View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Institutional Knowledge at Singapore Management University

Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information Systems | School of Information Systems

7-2013

An empirical study of bugs in software build systems

Xin XIA

Xiaozhen ZHOU

David LO

Singapore Management University, davidlo@smu.edu.sg

Xiaoqiong ZHAO

DOI: https://doi.org/10.1109/QSIC.2013.60

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research

Part of the Software Engineering Commons

Citation

XIA, Xin; ZHOU, Xiaozhen; LO, David; and ZHAO, Xiaoqiong. An empirical study of bugs in software build systems. (2013). *2013 13th International Conference on Quality Software (QSIC 2013): Proceedings: 29-30 July 2013, Nanjing, China*. 200-203. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/2023

This Conference Proceeding Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email libIR@smu.edu.sg.

PAPER

An Empirical Study of Bugs in Software Build System

Xin XIA[†], **Xiaozhen ZHOU**[†], **David LO**[††], **Xiaoqiong ZHAO**[†], ***Nonmembers***, ***and*** **Ye WANG**[†††a)], ***Member***

SUMMARY A build system converts source code, libraries and other data into executable programs by orchestrating the execution of compilers and other tools. The whole building process is managed by a *software build system*, such as Make, Ant, CMake, Maven, Scons, and QMake. Many studies have investigated bugs and fixes in several systems, but to our best knowledge, none focused on bugs in build systems. One significant feature of software build systems is that they should work on various platforms, i.e., various operating systems (e.g., Windows, Linux), various development environments (e.g., Eclipse, Visual Studio), and various programming languages (e.g., C, C++, Java, C#), so the study of software build systems deserves special consideration. In this paper, we perform an empirical study on bugs in software build systems. We analyze four software build systems, Ant, Maven, CMake and QMake, which are four typical and widely-used software build systems, and can be used to build Java, C, C++ systems. We investigate their bug database and code repositories, randomly sample a set of bug reports and their fixes (800 bugs reports totally, and 199, 250, 200, and 151 bug reports for Ant, Maven, CMake and QMake, respectively), and manually assign them into various categories. We find that 21.35% of the bugs belong to the external interface category, 18.23% of the bugs belong to the logic category, and 12.86% of the bugs belong to the configuration category. We also investigate the relationship between bug categories and bug severities, bug fixing time, and number of bug comments.

key words: *software build system, bug category, empirical study*

1. Introduction

The common goal of build systems is to convert source code, libraries and other data into executable programs by orchestrating the execution of compilers and other tools. In addition, build systems also support the packaging of web-based applications, the generation of software product documentation, the automatic static analysis of source code, and other related activities [1].

Many research studies have investigated different aspects of build systems. Tamrawi et al. and Adams et al. analyze the Makefile by performing symbolic execution [2] and by constructing a build graph [3]. Neitsch et al. analyze issues in build systems for multiple programming languages [4]. Suvorov et al. investigate the migration of build systems in practice; it analyzes Linux Kernel and KDE as two case studies [5].

The whole build process use various systems and tools: *version-control tools*, which store the source code and ensure concurrent development for developers; *compilation tools*, which convert input source code into object code or executable programs; *software build systems*, which collect sufficient information about the relationship between source files and object files, and use necessary compilation tools to produce the final build output (e.g., executable programs, software package, documentation, static analysis results). *Software build systems* play the most important roles in building systems, since they orchestrate the entire build process, and control the final build output. There are various software build systems, such as Make, Ant, CMake, Maven, Scons, and QMake.

Understanding software build systems could provide a better guide for constructing a robust and fault-tolerant build systems. One necessary step to understand build system behavior is to understand the features and characteristics of bugs in their software build systems. To better understand the nature of bugs in software build systems and to potentially help to prevent, resolve, mitigate, or manage such bugs, in this paper, we perform an exploratory study of real bugs found in these tools.

A number of studies have investigated bugs and their fixes in various systems [6]–[13], these studies provide guide for triaging bug reports, detecting duplicated bug reports, designing bug location tools, reduce testing and maintenance costs, and helping to improve development efficiency. However, to our best knowledge, none of these studies focus on bugs in software build systems. One significant feature of software build systems is that they should work on various platforms, i.e., various operating systems (e.g., Windows, Linux), various development environments (e.g., Eclipse, Visual Studio), and various programming languages (e.g., C, C++, Java, C#). Thus, analyzing bugs and their fixes in software build systems deserves a special consideration.

In this study, we analyze four software build systems:

1. Apache Ant*, one of the most popular build systems for Java-based projects.
2. Apache Maven**, a software build automation and comprehension tool used primarily for Java projects.
3. CMake***, a software build automation which trans-

Manuscript received November 15, 2013.
Manuscript revised February 15, 2014.

