed to be mostly false positives and we remove them. For the overridden methods, we discard them only if their method bodies do not conform to the correct contract structure [29] as defined by the Java Specification [27]. For example, according to the Java Specification, the `equals()` method must be reflexive, symmetric, transitive and consistent. In the case of symmetric, it is stated that \(x.equals(y)\) should return true if and only if \(y.equals(x)\) returns true. We noted that the use of `instanceof` is asymmetric and is not a valid symmetric contract. If the `equals()` body uses `instanceof` for comparison, then the `equals()` method is seen as non-conformance to the Java Specification and will not be discarded. A total of 242 warnings were removed from the 8 subjects.

- **Single Statement Block**: Within QFn, a single statement in a block of curly brackets is removed as we observed that many of them are trivial and are false positive. A total of 205 warnings were removed from the 8 subjects.

3.4.2 Similarity Filters

After applying the non-essential statement filters, we perform Similarity Filter #1; and Similarity Filter #2; separately to ensure the residual code from QFn and Mm have some matching. The distinct output from these two filters serve as the final warnings generated by our technique.

- **Similarity Filter #1**: Warnings whose distinct matching code-like term in QFn and A (Figure 4) having a QFn/A ratio of less than 0.8 are removed. We experimentally found that 0.8 is the best threshold in our study for this filter. A total of 4,339 warnings were removed from the 8 subjects.

- **Similarity Filter #2**: If there are warnings whose distinct matching code-like term in QFn and A having a QFn/A ratio of less than 0.37, and distinct matching code-like term in Mm and A having a Mm/A ratio of less than 0.25, then they are removed. We experimentally found that 0.37 and 0.25 are the best thresholds in our study for this filter. A total of 10,814 warnings were removed from the 8 subjects.

3.5 Reporting Defective Code Fragments

We collect the warnings generated and filtered from Section 3.3 and Section 3.4 and report them to developers of each project. We discuss the false positives, together with the disagreed bugs by the developers, in Section 5.

We report all found suspicious defective code fragments to developers for their concurrence in the form of a simplified bug report that is generated automatically.

The bug report includes the source file name, its defective code fragments with line numbers, a summarized suggested solution from SO and the crowd’s explanation on why the code fragments are defective with a link to the original SO post.

![Figure 2: Report structure of a defective code fragment from JFreeChart’s AxisEntity.java](stackoverflow.com/questions/7132649)

For example, a report sample from JFreeChart is shown in Figure 2. Besides indicating the source file (`AxisEntity.java`) and the defective code fragment `M`, we also highlighted to the developers the defective code fragment and the explanation from the crowd. According to the crowd, `M1` represents an incorrect symmetric contract inside the `equals()` method due to the use of `instanceof` in the object comparison. For the `equals()` method to be contractually correct, the comparison of objects within `equals()` must be symmetric. The suggested solution from the crowd is to use `getClass()` instead. We also...
cited the SO post URL\footnote{http://stackoverflow.com/questions/7132649} in the report for the developers’ reference.

4. EVALUATION SETTINGS

In this section, we describe how to evaluate our approach for its effectiveness in detecting defective code fragments in software projects.

4.1 Research Questions

To evaluate our approach, we seek to address the following research questions in our experiment:

RQ1: (Detectability) How many warnings can be detected using our technique? We address the effectiveness of our technique in leveraging the crowd’s bug patterns to detect warnings in software projects. More specifically, in this RQ, we are interested in the quantity of the detected warnings, which serves to imply how thorough and effective our technique is in unravelling defective code fragments in software projects.

RQ2: (Confirmed Bugs) How many warnings from our technique are confirmed as bugs by developers? By presenting the total number of warnings generated from the tool, it is imperative to distinguish between false and true warnings. We investigate if the findings from our technique make sense to the developers and whether they will accept the solution proposed by the crowd.

