2014 IEEE International Conference on Software Maintenance and Evolution

On the Effectiveness of Information Retrieval Based Bug Localization for C Programs

Ripon K. Saha* Julia Lawall† Sarfraz Khurshid* Dewayne E. Perry*
*The University of Texas at Austin, USA
†Inria/LIP6/UPMC/Sorbonne University, France
ripon@utexas.edu, Julia.Lawall@lip6.fr, {khurshid, perry}@ece.utexas.edu

***Abstract*—Localizing bugs is important, difficult, and expensive, especially for large software projects. To address this problem, information retrieval (IR) based bug localization has increasingly been used to suggest potential buggy files given a bug report. To date, researchers have proposed a number of IR techniques for bug localization and empirically evaluated them to understand their effectiveness. However, virtually all of the evaluations have been limited to the projects written in object-oriented programming languages, particularly Java. Therefore, the effectiveness of these techniques for other widely-used languages such as C is still unknown. In this paper, we create a benchmark dataset consisting of more than 7,500 bug reports from five popular C projects and rigorously evaluate our recently introduced IR-based bug localization tool using this dataset. Our results indicate that although the IR-relevant properties of C and Java programs are different, IR-based bug localization in C software at the file level is overall as effective as in Java software. However, we also find that the recent advance of using program structure information in performing bug localization gives less of a benefit for C software than for Java software.**

***Keywords*—Bug Localization, Information Retrieval, Search**

I. Introduction

Large, widely used software projects typically combine many modules with different, interrelated functionalities and involve many developers. In this environment, it is difficult for a user that encounters a bug to know precisely where the bug occurs. Even a developer who reads a bug report may not immediately know what files are relevant, particularly if the report does not relate to his own code. If the code relevant to a bug report is not immediately apparent, the report can be ignored for a long time, or can be assigned to the wrong developer, wasting developer time [1].

To ease the process of identifying the source code that is relevant to a particular bug report, a number of automated approaches have been developed using Information Retrieval (IR) techniques such as Latent Dirichlet Allocation (LDA) [2], Vector Space Model (VSM) [3], Latent Semantic Analysis (LSA) [4], Clustering [4], and various combinations. These techniques have low cost and rely only on information from the bug report and the source code, with no other external dependencies. Recently, we have proposed a new IR-based technique, BLUiR, which takes into account program structure, distinguishing between different kinds of terms in source code based on program constructs [5]. BLUiR outperforms previous techniques on standard Java benchmarks.

While previous studies have shown that these IR-based bug localization approaches give good results, a limitation of these studies is that they focus on software written in object-oriented languages, primarily Java. On the other hand, much of the most critical and widely used software, such as operating systems, compilers, and programming language runtime environments, is written in C. Indeed, as of May 2014, C is the most popular programming language according to the TIOBE programming language popularity index [6]. Nevertheless, there is a lack of an established dataset of large-scale, widely used C software, and a lack of easy-to-use tools for manipulating C code. Therefore, we yet do not know the efficiency of IR-based bug localization tools for C code. Most previous bug localization studies have also acknowledged this limitation [5], [7], [8].

In this paper, we perform a large-scale experiment to investigate the efficiency of IR-based bug localization for C systems. To this end, we have created a dataset consisting of more than 7,500 bug reports from five popular C projects, and tested BLUiR on this dataset. We focus on the following research questions:

- RQ1. How do the IR-related properties of C software compare to those of Java software?
- RQ2,3. How does the accuracy of bug localization compare between C and Java software, at the file level (RQ2) and at the function level (RQ3)?
- RQ4. How does the use of English words in software affect the accuracy of C and Java bug localization?
- RQ5. How do preprocessor directives and macros in C code affect the accuracy of bug localization?
- RQ6. How much do the different structural elements of C and Java code contribute to the accuracy of bug localization and how does this contribution vary between C and Java code?

Our results show that:

- While structured IR-based bug localization gives comparable accuracy for C code and for Java code, the benefit for C code over language-independent IR-based bug localization is less than for Java code.
- The rate of English words in methods and identifiers differs greatly between C code and Java code; the fact that IR-based bug localization gives good results on both suggests that the rate of English words is not a good predictor of bug localization success across programming languages. However, we did find that for C programs, there is a correlation between the use of English words in the code and success of bug localization.

1063-6773/14 $31.00 © 2014 IEEE
DOI 10.1109/ICSME.2014.38

IEEE computer society

- Adequate parsing technology exists such that macros are not a major obstacle to IR-based bug localization.
- Bug localization for both C and Java code mostly relies on similar information: names of defined methods/functions and names of referenced identifiers. Bug localization for Java also benefits from the name of the defined class, while the C counterpart, *i.e.*, the file name, provides less information.

Our contributions include: 1) a dataset consisting of more than 7500 bug reports with their location in the source code at file level and function level for C programs, 2) a prototype to localize bugs in C systems; and 3) more generalizable results on the efficiency of IR-based bug localization.

The rest of this paper is organized as follows. Section II reviews our previous work on Java code. Section III describes the tools that we have developed to carry out our comparative study. Section IV presents our proposed benchmark and the metrics we use. Section V presents the results for our research questions. Section VI considers threats to validity. Finally, Section VII presents related work and Section VIII concludes.

II. Background

We first discuss how IR-based bug localization finds buggy files for a given bug report. We also briefly describe BLUiR, our recently introduced IR-based bug localization tool [5], which is currently one of the most accurate such tools available.

A. IR-based Bug Localization

When a developer, tester or user submits a bug report, generally they write a brief summary of the bug and a more detailed description of the buggy behavior. The basic assumption of IR-based bug localization is that some terms in a given bug report will be found in the source files that need to be fixed. Therefore, in IR-based bug localization, a software project's source code files are considered as a *document collection* and a bug report is considered as a search *query*. Finding candidate files that should be fixed to eliminate the bug then reduces to standard IR ranking of documents (source files) based on their estimated relevance to the query (bug report).

Generally an IR system comprises three major steps: text preprocessing, indexing, and retrieval. Preprocessing involves text normalization, stop-word removal, and stemming. Text normalization removes punctuation, performs case-folding, and tokenizes terms. Stop-word removal removes frequently used terms such as prepositions, articles, etc., in order to improve efficiency and reduce spurious matches. Finally, stemming conflates variants of the same term (e.g., go, going, gone) to improve term matching between the query and the document. Then, the documents are indexed for fast retrieval. Once indexed, queries are submitted to the search engine, which returns a ranked list of documents in response. Finally, the search engine is evaluated by measuring the quality of its output ranked list relative to each user's input query. For a broad overview of IR, please refer to the book by Manning [9].