[†]The authors are with the College of Computer Science and Technology, Zhejiang University, China.
[††]The author is with the School of Information Systems, Singapore Management University, Singapore.
[†††]The author is with the School of Computer and Information Engineering, Zhejiang Gongshang University, China.
a) E-mail: yewang@mail.zjgsu.edu.cn

DOI: 10.1587/transinf.E97.D.1769

*http://ant.apache.org/
**http://maven.apache.org/
***http://www.cmake.org/

Copyright © 2014 The Institute of Electronics, Information and Communication Engineers

lates a high-level build description into a lower-level description which can be used by other build system, such as GNU Make.
4. QMake[†], a part of the QT development environment, which is similar to CMake.

To investigate bugs that appear in the four systems, we analyze their bugs and code repositories. We collect bug reports with status "closed", and manually check their fixes. Apache Ant uses Bugzilla to track all the bugs; we manually check the commit logs in CVS to retrieve the fixes related to a bug. Apache Maven and QMake use JIRA to track bugs, and JIRA contains links from bug reports to a list of changes in the source control repositories that fix those bugs. CMake uses MantisBT to track bugs, and uses Github[††] to manage source code, and we notice developers post links of source code changes related to the corresponding bugs in Github.

The goal of our study is to characterize bugs that appear in build systems (their frequency, their categories, their severities, and the difficulties to fix these bugs (measured by fix time and number of comments)). To achieve these goals, we would answer the following questions: How often a bug appears in software build systems? How much bugs per kLOC? What categories of bugs appear in software build systems? What is the severity distribution for each category of bugs? How long does it take to fix various categories of bugs? How many comments of a bug received for each category of bugs? To answer the above questions, we perform both manual and automated analysis on a randomly sampled set of "closed" bug reports and their corresponding bug fixing commits. We find that 21.35% of the bugs belong to the external interface category, 18.23% of the bugs belong to the logic category, and 12.86% of the bugs belong to the configuration category.

This paper extends our preliminary study published as a short paper in a conference [14]. It extends the preliminary study in various ways: two additional research questions are investigated, descriptions of datasets and our empirical study methodology are extended, related work section is expanded, and a number of examples are added.

The main contributions of this paper are as follows:

1. To our best knowledge, this is the first time that a large scale, semi-automated empirical study of bugs in software build systems has been performed. Software build systems should adapt to different platforms, development environments, and programming languages, which is different from other systems.
2. We collect and manually label 800 bug reports from four software build systems into various categories by extending the bug categorization in [15].
3. We investigate the relationships between bug categories and bug severities, bug fixing time, and number of bug comments. We believe our analysis of the bugs would provide guidance for preventing, detecting, mitigating, resolving, and managing bugs in software build systems, and also help construct reliable and fault-tolerant build systems.

The remainder of the paper is organized as follows. In Sect. 2, we describe the detail information of our collected datasets and our empirical study methodology. In Sect. 3, we present our empirical study which addresses and answers a set of research questions. In Sect. 4 we briefly highlight related studies. In Sect. 5, we conclude and present future work.

2. Dataset & Methodology

In this section, we first describe the four software build systems that analyzed in the study in Sect. 2.1. Then, we present our methodology in Sect. 2.2.

2.1 Dataset Collection

2.1.1 Apache Ant

Apache Ant, maintained by Apache Software Foundation, is one of the most popular software build systems for Java applications. It is a Java library and command-line tool which uses XML files to describe the build process and its dependencies. A number of built-in features allowing it to compile, assemble, test and run Java applications are supported. It was officially released as a stand-alone product on July 19, 2000 with version 1.1. On May 23, 2012, its latest version (version 1.8.4) was released. Ant consists of 254,431 lines of source code, and 1,167 Java files (on March 31, 2013). To analyze bug reports of Ant, we analyze its Bugzilla bug tracking system located at:

http://ant.apache.org/bugs.html

For our manual analysis, we randomly select 199 bug reports out of 2,567 bug reports with status of "closed" and "fixed", contained in Bugzilla tracking system. The reason that we pick bug reports with "fixed" status is that there are only 177 bug reports with status of "closed", which is quite a small proportion of the whole Ant bug repository.