RQ3: (Comparison) How many confirmed bugs from RQ2 can be detected/missed by other static analysis tools such as FindBugs, JLint and PMD, and how many bugs are missed from our technique as compared to the static analysis tools mentioned? Existing static analysis tools can detect some types of bugs. In this RQ, we investigate how many of the confirmed bugs detected from our technique can also be detected likewise by the static analysis tools. To make our evaluation more holistic, we also investigate on the number of useful warnings missed from the static analysis tools when leveraging our crowd debugging technique.

4.2 Subjects for Evaluation

We evaluate our technique with the latest version of eight open-source projects at the time of our experiment shown in Table 3. We exclude unit test classes as typically, unit test warnings, which serves to imply how thorough and effective our technique is in unravelling defective code fragments in software projects.

4.3 Evaluation Methodology

We run our technique on these subjects to generate warnings. This corresponds to the approach for RQ1.

Then, we present the generated warnings in emails to the developers (most recent committers of the source files that display the generated warnings) of those subjects to solicit their feedback. The developers’ individual email addresses are located publicly in the repositories (e.g. SVN/GitHub). If there are no responses from the developers or if the developers do not wish to engage in private emails, our follow up strategy is to send the same emails to the developer’s group mailing lists, in hope for feedback from a bigger group of developers.

We include in the email manually the detected warning which consists of the source file and its defective code fragment, the proposed patches with explanation to the defective code fragment based on the crowd’s suggestion, and the related Stack Overflow post for references. We seek developers’ opinions on the reported warnings. We present the response and concurrence rates in Section 5. This said approach corresponds to the approach of RQ2.

For RQ3, we first check for similarity by comparing the confirmed bugs from our approach with the warnings generated by the static analysis tools. Then, we sample, inspect and verify the warnings generated by the static analysis tools. These are then compared with our approach to check for missed bugs. More details are illustrated in Section 4.4.

4.4 Static Analysis Tool Comparison

Many existing bug detection tools use static analysis approaches with predefined bug patterns to check for buggy code in source files \cite{34,33}.

We selected three tools, PMD \cite{35}, FindBugs \cite{36} and JLint \cite{37}, and ran them with all the eight evaluated subjects. These three tools are chosen as they have been widely used in many bug related research \cite{38,34,39}. We used the default options for FindBugs and JLint. For using PMD, we need to supply the tool command with rulesets for it to function accordingly as there is no default rulesets built-in. We selected five rulesets including “basic”, “design”, “type resolution”, “optimization” and “controversial” which cover a good spectrum of debugging patterns for use in our comparison.

We manually inspect the warnings between these three tools and our technique. We first checked for the number of same bugs detected in both our technique and the static analysis tools.

We then perform manual inspection on the warnings of the top 3 buggy files generated by the tools. Many warnings produced by existing static analysis tools have been shown to display a high false positive rate \cite{38}. We are interested in warnings that are useful. Therefore, we characterized a warning to be useful if it is non-trivial and identifies either a bug, poor performing code or code that can be refactored for easier maintenance.

Due to limited resources and the large number of generated warnings from the tools, we sampled the top 3 files with the highest frequency of warnings from the top 3 largest subjects (lucene, log4j & JFreeChart). We believe that a source file with more warnings has a higher tendency to be

---

Table 3: The list of subjects with their latest version and size.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Version</th>
<th># Files</th>
<th>K LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-lang</td>
<td>3.4-snapshot</td>
<td>274</td>
<td>63</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>1.0.19</td>
<td>654</td>
<td>96</td>
</tr>
<tr>
<td>joda-time</td>
<td>2.4</td>
<td>315</td>
<td>80</td>
</tr>
<tr>
<td>JStock</td>
<td>1.0.7r</td>
<td>290</td>
<td>50</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>1.0</td>
<td>34</td>
<td>3</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>2.0</td>
<td>56</td>
<td>4</td>
</tr>
<tr>
<td>log4j</td>
<td>2.0.1</td>
<td>905</td>
<td>61</td>
</tr>
<tr>
<td>lucene</td>
<td>4.9.0</td>
<td>4826</td>
<td>684</td>
</tr>
</tbody>
</table>
defective and thus may contain more useful warnings. Two independent graduate students were chosen to verify and inspect the usefulness of the warnings from the tools. If there are differences in the checking, a third independent graduate student will be roped in to check for the inconsistencies and differences will be reconcile amongst the three. These three inspectors have Java programming experiences of at least five years on average, and have previously worked in the industry as developers.