The better an IR system can interpret the bug report and source files, the more accurately it is expected to highly rank the source files to be fixed. While deep semantics remain elusive, shallow matching often works quite well, in part because developers tend to embed semantic clues in names.

B. BLUiR

In recent work, we have introduced an IR-based bug localization tool, BLUiR, which is built upon an existing, highly-tuned, open source IR toolkit, Indri [10]. BLUiR has three advantages over existing IR-based bug localization approaches. First, it uses one of the best variants of TF.IDF formulation based upon the well-established BM25 (Okapi) model as the underlying retrieval model [11]. This model has been rigorously evaluated over a decade of use in IR and found to be very effective. Second, BLUiR uses Indri's indexing and retrieval system, which is highly efficient. Third, BLUiR has three modes of operation: *programming language independent* retrieval, *flat-text* retrieval, and *structured* retrieval. In each mode, BLUiR characterizes a bug report in terms of its individual words. However, preparation of documents varies depending on a given mode. For language-independent retrieval, BLUiR uses simple text processing (instead of parsing) to prepare source code for retrieval. This includes splitting terms based on CamelCase and underscores, and removing numbers and mathematical operators. Flat-text retrieval mode characterizes a source code file in terms of its class name, method names, identifier names, and comments, but does not distinguish between them. Structured retrieval mode additionally distinguishes between the summary and description in bug reports, and between the above four different kinds of terms in the source code. To exploit all of these different types of query and document representations, BLUiR structured retrieval performs a separate search for each of the eight combinations of query representation and document term type. Then, BLUiR sums the similarity scores across all eight searches to rank program files. More details are available in our previous work [5].

Based on an experiment using Zhou et al.'s dataset [3] containing more than 3,400 bug reports, we showed that BLUiR's flat-text retrieval already exceeded the accuracy of the existing best available tool. Adding structural modeling significantly improves accuracy further. In this paper, we adapt BLUiR for C programs.

III. Methodology

We now describe the methodology that we use to set up our experiments. This includes creating a large-scale dataset for C programs and adapting BLUiR for C code.

A. Creating a Dataset

To evaluate an IR-based bug localization tool retrospectively on a software project, we need to have the project's bug reports, program source code, and a means of identifying the files that were eventually fixed for the given bugs. Although getting bug reports and source code for various open source projects is fairly straightforward, determining the fixed files for a given bug is more challenging, since typically the bug tracking system and the version control system are independent of each other. Although there are many commits where developers indicate that a bug has been fixed, projects vary in the degree to which a reference to the bug tracker is provided.

Furthermore, different projects have different conventions for how bug tracker references are indicated. We now describe how we map bug reports to the commits and to the affected code, at both the file and the function level.

1) At the File Level: Most of the software projects in our dataset, presented in Section IV, use `git` for version control and `Bugzilla` for bug tracking. Git commit messages are free form, and thus developers may reference bug reports in any manner. To determine how the developers of a given project typically refer to bug reports, we first searched through all the commit messages for the keywords `bug`, `issue`, and `fix`. If we found any of them, we then searched for any numbers, that could be bug numbers. Then, we manually analyzed a number of commits selected in this manner from each system. This analysis revealed some common patterns, including a complete Bugzilla URL for the Linux kernel and the keyword `PR` followed by a bug number for GDB and GCC. For one of our considered projects, WineHQ, however, the above process gave no results. We thus consulted with a developer from the WineHQ community who informed us that in this community the convention is for the bug report to refer back to the commit, rather than the commit referring to the bug report. Indeed, in the WineHQ Bugzilla, there is a dedicated field for a git commit id. However, many of these fields are empty since the field is not required. After identifying the bug fixing commits, we extracted the names of the files that were changed and stored the bug report id and corresponding changed files in a JSON file for each project.

One of our considered projects, Python, uses `mercurial` rather than `git` and uses a dedicated bug tracking system rather than Bugzilla. Nevertheless, the process is essentially the same. By following the above process, we have found that Python bug report numbers are indicated by # followed by a number in the mercurial commit messages.

2) At Function Level: To construct the dataset at function level, we need to know the name of the function in which each bug fix occurs. For this, we use the command `git show -U0` to list the differences between the state of each file before and after the bug fix. `git show` produces the result in "unified diff format" [12], which normally shows the function header preceding each hunk, as illustrated by the following:

```
@@ -71,6 +71,17 @@ static int acpi_sleep_prepare(u32
acpi_state)
```

The `-U0` option furthermore tells `git show` to use no context information, which reduces the chance that a hunk will cross function boundaries. This approach, however, is still not completely reliable, *e.g.*, producing a recent label rather than a function header. We consider only those cases where the hunk header contains an open parenthesis, which has a high probability of indicating a function name.

3) Collecting information from bug reports: From the mapping we created in step 1, we have bug identifiers. Then, we use the Bugzilla Java API to download the summary and description for the bug report associated with each bug identifier. For Python, which does not use Bugzilla, we download the corresponding page from the dedicated repository[1] and parse it in an ad hoc manner to extract the summary and description. In each case, the description contains only the original report text, and does not contain any subsequent discussions, subsequently proposed patches, lists of fixed files, etc.

[1]http://bugs.python.org/

B. Adapting BLUiR

Our experiment uses BLUiR for C code. However, since BLUiR was designed for Java code, we have had to adapt it for experiments involving the C programming language.

1) Collecting information from C code: From the C code, we need to obtain the names of the defined functions and the identifiers used in each file. For this, we must parse the C code. A challenge in parsing C code is that such code may use C preprocessor directives, to include header files, to express conditional compilation and to use macros. One strategy would be to apply the C preprocessor before processing, resulting in a file that conforms to the standard C grammar. This approach, however, has numerous disadvantages. It would duplicate commonly used header files in every C file that uses them, which would explode the code size and could dilute the information that is relevant to bug localization. Furthermore, preprocessing the code would eliminate macro names, which are often more informative than the code that they expand into. In the case of software in which variability is expressed using conditional compilation, it would result in discarding the code that does not correspond to a chosen configuration. Finally, in the case of the Linux kernel, we have found that the result of preprocessing is so large that collecting and processing the relevant terms from the expanded code is impractical, in terms of both computing time and disk usage.