2.1.2 Apache Maven

Apache Maven, also maintained by Apache Software Foundation, serves a similar purpose as Apache Ant, but is based on different concepts and works in a different manner. Maven works on the concept of a project object model (POM), and uses a plugin-based framework that allows it to make use of any application controllable through standard input. Maven was released with version 1.0 in July 2004. And on February 23, 2013, its latest version (version 3.0.5) was released. Maven consists of 51,651 lines of source code, and 346 Java files (on March 31, 2013). To analyze bug reports of Maven, we analyze its JIRA bug tracking system located at:

[†]http://qt-project.org/doc/qt-4.8/qmake-manual.html
[††]https://github.com/

http://jira.codehaus.org/browse/MNG

For our manual analysis, we randomly select 250 bug reports out of 2,945 reports tagged as bugs[†], with status "closed", contained in the JIRA tracking system.

2.1.3 CMake

CMake is a cross-platform, open-source build system, which is different from Make, Ant, and Maven, since it does not actually execute the build process. The build process with CMake takes place in two stages. First, CMake translates a high-level build description into a native build system's own language; Then, the platform's native build system is used for the actual building. CMake can generate Makefiles for various platforms and IDEs, such as Unix, Windows, Mac OS X, etc. It was officially released as a stand-alone product on January 01, 2003 with version 1.0. On November 7, 2012, its latest version (version 2.8.10) was released. CMake consists of 406,715 lines of source code, and 1,104 C/C++ files (on March 31, 2013). To analyze bug reports of CMake, we analyze its MantisBT bug tracking system located at:

http://public.kitware.com/Bug/my_view_page.php

For our manual analysis, we randomly select 200 bug reports out of 5,192 bug reports with status of "closed", contained in MantisBT tracking system.

2.1.4 QMake

QMake is a part of the QT development environment, which is similar to CMake, and can be used to generate either a Makefile or a Visual Studio project. QMake integrates with the QT framework, and automatically create moc (meta object compiler) and rcc (resource compiler) sources. QMake can generate Makefiles for various platforms, such as Unix, Windows, Mac OS X, etc. QMake is distributed with QT toolkit 3.X around 2003. On January 31, 2013, its latest version (Qt toolkit 5.0.1) was released. QMake consists of 33,583 lines of source code, and 53 C++ files (on March 31, 2013). To analyze bug reports of QMake, we analyze its JIRA bug tracking system located at:

https://bugreports.qt-project.org/browse/QTBUG

For our manual analysis, we randomly select 151 bug reports out of 882 reports tagged as bugs, with status "closed", contained in the JIRA tracking system. We notice that only 2 closed bugs of QMake appear from 2003 to 2005. Thus, in this study, we choose to analyze QMake bug reports reported after 2005.

2.2 Methodology

We focus on only closed bugs from the bug repositories, as bug reports that are not closed may not be bugs or have no fixes or enough information for our analysis yet. Table 1 shows version, lines of source code, number of source files, and the numbers of closed bugs for Ant, Maven, CMake, and QMake, respectively. We also show the durations (in years) between the first and last bugs we collected in column Duration.

[†]JIRA provides various issue types: bugs, improvements, new features, tasks, tests, wishes, brainstorming, etc

Table 1 Statistics information of closed bugs in software build systems.

Projects	Version	Lines of Code	No. Files	Bug Count	Duration
Ant	1.8.4	254,431	1,167	2,567	5.18 years
Maven	3.0.5	51,651	346	2,945	10.66 years
CMake	2.8.10	406,715	1,104	5,192	9.72 years
QMake	5.0.1	33,583	53	844	6.96 years

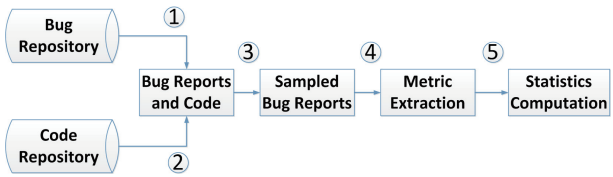

Fig. 1 The whole process of the empirical study.

Figure 1 presents the whole process of our empirical study. We first download the bug reports from bug repositories and the source code from code repositories of the four software build systems (Steps 1 and 2). Next, we randomly select a set of bug reports from the whole bug report collection for each system (Step 3). Then, we extract some information, identity the fixes for each bug report in the four systems, and assign bug reports into different categories (Step 4). Finally, we compute some statistics based on the collected information extracted in Step 4 (Step 5). We explain more on the information extraction and statistics computation (i.e., Steps 4 and 5, respectively) in the following paragraphs.

2.2.1 Information Extraction

In our empirical study, we extract the following pieces of information:

1. **Bug Severity**: This metric can be fetched directly from bug reports.
2. **Bug Fix Times**: This metric can be computed by measuring the difference of bug creation time and bug closed time.
3. **Number of Bug Comments**: This metric is computed by counting the number of times a bug has been discussed.