We report our comparison in Section 5.3.

5. STUDY RESULTS

This section presents our experimental results by addressing the research questions (Section 4.1).

Table 4: The number of warning types and warnings for each subject by our technique. The warning types reflected in this table are non-distinct across all the subjects. The distinct warning types for each subject are presented in Table 5. This table also presents the number of confirmed bugs by the developers.

<table>
<thead>
<tr>
<th>Subject</th>
<th># of warnings</th>
<th># of warning types</th>
<th># of reported warnings</th>
<th># of confirmed bugs</th>
<th># of rejected warnings</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-lang</td>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>102</td>
<td>2</td>
<td>102</td>
<td>102</td>
<td>0</td>
</tr>
<tr>
<td>joda-time</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JStock</td>
<td>19</td>
<td>2</td>
<td>18</td>
<td>18</td>
<td>0</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>log4j</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lucene</td>
<td>10</td>
<td>1</td>
<td>10</td>
<td>10</td>
<td>0</td>
</tr>
</tbody>
</table>

Total 189 15 181 171 1

Table 5: The different types of warnings for each subject by our technique.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Detected Warning Types</th>
<th>SO Question ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-lang</td>
<td>Optimizing array copying performance</td>
<td>10416290</td>
</tr>
<tr>
<td></td>
<td>Best practice in using volatile fields</td>
<td>16101203</td>
</tr>
<tr>
<td></td>
<td>Better version of Double Checked Locking</td>
<td>17169145</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>Bug in overridden equals() in subclass</td>
<td>7132649</td>
</tr>
<tr>
<td></td>
<td>Non-conformance to Java Spec.</td>
<td>7132649</td>
</tr>
<tr>
<td>JStock</td>
<td>Refactoring for Code Maintenance</td>
<td>1347996</td>
</tr>
<tr>
<td></td>
<td>Bug in JTable autoscrolling</td>
<td>1347996</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>Preventing NullPointerException</td>
<td>2032770</td>
</tr>
<tr>
<td></td>
<td>Java MDS Compilation Error</td>
<td>4804615</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>Comparing two dates in Hibernate</td>
<td>5916454</td>
</tr>
<tr>
<td>log4j</td>
<td>Asking if concurrency bug is possible</td>
<td>18154004</td>
</tr>
<tr>
<td></td>
<td>Bug in Overriding equals()</td>
<td>22346976</td>
</tr>
<tr>
<td>lucene</td>
<td>Bug in deprecation in transferTo</td>
<td>7132649</td>
</tr>
<tr>
<td></td>
<td>Non-conformance to Java Spec.</td>
<td>7132649</td>
</tr>
<tr>
<td></td>
<td>Non-conformance to Java Spec.</td>
<td>19642810</td>
</tr>
</tbody>
</table>

5.1 RQ1: Detectability

We investigate if our technique in leveraging crowd knowledge can detect anomalies.