To avoid these problems, we use a parser for C code, developed as part of the program matching and transformation tool Coccinelle [13], that does not require preliminary processing by the C preprocessor. Instead, the Coccinelle parser makes an effort to parse macro references, to parse around conditional compilation directives, and to parse other preprocessor directives, such as `#ifdef` and `#define` directly [14]. When parsing fails, the parser recovers at the next top-level program unit, *e.g.*, function, variable, or type definition, thus minimizing the impact of the failure. The Coccinelle parser also has the ability to give feedback to the user about the most common parsing problems. Typically, these problems can be solved by providing a few artificial macro definitions in a configuration file. This configuration file typically needs to be created only once per software project, for use across multiple versions, as in our experience the set of problematic macros changes rarely. Except where noted, all of our experiments use these dedicated macro definition files whenever parsing is required. We examine these files in more detail in Section V-E.

Our extension of BLUiR, built on the Coccinelle C parser, collects function names, identifiers, and words appearing in comments. Function names and identifiers are collected both in their entirety and are split at underscores and according to Camel Casing. Identifiers are collected from variable names, function names, type names, structure field names, function parameter names, and goto label names.

2) Retrieval: As was discussed in Section II-B, BLUiR supports *language-independent* retrieval, *flat-text* retrieval and *structured* retrieval. Since language-independent and flat-text retrieval are not concerned with distinguishing terms, and differ

only in the strategy for extracting them, retrieval is the same for both C and Java code. However, since structured retrieval distinguishes between different types of terms based on program constructs (class, methods, variables, and comments) we have to classify C constructs within these categories. We consider C file names to be equivalent to Java class names and C function names to be equivalent to Java method names. Identifier names and comments are the same for both languages. Then, we apply the same underlying technique to compute the similarity score between a query and a collection of documents, to rank C files as described in Section II-B.

IV. Datasets and Metrics

Because there is no standard dataset for C software, we had to create one. In this section, we motivate our choice of software, and present the metrics that we use to compare bug localization for C code with bug localization for Java code.

A. Datasets

Generally, bug localization is more useful for large-scale systems, where developers could have trouble localizing bugs manually. Therefore, we have selected a number of C projects that are well known and large, that have a long development history, and that have a dedicated bug tracking system containing a large number of bug reports. In this way, we have chosen five open source projects, ordered below from smallest to largest in terms of the number of lines of code:

- Python 3.4.0: The runtime of the Python programming language.[2]
- GDB 7.7: A debugger for programs written in C, C++, and many other programming languages.[3]
- WineHQ 1.6.2: A compatibility layer, making it possible to run Windows applications on POSIX compliant operating systems.[4]
- GCC 4.9.0: A compiler for programs written in C, C++, and many other programming languages.[5]
- Linux Kernel 3.14: The kernel of the Linux operating system.[6]

Table I presents some properties of these projects. For GCC we have created two versions since GCC has a large testsuite that consists of more than 20,000 files in form of C code. GCC_NT (no test) represents the version with these test cases removed. For comparison with Java, we use Zhou et al.'s [3] dataset, presented in Table II, which we have used in previous work on BLUiR. We have not included the project ZXing from this dataset, because this project has only 20 bug reports.

In terms of size, the C projects fall into three groups: Python at under 400,000 lines of code, GDB, WineHQ, and GCC at around 2 million lines of code, and the Linux kernel at over 11 million lines of code. For both C and Java software, we computed the size using David Wheeler's SLOCcount,[7] which includes only the number of lines of C or Java code, respectively, not whitespace, comments, or code written in other languages. The C projects are substantially larger than the Java projects, with the second smallest C project, GDB, being 25% larger than the largest Java project, Eclipse.

TABLE I. Dataset Description for C Systems

Subjects Systems	Oldest patch	SLOC	For File Level		For Function Level	
			#Bugs	#Files	#Bugs	#Functions
Python	1990	380K	3,407	488	-	-
GDB	1988	1,982K	195	2,655	177	41,298
WineHQ	1993	2,340K	2,350	2,815	2,218	89,430
GCC	1988	2,571K	216	22,678	193	75,746
GCC_NT	-	2,062K	-	2,473	-	40,684
Linux kernel	2005	11,829K	1,548	19,853	1,178	347,057

TABLE II. Dataset Description for Java Systems

Project	Description	Oldest Patch	SLOC	#Bugs	#Files
SWT 3.1	Widget toolkit for Java	2004	78K	98	484
AspectJ	Aspect-oriented extension to Java	2002	323K	286	6485
Eclipse 3.1	Popular IDE for Java	2002	1,579K	3,075	12,863

In terms of development history (column **Oldest patch**, Table I) all of our C projects date from around 1990. The current git repository of the Linux kernel, however, only contains commits going back to 2005, when git was adopted by the Linux kernel developers. Other projects imported their previous version control history into git, and thus we have commits from a wider time span for these projects.

Finally, the number of bug reports available for the different C projects varies widely, but remains within the same order of magnitude as the number of bug reports available for the different Java projects. We have followed the procedure described in Section III-A for identifying bug reports that can be linked to commits. We take only the bug reports for which the fixes touch at least one C file, and for which at least one of the affected files still exists in the considered version of the software. At the function level, we have only considered the bug reports for which at least one name of an affected function can be identified from the associated patch, as described in Section III-A. For Python, we have no function-level information, as Python uses `mercurial`, whose patch viewer does not make function header information available.

B. Evaluation Metrics

To evaluate the efficiency of BLUiR for C systems, we calculate three metrics: Recall at Top N, Mean Average Precision, and Mean Reciprocal Rank. These metrics have been extensively used in prior IR-based bug localization research [3], [5].

Recall at Top N: To calculate this metric, for each bug report, we first rank all the source code files and then take the top N files to see if there is any file containing a bug related to the report present in the list. Then, we calculate the proportion (basically *recall*) of the bug reports for which we located the bug successfully. We calculate Recall at Top 1, Top 5, and Top 10, as done in previous studies [3], [5].

Mean Average Precision (MAP): While *Recall at Top N* emphasizes locating a single buggy file, MAP takes all the buggy files associated with a given bug report into account.