To get bug fixes we check both bug and code repositories. For Apache Ant, since it doesn't contain source code change information in bug reports, we check its commit log in CVS to identity the fixes (bugs whose fixes can't be identified from the commit logs are discarded). For Apache Maven, CMake, and QMake, we find the list of source code changes for each closed bug report either in the comment

Table 2 Bug categories for software build systems.

Category	Definition
algorithm/method	The implementation of an algorithm/method works in an unexperted way.
assignment/ initialization	A variable or data item is assigned a wrong value or not properly used.
checking	Missing necessary checks for potential error conditions, or an error response specified for error conditions.
data	Wrong usage of data structure, point, and type conversions.
logic	Incorrect logical expression in condition statements (e.g., if, case, and loop blocks).
non-functional defects	Failure to meet non-functional requirements, such as defines improper variable or method, implement non-compliance method with standard documentation.
timing/ optimization	Error related to times, concurrency or performance issues, e.g., slow complication, memory leak, etc.
internal interface	Errors in interfaces between different component in the same system, such as incorrect operation of files or database, and errors of inheritance. For software build systems, it also refers to errors of build system dependency graph, such as generating wrong dependency, losing dependency, etc.
external interface	Errors of user interface (including usability issues). For software build systems, it also refers to errors of the usage of build system tools in different platforms, such as various operation systems (e.g., Windows, Linux), various development environments (e.g., Eclipse, Visual Studio), various program languages (e.g., C, C++, Java, C#).
configuration	Error in non-code files (e.g., configuration files) that cause error in functionality.
others	Other bugs not fall into one the above categories.

text (CMake) or in the source section (Maven and QMake).

Due to the large number of bug reports in these 4 software build system, we randomly select a subset of bug report to study the categories of bugs in software build systems followed by previous studies [9], [11], [16], [17]. We assign bug reports to several categories manually. We make use of bug description, bug comment textual information, and also bug fixes to infer the bug category. We use the set of categories proposed by Seaman et al. in [15], and we extend it by adding 1 category (i.e., configuration). Table 2 presents the bug categories and their description.

2.2.2 Statistics Computation

We compute various statistics for each bug category and for all the bug reports we investigate in the four software build systems. These statistics are then used to answer various research questions in Sect. 3.1. We investigate the relationships between bug categories and bug severities, bug fixing time, and number of bug comments by using these statistics.

3. Empirical Study

In this section, we present the research questions and their answers of our empirical study, and threats to validity.

3.1 Research Questions

We are interested in the following research questions:

RQ1 *How often bugs appear in software build systems?*

Software build systems are popular and complex, almost every non-trivial software system would use software build systems. In this research question, we would like investigate the bug densities of Ant, Maven, CMake, and QMake. To answer this research question, we compute the number of bugs per kLOC, per source code files, and per year.

RQ2 *What are the categories of bugs appearing in software build systems?*

To better understand bugs, Seaman et al. have proposed a categorization of bugs [15]. Thung et al. have manually assigned a number of machine learning bugs to Seaman et al.'s categories to understand the nature of these bugs [18]. Improved understanding of bugs can help developers to improve the quality of a software system. Developers can spend more testing and code review effort on the categories of bugs that are particularly problematic (e.g., most of the reported bugs fall under this category). In this research question, we investigate the categories of bugs that appear in popular build systems and analyze whether bugs of some categories appear more often than bugs from other categories. To answer this research question, we first manually categorize the bugs into different categories presented in Table 2, and we analyze the number of bugs in each category.

RQ3 *What are the severity distributions of the various categories of bugs?*

The severity level of a bug report indicates the seriousness of the bug [19]. In this research question, we investigate severity of bugs under each category. The answer to this research question could help developers better understand bugs under each category and take appropriate action. For example, if most of the bugs in a particular category are given high severity levels (e.g., blocker, and critical), then developers should make more effort to reduce bugs in this category. To answer this research question, we compute the distribution of bugs of various severity levels under each category.

RQ4 *How long does it take to fix various categories of bugs?*

In this research question, we investigate the fixing time of bugs under each category. The answer to this research question could help developers or project managers estimate bug fixing time once they categorize a bug, which can help to better organize a bug fixing plan and in allocating resources to fix various bugs. To answer this research question, we first collect the creation date and closed date (last update date) of each closed bug report, and we measure bug

Table 3 Bug densities in four software build systems.