We ran our experiment in a desktop computer running an Intel Core i3-2100 3.1 GHz CPU, 4 GB RAM, Windows 7 64 bit OS and a 250 GB 7200 RPM HDD. The experiment on all the eight subjects took between 4.914 seconds and 21.222 seconds, with an average processing time of 10.667 seconds. After running our tool on the eight subjects (Table 4), our tool returns a total of 189 warnings with various warning types in each subject. These types are bugs, non-conformance to Java Specification, code performance optimization, code best practices, prevention of NullPointerException and code refactoring of smelly codes (Table 4). For example, in JFreeChart, there are 6 warnings that belong to the same bug type. This bug is about overriding of the equals() method. The child class uses instanceof for the comparison inside the equals() method, but this is non-symmetric and will result in the wrong comparison between the child and the parent object. In Table 4 we display the subjects with their corresponding warnings and warning types in the first three columns, where the first column shows the subjects and the second and third column display the number of warnings detected and the number of warning types respectively.

A total of 152 warnings were detected in JFreeChart, followed by the runner-up subject JStudyPlanner which starred 19 warnings and the second runner-up subject lucene of 10 warnings. Commons-lang exhibited 3 warnings and a couple of subjects, JStock and log4j exhibited 2 warnings. Our technique also detected 1 warning in JStudyPlanner 2 but did not detect any warnings in joda-time. We believe that 189 is a reasonable number as it is not too many to exhaust the developers’ time in checking the warnings.

Clearly, this shows that our technique is able to detect warnings on different variety of subjects and the number of warnings varied depending on the existing anomalies presiding in the subjects and the preceded crowd pattern.

Upon answering this RQ, it naturally serves as a connector to lure us into the next RQ, which is to check if developers agree and confirm to the detections.

5.2 RQ2: Confirmed Bugs

This RQ serves to indicate if developers agree with the warnings produced by our tool.

As presented in Section 4.3 we reported all warnings on the subjects generated by our technique (after removing the obvious 8 false positives which will be discussed in the later paragraphs) to the developers. The warnings are complemented with the SO posts which contain the suggested solutions and explanation on the issues.

Table 5 shows the number of produced warnings reported to the developers and the number of warnings agreed/disagreed by the developers.

Amongst the 181 reported warnings, developers responded to 171 of them and 9 are pending. The number of distinct confirmed warning type is 4 out of 14. Even though developers did not respond to some of the reported warnings, there is still a promising 90.5% agreement in all the reported warnings. For JFreeChart, the developer mentioned that he would make all the changes in JFreeChart - Future State Edition, the next major revision of JFreeChart. Similarly, for JStock, although the developer agrees with the warning, he requested to submit a Git pull-request for the changes. We also noted that for JStock, the user who asked the question in SO is the same JStock developer and that he has marked the answer as accepted in SO.

The high number of agreement in the warnings by the developers implies that most of our detected warnings are
true positives. Developers recognize the code issues and accept the proposed solution. It ultimately serves the goal of debugging: Detect (warnings/anomalies in a software project) and accept (the proposed solution).

In addition, our tool also provides comprehensive explanation using the SO posts to the developers. This will further enhance their understanding on the detected warnings. Our crowd debugging indirectly transfers tacit knowledge from the crowd to the developers.

We showcase some of the email responses by developers who concurred with the warnings in Figure 3 and Figure 4.

Figure 3: JFreeChart Project Leader acknowledging the reported warnings on the bug within the overridden equals() method as well as non-conformance to the symmetric contract rule specified in the Java Specification.

Figure 4: JStudyPlanner Project Owner acknowledging the reported warnings on possible NullPointerException.

In Figure 3, our technique detected that there are several database access methods that might cause NullPointerException as highlighted by the crowd. It is suggested to have null checks on the objects before calling any of their methods. The developer acknowledged and appreciated these findings.

Discussion on false warnings We discuss the 8 false positives generated by our technique and the disagreement in warnings by the developers. We also discuss the 9 pending reported warnings that have not been responded by the developers.

Querying SO for Better Code or Code Explanation
In common-lang, we detected 3 different warnings. Users are asking in SO on how to optimize a certain code and whether there is a better version of the provided code. In log4j, we detected 2 warnings. The first warning is about asking for explanation for a code that did not perform as intended whereas the second warning is about asking if a concurrency bug may be present in a certain code. We noted that these detected defective code fragments in the Target subjects are either the solutions provided by the crowd, or that it is not directly relevant to the target context. Thus, we did not report them to the developers.