[2] https://www.python.org/, http://hg.python.org/cpython
[3] http://www.sourceware.org/gdb/, git://sourceware.org/git/binutils-gdb.git
[4] http://www.winehq.org, git://source.winehq.org/git/wine.git/
[5] http://gcc.gnu.org/, git://gcc.gnu.org/git/gcc.git
[6] https://www.kernel.org/, git://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git
[7] http://www.dwheeler.com/sloccount/

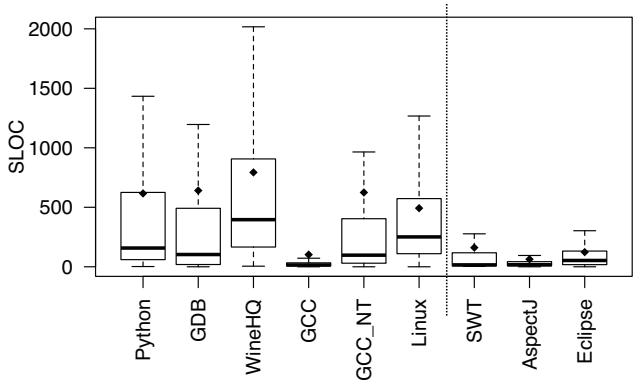

Fig. 1. Comparison of File Size

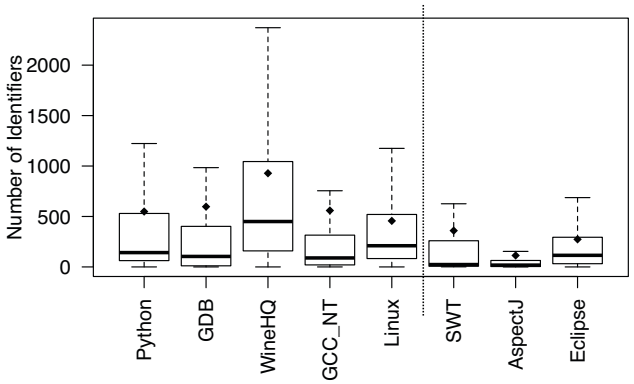

Fig. 2. Comparison of Number of Terms

The Average Precision of a single query is computed as:

$$AP = \sum_{k=1}^{M} \frac{P(k) \times pos(k)}{number\ of\ positive\ instances} \quad (1)$$

where M is the number of source files in the ranked list, k is the rank, and $pos(k)$ is a binary (0 or 1) indicator of whether or not the item at rank k is a buggy file. $P(k)$ is the precision for top k files. The MAP for a set of queries is simply the mean of the AP values for all queries. A higher value is better.

Mean Reciprocal Rank (MRR): The reciprocal rank for a query is the inverse rank of the first relevant document found. MRR is the reciprocal rank averaged over all queries:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i} \quad (2)$$

V. Results

We now present our experimental results. First, we consider some properties of the C and Java software that may affect the applicability of IR-based bug localization. Then, we consider our research questions, as defined in Section I, related to the accuracy of various BLUiR-based approaches and to several details of the bug localization process.

A. RQ1: IR-related properties of C and Java software

We first investigate two IR-relevant properties: file length and nature of terms used in C and Java programs, which have an impact on the results of any IR system. If these properties of C and Java software are different, then the effectiveness of IR-based bug localization may be different as well.

File Size. Figure 1 displays the file size of each software project in terms of number of lines of code without comments (SLOC). We observe that the SLOC distribution of GCC files is completely different from that of the other projects. As we discussed in Section IV-A, the source code of GCC contains a large suite of test cases, amounting to more than 20,000 small .c files. As we expect that few bug reports relate to bugs in test cases, we also include in Figure 1 information for GCC with all test cases excluded (GCC_NT).

Figure 1 shows that the median sizes of C files, indicated by the thickest horizontal line, vary from 97 (GCC_NT) to 396 (WineHQ) SLOC, whereas they vary from 16 (SWT) to 53 (Eclipse) SLOC for Java files. However, for the C projects, the average, marked by the diamond, is typically much higher than the median, indicating that the file sizes are highly skewed. The average size of the C files varies from 492 (Linux) to 793 (WineHQ) SLOC, whereas it varies from 63 (AspectJ) to 161 (SWT) SLOC for the Java projects. Therefore, overall, C files are, on average, substantially larger than Java files, for our considered projects.

As an alternate measure of size, we also investigate the total number of terms in the source code that would be actually used for IR, as presented in Figure 2. We see that the number of terms present in the C files is also considerably higher than that of Java files. The average number of terms in the C files varies from 456 (Linux) to 928 (WineHQ), whereas for Java it varies from 114 (AspectJ) to 360 (SWT).

Terms in Source Code.. Bug reports are generally written in natural English. IR-based bug localization generally focuses on identifier (class, method, and variable) names and comments, because these are the places where developers can use natural English. In object-oriented programming languages, developers are strongly encouraged to use meaningful words in identifier names. For example, the Eclipse Foundation has very specific naming conventions.[8] Therefore, IR-based bug localization is expected to work well for Java projects. Since the C programming language is used by a different group of people and is generally used for different types of software (e.g. systems software rather than applications) than Java, the programming style of C may be considerably different.

Our study of the use of English words focuses on method and identifier names, omitting comments on the assumption that comments almost always contain natural English text, regardless of the programming language. To have the greatest chance of finding English words, we split method and identifier names according to the conventions of CamelCase and additionally split C names at underscores (_), following the conventions commonly used in our software projects. Finally, we exclude the words `get` and `set` when counting English

[8] http://wiki.eclipse.org/Naming_Conventions

TABLE III. PRESENCE OF ENGLISH WORDS IN SOURCE CODE

Term Type	Function/Method-Terms*		Identifier-Terms	
	Actual %	Unique %	Actual %	Unique %
Python	66%	44%	67%	33%
GDB	67%	32%	59%	18%
WineHQ	72%	20%	59%	12%
GCC	71%	30%	69%	21%
GCC_NT	76%	46%	72%	25%
Linux	59%	20%	59%	10%
SWT	95%	85%	84%	48%
AspectJ	92%	67%	91%	52%
Eclipse	97%	75%	94%	48%

words in Java programs, since these words are very frequent, due to the use of getter and setter methods. In our C projects, we have found that developers use underscores in 88%-97% of method names and 43%-55% of identifier names, and in our Java projects, we have found that developers use CamelCase in 62%-80% of method names and 38%-46% of identifier names.

Once the identifier names have been split into tokens (or terms), we match each result against a comprehensive list of 354,983 English words.[9] Then, we calculate the percentage of terms that are found in the list of English words. Since a term can appear multiple times in the code, we also calculate the percentage of unique terms that are also English words. That is, if E is the set of terms that are found in the dictionary, with all duplicates removed, and T is the complete set of split words, again with duplicates removed, we calculate the unique word percentage as $\frac{|E|}{|T|} \times 100$. For function (or method for Java) names, T is the set of split words obtained from the names of defined functions, while for identifier names, T is the set of split terms obtained from all identifier names, including the names of called functions. From the results, we see that developers tend to use English terms in both function and identifier names. However, the higher unique percentages for function names show that there are more non-English words in identifier names than function names. From the results we also see that the presence of English words in Java programs is considerably higher than in C programs, both in terms of actual and unique percentages.