String Literals as Method Arguments In JStudyPlanner and JStudyPlanner 2, our technique detected 1 warning each in both subjects. The detected warning in JStudyPlanner is on Java MD5 compilation error and in JStudyPlanner 2 is on comparing two dates in Hibernate. Although the code is similar between the Target and SO, in JStudyPlanner, the detected code fragment is on SHA-256 instead of MD5, and in JStudyPlanner 2, the detected code fragment is about querying a database table in Hibernate instead of comparing two dates in Hibernate. We noted that these two warnings exhibit the same characteristic of using string literals as method arguments. For example, in JStudyPlanner, the developer’s code is MessageDigest.getInstance(“SHA-256”); instead of MessageDigest.getInstance(“MD5”); in SO. These are false positives and we did not report them to the developers.

Omission of Bug Related Statements in Target In JStock, we detected a warning where a SO user has complained about an autoscrolling bug in JTable. The code fragments in both the Target and SO are very similar with the exception that in SO, there is an additional statement of resizing the JScrollPane. This extra statement is the cause of the bug but it was absent in the Target code. We noted that this is a false positive and we did not report them to the developers.

File Transfer In Lucene, we reported to the developers that the use of transferTo should decrement the max count in every iteration based on the suggestion given by the crowd. transferTo is used in Lucene to transfer files from one location to another. The current implementation in Lucene forsakes the decrement of the max count in the iterations. Developers do not totally agree with our warning and argued that the SO post is mentioning about file transfer of size larger than 64MB whereas in Lucene it is limited to 32MB. Surprisingly, we observed that after the developer responded to us, the said transferTo method was removed and replaced with another similar function from commons-io. This observation has an implication that crowd debugging does provide alternative insights and impact on the source code.

Non-conformance to Java Specification In Lucene, we detected and reported 9 warnings that are related to the overriding of the equals() method. The detected code fragments do not conform to the symmetric contract rule as stated in the Java Specification. The crowd suggested the use of getClass() instead of instanceof in comparing two objects and this will satisfy the symmetric contract rule in the Java Specification. We believe that these are true warnings and will ensure that the code fulfill all the contract rules stated in the Java Specification.

Although our technique detected some false positives, we noted that several of them are related to code explanation
and developers may still benefit from the solution given by the crowd. Developers may gain additional insights into the code that they are writing or pay more attention to the code since others have previously casted doubts on them. This is especially true if the explanation is on complicated or hard-to-understand code. In addition, Bessey et al. [40] have argued that to be a useful tool, its false positives should be less than 30%. Our study shows that 9.5% of our reported warnings are false positives, which indicates our tool is useful in practice.

Overall, the large percentage of concurrence (90.5%) by the developers and low false positive rate (9.5%) indicate our crowd debugging tool is promising in practice.

5.3 RQ3: Comparison

This RQ serves as a comparison between the warnings produced by existing static analysis tools and our tool. It checks for similarities between the confirmed bugs from our tool and the warnings generated from the static analysis tools. It also serves to detect the number of missed bugs by our tool when compared with the static analysis tools.

Table 6: The number of confirmed bugs detected by our technique and static analysis tools PMD, FindBugs and JLint. The 165 confirmed bugs detected by our tool are undetectable by PMD, FindBugs and JLint.

<table>
<thead>
<tr>
<th>Subject</th>
<th># of confirmed bugs</th>
<th>FindBugs</th>
<th>JLint</th>
<th>PMD</th>
</tr>
</thead>
<tbody>
<tr>
<td>commons-lang</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JFreeChart</td>
<td>152</td>
<td>6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>joda-time</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JStock</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JStudyPlanner</td>
<td>18</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>JStudyPlanner 2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>log4j</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>lucene</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>171</strong></td>
<td><strong>6</strong></td>
<td><strong>0</strong></td>
<td><strong>0</strong></td>
</tr>
</tbody>
</table>

In addition, we check for missed bugs from the static analysis tools. After manually inspecting the warnings generated by the static analysis tools, out of the 2,722 generated warnings from the sampled files, 3 (0.1%) are useful in lucene’s SolrCore.java file. These 3 warnings indicate that a particular object can be null but lacks the null checking on the object before accessing its methods in the code. We believe that these are useful warnings in preventing a NullPointerException. Although we missed 3 useful warnings, we have shown that by leveraging crowd knowledge, our tool is more efficient in detecting bugs that have been missed by these static analysis tools. This also implies that many of the crowd knowledge patterns are unique and are not easily replicated using the static analysis approach.

6. RELATED WORK

Source Code Pattern Matching Techniques Much research has been conducted on code clone techniques and they can be broadly categorized into string based [41, 12], token based [43, 44, 45], tree based [46, 17, 22, 48] and semantic based [49, 50]. However, in several practical applications, code clone alone is usually insufficient or inappropriate to apply. Thereafter, other research work has been incorporating code clone techniques as a sub-task or designing different techniques for source code pattern matching. Liu et al. [51] used call graphs and dependency graphs to detect performance buggy code in Android. Lin et al. [52] identified similarities and differences of multiple code clones by first tokenizing all the clone instances and computing the longest common subsequence. This will speed up comparing code clones in a pair-wise manner iteratively. Liang et al. [43] represented code as finite state machines to track similar bugs. Meng et al. [53] used dependency graphs to extract the best path for the source and target code in systematic matching. In another separate work, Meng et al. [54] used CCFinder [43] to get a high level detection before transforming the source code into AST to get the longest common subtree. Kim et al. [55] inspected human-written patches to uncover source code patterns for use in automatic patching. Sanchez et al. [56] proposed an idea of a search engine to reformat previous query results by eliminating redundant code or modifying existing code, and then transforming them into another input to the search engine for a better code matching result. Ponzanelli et al. [57, 58] tokenized code and natural texts, and used Lucene as a search engine to produce matches. Jiang et al. [59] used Deckard [22] to detect code clones in AST form and used control dependency graphs to eliminate inconsistencies for extracting clone related bugs. Ashok et al. [60] designed a search engine that tokenized code with other debugging information and used TF-IDF to search through Microsoft repositories to produce a ranked result based on previous debugging information. Our code pattern matching technique uses CCFinderX, similar to Meng et al. [53] as the initial processing but differs in the latter stages where we tokenize the code in QA to detect relevancy.
Informal Documentation Bacchelli [29] [61] used mailing list to leverage its content for developer’s software implementation. Dagenais et al. [14] discovered linkage between API and informal documentation such as mailing list and discussion forums. With respect to SO, multiple work have been conducted on it. Rigby et al. [15] identified important code elements in SO as an entire entity. Ponzanelli et al. [57] [58] used SO for designing search engine capabilities to prompt developers who are in their current stage of coding for similar code discussion to ease implementation. Linares-Vasquez et al. [62] investigated the quantity and types of questions asked when API changes in Stack Overflow, thus allowed insights into some of the popular new API in discussion. Saha et al. [63] investigated the reasons behind unanswered questions in SO and concluded that the majority of them were due to low interest in the community. Bajaj et al. [64] mined the SO to uncover the different types of questions asked by web developers. They analysed the questions and noticed web development related questions were increasing over time. They discussed that educators can use the results to enhance the developer’s understanding and researchers can focus on the more discussed web development areas. Similar to several previous work, we incorporate SO in our research but differ in usage which in our case is unique and is used for crowd debugging.