Therefore, the overall results show that C and Java are not only different due to programming paradigms (procedural vs. OOP) but also different from IR perspectives. Our next research questions investigate how IR-based bug localization, which has previously mostly been evaluated for Java programs, performs in practice for C programs.

B. RQ2: Accuracy of Bug Localization at the File Level

Figures 3 and 4 show the accuracy of bug localization for the C and Java projects, using language-independent retrieval, flat-text retrieval, and structured retrieval, in terms of Recall at Top 1, Top 5, and Top 10, and MAP and MRR. In Figure 3, the Recall at Top 5 and Recall at Top 10 bars represent the increase in recall as compared to taking into account the Top 1 or Top 5 files, respectively. We first assess the overall results, and then compare the results for C and Java projects as the number of files considered from the top of the ranked list changes, and in terms of the bug localization strategy.

[9]http://www.infochimps.com

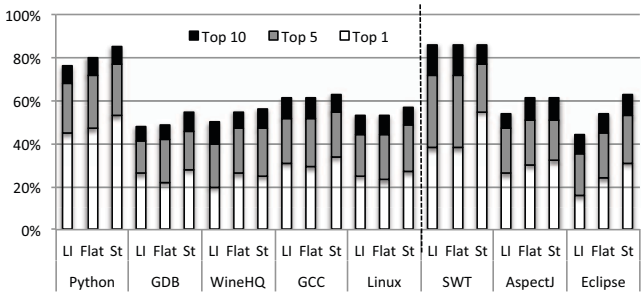

Fig. 3. Comparison of Recall at Top N for Different Strategies

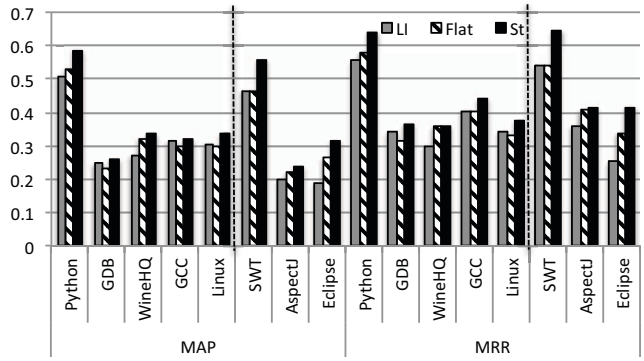

Fig. 4. Comparison of MAP and MRR for Different Strategies

For all but the smallest projects (Python and SWT), whether C or Java, bug localization gives roughly the same accuracy, with Recall at Top 1 values of 20-34% for C and 16-32% for Java, and Recall at Top 10 values of 48-63% for C and 44-63% for Java. We see the same similarity in the MRR scores. The C projects, however, have higher MAP scores than the Java projects, ranging from 0.249 to 0.337 for C (with 0.586 for Python) and from 0.190 to 0.316 for Java (with 0.557 for SWT). Thus, bug localization is more successful in finding all of the files that should be changed for the C projects.

Next, we consider the impact on accuracy of considering more files in the ranked list, by comparing the result for Recall at Top 1 with the results for Recall at Top 5. For language-independent retrieval, considering more files gives less of an improvement for the C projects (51-76%, except for WineHQ, where there is a 100% improvement) than for the Java projects, where the improvement is always over 80%. On the other hand, for flat-text retrieval, the improvement is about the same, being 70-90% for all projects except Python. For structured retrieval, considering more results gives more of an improvement for C (62-88% for all C projects except Python) than for Java (59-71% for all Java projects except SWT). The smallest projects, Python and SWT, achieve an improvement of 45% and 40%, respectively, for Recall at Top 5 as compared to Recall at Top 1, but their Recall at Top 1 rates were already much higher, at 53% and 55%, respectively, than for the other projects.

Finally, we consider the improvement provided by flat-text retrieval and structured retrieval as compared to language-independent retrieval. In the case of flat-text retrieval, there is again a major difference between C and Java projects. For three

of the C projects, GDB, GCC, and Linux, flat-text retrieval gives a *worse* Recall at Top 1, by up to 15% for GDB, while for Java, the result is always the same (SWT) or better. Indeed, flat-text retrieval increases Recall at Top 1 by 50% for Eclipse as compared to language-independent retrieval. However, the results for flat-text retrieval for C projects are not all negative; for WineHQ, which has the lowest accuracy for language-independent retrieval, at 20%, flat-text retrieval gives a 30% improvement, resulting in an accuracy that is comparable to that of the other larger projects. Structured retrieval gives an improvement over language-independent retrieval for all projects. Nevertheless, the improvement is quite small for some C projects: from 26% to 28% for GDB, from 31% to 34% for GCC, and from 25% to 27% for Linux. In these cases, structured retrieval mostly just reverses the losses observed for flat-text retrieval. Indeed, WineHQ, which benefited most from flat-text retrieval, obtains a slightly worse Recall at Top 1 result with structured retrieval than with flat-text retrieval, from 26% to 25%. These results contrast with the results for Java, where structured retrieval gives a substantial improvement in accuracy, including an improvement of 94% in Recall at Top 1 for Eclipse as compared to language-independent retrieval.

In summary, we find that for both C and Java, the accuracy is roughly similar, except in the case of MAR, where bug localization is more successful for the C projects. We also find that the impact of taking into account more reported files varies between C and Java, depending on the retrieval strategy, and that structured retrieval provides less benefit for C projects.

C. RQ3: Accuracy of Bug Localization at the Function Level

Thus far, our results have been expressed at the file level. We have seen in Section V-A, however, that for our projects, the C files are much larger, on average, than the Java files. While knowing the affected file may permit the developer to hone in directly on the problem, in the worst case, the C developer has on average, *e.g.*, approximately 2500 to 4000 lines of code to inspect when considering the Recall at Top 5 results, while the Java developer has only at worst on average 300 to 800 lines of code to inspect. Thus, we consider whether BLUiR can be effective at the function level on C projects, to further narrow down the search space of developers. Over all of the C projects, the average function size varies from 28 to 42 lines of code.[10] Thus, Recall at Top 5 at the function level for our C projects would be roughly comparable to Recall at Top 1 on average for our Java projects at the file level based on the number of lines to be inspected.

To investigate the accuracy of BLUiR at the function level, we constructed a document collection containing one document per function and applied BLUiR to it. Our results (in Table IV) show that the accuracy of BLUiR is much lower at the function level than at the file level. The recall in Top 1 ranges from 7% to 11%, whereas the recall in Top 10 ranges from 21% to 27%. We think this result is not surprising since an individual function provides much less information than a complete file. Furthermore, retrieval at the function level can be more expensive than retrieval at the file level since the number of functions can be much greater than the number of files. For example, in Linux Kernel, BLUiR ranks all the source code files in 5 seconds on average for a given bug report, whereas it takes 55 seconds per query at function level, on a machine having an Intel Core i7 @ 3.50GHz processor and 16GB memory.