Leveraging Software Repositories and Artifacts in Debugging Liang et al. [33] used generic bug patterns to identify project-specific bugs and used them to detect other similar bugs that exhibited the same pattern. Jiang et al. [59] detected clone related bugs by using control dependency graph to compare the consistencies between the clones. Inconsistencies in the control flow were treated as clone related bugs. Kim et al. [55] inspected a large volume of human-written patches for debugging to uncover bug patterns for automatic patching. Ashok et al. [60] searched through Microsoft repositories for previous similar debugging information to aid developers in fixing their code. Hartmann et al. [65] proposed detecting bugs based on code revisions of buggy code and fix code. Their evaluation was based on examples (buggy and fix code-pairs) from a debugging textbook. Mujumdar et al. [66] proposed using changes in unit test cases to identify and fix bugs. The original unit test case represented a suspected bug and the latter unit test case represented the fixed bug. Several of the work mentioned [65] [66] require manual intervention to input debugging explanation while others have limited debugging explanation. Gu et al. [67] used bug execution traces from failed JUnit tests to populate a bug database for detecting bugs in the target projects. They also instrumented the target’s code using ASM Java bytecode for the detection. Zimmermann et al. [68] studied the correlation between code complexity metrics of projects and bug history from post release projects (Microsoft Windows and Eclipse) to predict defects for debugging. Our work here differs in the use of the repository for debugging. We leverage the crowd knowledge where the crowd’s solutions and explanations are presented to developers. Also, our technique does not require instrumentation to the target’s code.

7. LIMITATIONS AND THREATS TO VALIDITY

We identify the following limitations and threats to validity of our experiment.

Type Insensitive In our approach, we do not consider the variable type due to partial code limitation in SO. However, as it is infeasible to experiment on the entire suite of existing software projects, it might be possible that the detected code fragments have type mismatch and are missed by developers in other software projects.

Code Terms outside Code Blocks We consider only code terms that are embedded inside code blocks and SO title as we observed that many posts have such characteristics. Identifying code terms inside code blocks creates a higher likelihood of extracting bona fide code terms. However, it is possible that developers may present some code terms in natural texts in a SO question and SO answer which we may have missed identifying them.

Post Filtering Although we did not encounter any missed defective code fragments in our observation during the experiment, it is not representative and defective code fragments may exist within these filtered warnings in other subjects. Furthermore, we may miss some real defective code fragments in checking the filtered warnings due to human errors.

Evaluation of warnings from Static Analysis tools Our manual evaluation on the sampled warnings generated from Static Analysis tools might be biased. We mitigate this threat by having two independent human evaluators to evaluate the warnings, and rope in an additional human evaluator if there are differences. Where differences in opinions arises, discussion will take place amongst the three independent human evaluators to reach a consensus.

8. CONCLUSION AND FUTURE WORK

In this paper, we propose a novel crowd debugging technique that leverages the crowds’ knowledge to detect defective code fragments in developers’ source code and provide the crowds’ suggested solutions with explanations for developers. We used the code blocks from SO to perform crowd debugging in eight high quality and well-maintained projects. Our experiments show that our crowd debugging technique is promising and of practical use. It is able to detect a reasonable 189 warnings and of those, 171 (90.5%) were confirmed by developers. We further compared our results with existing static analysis tools such as FindBugs, JLint and PMD. FindBugs can detect six same bugs whereas JLint and PMD can detect none. 165 (96.5%) of the confirmed developers’ bugs are missed by these tools. Our crowd debugging technique has clear benefits. It is able to detect bugs that are invisible to existing tools and provide suggested solutions with comprehensive explanation from the crowd.

In future, we plan to investigate on developing automatic patching algorithms for the defective code fragments based on the crowd knowledge, transforming the crowd patterns as static analysis patterns, and categorizing the crowd patterns such as belonging to API Misuse or Language Misuse.

9. ACKNOWLEDGEMENT

We thank all reviewers for their invaluable feedback and Rui Zheng for his help in analysing warnings generated by the static analysis tools.
REFERENCES


[34] Pmd. http://pmd.sourceforge.net/