TABLE IV. FUNCTION-LEVEL RETRIEVAL ACCURACY

Project	Top 1	Top 5	Top 10	MAP	MRR
GDB	7%	21%	27%	0.073	0.145
WineHQ	8%	16%	21%	0.085	0.122
GCC	9%	18%	25%	0.081	0.144
Linux	11%	19%	24%	0.127	0.159

To improve the performance of BLUiR at the function level, we then tried a two-step approach. First, we ran BLUiR at file level and took the top k files for function retrieval. From our previous results, we found that for all projects BLUiR can localize more than 80% of bugs within the Top 100 files. Thus, if we consider only the functions from these files, the number of functions for retrieval would be reduced a lot, without losing the buggy functions for more than 80% of the bugs. Therefore, reducing the candidate functions in this way should reduce the retrieval time but have little impact on accuracy. Our results show that this alternative approach indeed reduced the retrieval time considerably, while maintaining almost the same accuracy as considering all functions. The function-level retrieval now requires only a fraction of a second after selecting the Top 100 files, reducing the total time from 55 seconds to 5 seconds.

D. RQ4: Impact of the use of English Words

We next investigate whether there is any relationship between the use of English words in method and identifier names and the accuracy of BLUiR. We noted previously that the Java projects use a substantially higher rate of English words than the C projects. Nevertheless, particularly in terms of Recall at Top 1, MAP, and MRR, Figures 3 and 4 show that we get equal or better results for the C projects than for the Java projects. Among the C projects, we have the highest rates of unique method names and unique identifier names for Python, and we obtain the highest accuracy for this project as well. We also observe that the Recall at Top N gradually increases with the increase in the unique percentages of English words in method names and identifiers, except for GDB. To show that this correlation is statistically strong, we calculated Pearson product-moment correlation coefficient between different accuracies and unique percentages. The correlation coefficient for method unique percentage and Recall at Top 1, and identifier unique percentage and Recall at Top 1 are 0.92 and 0.95, respectively. We also get 0.84 and 0.92 for Recall at Top 5. Figure 5 presents this trend with scatter plots and best fit regression lines computed by R.[11] But we do not observe the same trend in the case of Java, where the project with the highest accuracy, SWT, has the highest rate of unique method names, but does not have the highest rate of unique identifiers. We also have not found any systematic relationship for other Java projects. Therefore our overall results show that the greater usage of English words only increases the accuracy of bug localization for C projects.

[10] Function size is computed from the difference between the line number of the last line of the function and the line number of the first line, and thus may include lines containing only comments or whitespace.

[11] http://www.r-project.org

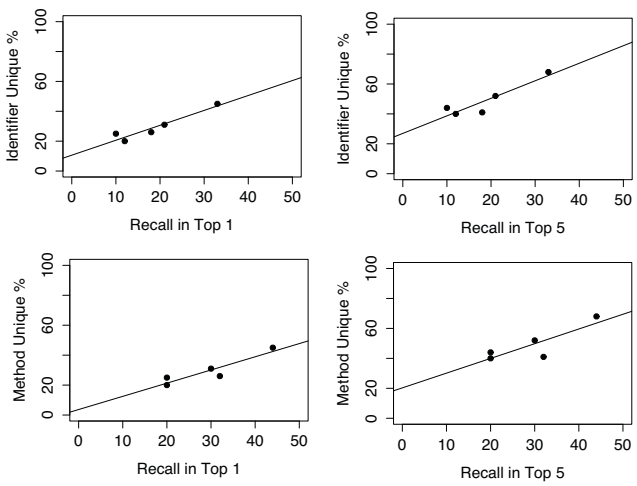

Fig. 5. Correlation between Percentages of English Words and Accuracy

TABLE V. PROPERTIES OF MACRO DEFINITION FILES

Project	No macro definitions: Parse time	No macro definitions: Success rate	Custom macro definitions: Macro definitions	Custom macro definitions: Parse time	Custom macro definitions: Success rate
Python	51s	62%	24	39s	88%
GDB	8m 45s	78%	18	8m 34s	84%
WineHQ	6m 51s	70%	13	6m 3s	89%
GCC_NT	6m 14s	59%	41	4m 26s	77%
Linux	24m 43s	61%	234	19m 51s	84%

E. Impact of tool features

In this section, we study our results in more detail, to better understand the relationship between the performance of IR-based bug localization on C and on Java programs. First, we consider the effect of macros, which complicate the processing of C code and which are not present in Java code. Then, we consider the contribution of each kind of information used by structured retrieval to the final result.

RQ5: Effect of Macros. In Section III-B1, we explained why the presence of preprocessor directives and macros in C code can affect the analysis results. While our C parser tries to cope with unknown macro uses, in some cases, its heuristics are not successful, and some top-level variable or function definitions are not taken into account, potentially reducing the amount of information that is available. Better results can be obtained by providing a configuration file giving definitions for a few macros that are difficult to parse, based on feedback from the parser about common parsing problems. The time required for creatig this configuration file mostly depends on the parsing time.

Table V shows the parsing time on our Intel Core I7 machine when no macro definitions are available, the percentage of files that are entirely successfully parsed in this case, the number of macro definitions in our customized macro definition file for each project, the parsing time when these definitions are available, and the percentage of files that are entirely successfully parsed in this case. For the Linux kernel, we use the existing default macro configuration file of Coccinelle [13], which targets Linux kernel code.

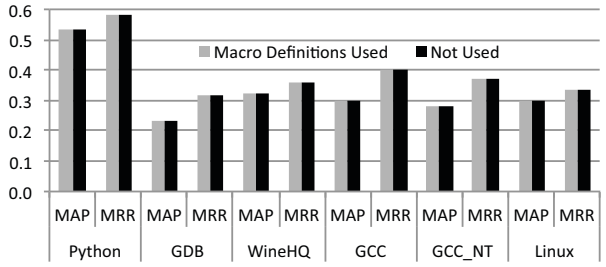

Fig. 6. Effect of Macro Definitions on Flat-Text Accuracy

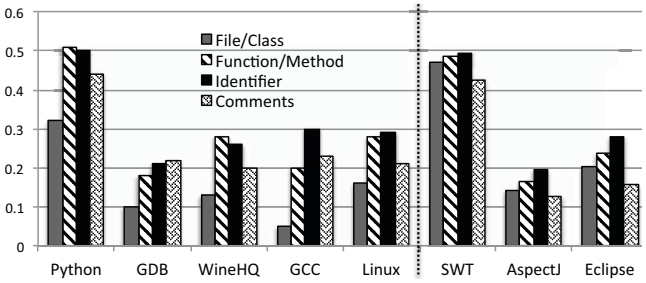

Fig. 7. Mean Average Precision (MAP) for different kinds of program terms

Figure 6 compares the accuracy of BLUiR when we use the project-specific macro definition and when we do not use them. Results show that even though the case without specific macro definitions results in *e.g.*, only 59% of the files being successfully parsed in their entirety for GCC, the results are essentially the same, with a difference of at most 0.002 for MAP and 0.003 for MRR. Indeed, our C parser recovers at the start of the next top-level definition that it is able to identify, and thus in practice a parse error in one part of a file has no impact on the parsing of the rest of the file. Thus, plenty of information is available for bug localization, and BLUiR is able to localize bugs despite the parse errors.

RQ6: Importance of Information Sources. To better understand the previous results, we investigate which kinds of program terms (file/class names, function/method names, variable names, and comments) are more important. To this end, we run BLUiR on each type of term separately. Since this produces four sets of results for each project, we present only MAP for conciseness. MAP takes all the buggy files into account, and thus is the most comprehensive metric.

The results show that Java class names are more important than C file names. For C programs, we see a large gap between the accuracy for file names and the accuracy for other terms for each project, whereas such gaps are small for Java programs. Also, we observe that although the number of method names is far smaller than the number of identifier names, method names carry a lot of information. For both C and Java programs, the MAP value based on only method names is very close to that of identifier names except for GCC. For some projects (Python, WineHQ, Linux Kernel, and all the Java projects), method names contribute more than comments. The overall results show that although all kinds of terms help localize bugs, for both languages, method/function names and identifier names are important for every project.

VI. Threats to Validity

This section discusses the validity and generalizability of our findings.

Construct Validity: We used two artifacts of a software repository: source code and bug reports, which are generally well understood. We have used three popular metrics: Recall at Top N, MAP, and MRR, which are standard in IR, have been used in previous IR-based bug localization studies, and are straightforward to compute.

Internal Validity: To create the benchmark for C projects, we have relied on the information in version histories and bug tracking systems. However, for some cases this information may be inaccurate or incomplete, which may affect our results. Indeed, for GDB, WineHQ, and GCC, which are about the same size, we have widely varying numbers of linked bug reports, which may indicate that the developers of some projects do not mention such links systematically.

We have used a single release for bug localization in each system. Bugs that were previously fixed are no longer present in that code, and for old bug reports, the code may have changed substantially since the bug was encountered. Ideally, for each bug report we would extract the version from when the bug was reported to get the actual buggy code. However, this approach is impractical for a large-scale experiment.

Most of our studied projects represent systems code rather than applications. Systems software may have its own set of development biases [15]. We may not capture concerns that are only present in software targeting other domains.

Like other IR-based bug localization studies, our results are intrinsically sensitive to the quality of the bug reports. An issue is the possible presence of "too well written" bug reports, *e.g.*, where a maintainer of the code has already solved the problem, and is using the bug repository to record his activities. Such reports could make bug localization unrealistically easy, as compared to reports from ordinary users, for which localization is actually needed. Indeed, Kochhar et al. [16] have found, in work concurrent with ours, that for three Java projects different from the ones considered here, around half of the bug reports contain the name of at least one of the classes that should be fixed, and that the sets of reports that contain the names of all of the classes that should be fixed have MAP scores 2.5-3 times higher than those that contain no such class names. In our dataset, we have found that 10%-19% of the bug reports of C projects contain the name of at least one file that was fixed, and likewise 5% to 29% for Java projects. If we ignore the extension (.c or .java) of the file name, the ranges vary from 25% to 29% for C projects and from 32% to 62% for Java projects. We furthermore observe that 19% of Python reports contain the name of at least one file that should be fixed, while only 10% of the WineHQ reports do, which may account for some of the difference in the success of bug localization on these two projects (see Figure 3). Nevertheless, it is hard to determine what proportion of these bug reports are actually bias in the dataset, since file names or class names may coincide with natural English words. We also found 34 Linux Kernel bug reports that contain the name of at least one file that was not fixed. Thus, file name information may not always make bug localization trivial. We leave a more detailed study of the kinds of information present in bug reports and how this information impacts the success of manual or automatic bug localization to future work.

Finally, our tools may contain errors. We have carefully inspected our code and rigorously tested it on a known dataset.

External Validity: We have used five C software projects and three Java software projects in our experiments. All are open source. Although, they are popular projects, our findings may not be generalizable to other open source projects or to closed source projects. However, to the best of our knowledge, this is the largest experiment for IR-based bug localization. The risk of insufficient generalization could be mitigated by expanding the benchmark to include more software projects (both open source and closed source). This will be explored in our future work.

VII. Related Work

The literature on finding bugs and other features of source code is enormous. We thus focus on related work on matching some form of user-provided query to regions of source code, as well as studies that compare results for C to results for Java.

Bug localization: IR-based bug localization techniques have recently gained attention from the software engineering research community. Researchers have proposed a number of retrieval techniques to improve the rank of buggy files for a given bug report query. Lukins et al. [2] use the Latent Dirichlet Allocation (LDA), a generative statistical model widely used for topic modeling, for bug localization. Rao et al. [4] compare a number of techniques such as the Unigram Model (UM), the Vector Space Model (VSM), the Latent Semantic Analysis Model (LSA), the Latent Dirichlet Allocation (LDA), the Cluster Based Document Model (CBDM), and various combinations to investigate their relative performance for bug localization. Based on their evaluation, they concluded that sophisticated models such as LSA, LDA, or CBDM are not necessarily better than simpler models such as UM or VSM.

Recent bug localization techniques go beyond traditional IR by using additional information from software repositories. Sisman and Kak [8] incorporate version histories in an IR model. Nguyen et al. [17] introduce BugScout, a topic model-based tool for narrowing the search space for buggy files. Zhou et al. [3] incorporate program file length and similar information into the TF.IDF term weighted VSM. BLUiR incorporates structural information into an IR model [5]. AmaLgam takes into account not only structure information, but also the set of files that have recently been subject to bug fixes and the set of files fixed by recent similar bug reports [18].

However, all of the above techniques have been evaluated on datasets containing object-oriented programs, particularly Java. Our focus is on whether techniques that work for object oriented languages also work for imperative languages. Thus, we evaluate IR-based bug localization with C programs.

Other IR problems and C code: Several other works have considered other kinds of localization problems for C code. Wang et al. [19] study the effectiveness of a wide range of information retrieval techniques on localizing concerns in Linux kernel source code. Rather than bug localization, they study the problem of feature localization, mapping a feature, expressed as a preprocessor flag, to the relevant source code,

defined as the function whose definition is somehow affected by the flag's value. Their experiment did not involve bug reports and was limited to the Linux Kernel. Poshyvanyk et al. [20] formulated the feature location problem as a decision-making problem in the presence of uncertainty and evaluated their approach by localizing bugs in Mozilla. Mozilla contains both C and C++ code, but only the C++ code was taken into account in the evaluation.

Marcus et al. [21] use an old version of the NCSA Mosaic web browser, written in C, to test their approach to concept location, where the goal is to find code relevant to a developer-provided code search request. Developer searches may have different textual properties than bug reports. They use latent semantic indexing (LSI), which is different than the TF-IDF based approach used by BLUiR. Finally, the source code size is small (95KLOC) and there is no comparison between C and Java.

C vs. Java: Lucia et al. [22] compare the result of spectrum-based fault localization, in which probable locations of faults are identified based on succeeding and failing execution traces, on C and Java programs, using a variety of metrics. They find that the results are overall better on C code than on Java code, and that the set of metrics that perform best is different in the two cases. Nevertheless, they consider a different kind of localization problem than the one considered here (execution trace based rather than IR-based), the C software projects considered are much smaller than the ones considered here, being at most 6,218 lines of code, and the issues of preprocessor directives and macros, which are critical to treating large C software projects, are not addressed.

VIII. Conclusion

In this paper, we have compared the results of IR-based bug localization on large, widely used C and Java software, thus giving a richer perspective on the effectiveness of bug localization than that provided by previous studies, which were primarily limited to Java. The main technical challenge in applying IR-based bug localization to real C projects is to cope with the use of C preprocessor directives and macros. We have shown that this issue can be addressed in a lightweight way using existing technology. Another main lesson of our work is that even though C developers use English words substantially less often in their method and identifier names than Java developers, IR-based bug localization can still be effective on C code, comparably to Java code. On the other hand, our considered C projects benefit less than our considered Java projects from taking into account information from program structure, an extension to IR-based bug localization that has been proposed in a number of recent techniques. This suggests that a greater understanding may be needed of the properties of bug reports and source code to make IR-based bug localization more effective in practice.

In future work, we will extend the experiment to more C and Java projects. We will also consider whether the insights obtained from examining C code can lead to better bug localization algorithms. A particular focus could be on improving bug localization at the function level for C programs, to mitigate the possible effect of the large size of C files.

Dataset. Our dataset of C projects is publicly available at https://utexas.box.com/icsme2014-dataset

Acknowledgements. This work was funded in part by the National Science Foundation (NSF Grant Nos. SRS-0820251 and CCF-0845628).

References

[1] R. K. Saha, S. Khurshid, and D. E. Perry, "An empirical study of long lived bugs," in *CSMR-WCRE*. IEEE, 2014, pp. 144–153.

[2] S. K. Lukins, N. A. Kraft, and L. H. Etzkorn, "Bug localization using latent Dirichlet allocation," *Information and Software Technology*, vol. 52, no. 9, pp. 972–990, 2010.

[3] J. Zhou, H. Zhang, and D. Lo, "Where should the bugs be fixed? - more accurate information retrieval-based bug localization based on bug reports," in *ICSE*, Zurich, Switzerland, 2012, pp. 14–24.

[4] S. Rao and A. Kak, "Retrieval from software libraries for bug localization: a comparative study of generic and composite text models," in *MSR*, 2011, pp. 43–52.

[5] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry, "Improving bug localization using structured information retrieval," in *ASE*, 2013, pp. 345–355.

[6] TIOBE Software, "TIOBE programming community index," May 2014, http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

[7] S. Davies, M. Roper, and M. Wood, "Using bug report similarity to enhance bug localisation," in *WCRE*, Kingston, ON, Canada, 2012, pp. 125–134.

[8] B. Sisman and A. Kak, "Incorporating version histories in information retrieval based bug localization," in *In MSR*, 2012, pp. 50–59.

[9] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[10] T. Strohman, D. Metzler, H. Turtle, and W. B. Croft, "Indri: A language model-based search engine for complex queries," in *Proceedings of the International Conference on Intelligent Analysis*, 2005, pp. 2–6.

[11] C. Zhai, "Notes on the lemur tfidf model (unpublished work)," Carnegie Mellon University, Tech. Rep., 2001.

[12] D. MacKenzie, P. Eggert, and R. Stallman, *Comparing and Merging Files With Gnu Diff and Patch*. Network Theory Ltd, Jan. 2003.

[13] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in Linux device drivers," in *EuroSys 2008*, Glasgow, Scotland, Mar. 2008, pp. 247–260.

[14] Y. Padioleau, "Parsing C/C++ code without pre-processing," in *International Conference on Compiler Construction (CC'09)*, York, UK, Mar. 2009, pp. 109–125.

[15] H. K. Wright, M. Kim, and D. E. Perry, "Validity concerns in software engineering research," in *FoSER*. ACM, 2010, pp. 411–414.

[16] P. S. Kochhar, Y. Tian, and D. Lo, "Potential biases in bug localization: Do they matter?" in *ASE*, Västerås, Sweden, 2014.

[17] A. T. Nguyen, T. T. Nguyen, J. Al-Kofahi, H. V. Nguyen, and T. Nguyen, "A topic-based approach for narrowing the search space of buggy files from a bug report," in *ASE*, 2011, pp. 263–272.

[18] S. Wang and D. Lo, "Version history, similar report, and structure: Putting them together for improved bug localization," in *ICPC*, Hyderabad, India, Jun. 2014.

[19] S. Wang, D. Lo, Z. Xing, and L. Jiang, "Concern localization using information retrieval: An empirical study on Linux kernel," in *WCRE*, Limerick, Ireland, Oct. 2011, pp. 92–96.

[20] D. Poshyvanyk, Y.-G. Gueheneuc, A. Marcus, G. Antoniol, and V. Rajlich, "Combining probabilistic ranking and latent semantic indexing for feature identification," in *ICPC*, Athens, Greece, 2006, pp. 137–148.

[21] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *WCRE*, Delft, The Netherlands, 2004, pp. 214–223.

[22] Lucia, D. Lo, L. Jiang, F. Thung, and A. Budi, "Extended comprehensive study of association measures for fault localization," *Journal of Software: Evolution and Process*, no. 26, pp. 172–219, 2014.