Projects	Bug Number Per kLOC	Bug Number Per File	Bug Number Per Year
Ant	10.09 bugs/kLOC	2.20 bugs/file	495.56 bugs/year
Maven	57.02 bugs/kLOC	8.51 bugs/file	276.27 bugs/year
CMake	12.77 bugs/kLOC	4.70 bugs/file	534.16 bugs/year
QMake	25.13 bugs/kLOC	15.92 bugs/file	121.26 bugs/year

fixing time as the difference of these two dates. We record the minimum, maximum, mean, and median number of days for each category.

RQ5 *How many comments of a bug received for each category of bugs?*

The number of bug comments represents the degree of developer activity in fixing a bug. For a bug, if there are more people joining the discussion and posting their comments, it means that either the bug is hard to fix or developers are more interested in the bug. In this research question, we investigate the number of comments that various bugs received under each category. To answer this research question, we record the minimum, maximum, mean, and median number of comments for each category.

3.2 RQ1: Bug Densities

We present the bug densities of Ant, Maven, CMake, and QMake in Table 3. We found that Maven has the highest average number of bugs per kLOC (57.02 bugs/kLOC), followed by QMake (25.13 bugs/kLOC), CMake (12.77 bugs/kLOC), and Ant (10.09 bugs/kLOC). These numbers indicate that for every line of code, developers of Maven need to fix more bugs than the others. We also notice that bug counts per kLOC of software build systems are much higher than those reported for algorithm-intensive machine learning systems [9] and operating systems [12], [20]. For example, one machine learning system Lucene only has 2.77 bugs per kLOC, but Maven has 57.02 bugs per kLOC, which is more than 20.58 times than that of Lucene.

We also report the average number of bugs per source file, and year in the last two columns of Table 3. QMake is a component of QT toolkit, which only has 53 C/C++ source files and 844 bugs reported in QT bug tracking system, but it contains the highest number of bugs per source file (15.92 bugs/file), followed by Maven (8.51 bugs/file), CMake (4.70 bugs/file), and Ant (2.20 bugs/file). Moreover, Maven and CMake have received bug reports for a long period of time — 10.66 and 9.72 years respectively. CMake has the highest average number of bugs per year (534.16 bugs/year), followed by Ant (495.56 bugs/year), Maven (276.27 bugs/year), and QMake (121.26 bugs/year). We also notice the average number of bug reported per year is much higher than those reported for machine learning systems [9]. For example, Lucene has 144.76 bugs/year, and CMake has around 3.69 times more bugs than Lucene annually.

Note that a high number of bugs or a high bug density does not necessarily mean that a project has low software quality [21]. There are various other factors which affect the number of reported bugs; For example, the popularity of a project (e.g., many developers contribute to the project, many users use this project and report the bugs), and the size and complexity of the project, affect the number of reported bugs and bug densities. As we noticed, almost every system of substantial size and complexity needs to use software build systems, which makes the build systems become popular to users. For this reason, although we find that the bug densities in software build systems are much more than those of other systems, the quality of these tools is not necessarily poor, due to the popularity of build tools and the fact that they are widely deployed. Still, the fact that there are many bugs affecting build systems, highlight the importance for the development of automated techniques that can help developers improve the quality of build systems.

Maven has a much higher number of bugs per kLOC (57.02 bugs/kLOC) than QMake (25.13 bugs/kLOC), CMake (12.77 bugs/kLOC), and Ant (10.09 bugs/kLOC), while CMake has a much higher number of bugs per year (534.16 bugs/year) than Ant (495.56 bugs/year), Maven (276.27 bugs/year), and QMake (121.26 bugs/year).

3.3 RQ2: Bug Categories

We randomly sample 800 bug reports from the four build tool systems. Since there more than 10,000 bug reports in the 4 build systems, similar to prior empirical studies on bug reports [9], [11], [16], [17], we only analyze a subset of these reports. We manually analyze a similar number of bug reports as those analyzed in prior studies.

There bugs are then manually assigned into different categories. The distribution of bugs based on the 11 categories is presented in Table 4. We notice that most bugs are categorized as external interface (21.35%), followed by logic (18.23%), and then followed by configuration (12.86%). There are only 2.12% of bugs that fall into the category others. The small proportion of bugs in the category others indicates that the remaining 10 categories are sufficient to cover most of bugs for software build systems.

External interface category corresponds to bugs of user interface and those related to usability issues. Software build systems need to work for various systems, and on various platforms, i.e., various operating systems (e.g., Windows, Linux), various development environments (e.g., Eclipse, Visual Studio), and various programing languages (e.g., C, C++, Java, C#). Also as users need to use these systems often, they would pay attention to build system usability and user interface. These explain why bugs in external

Table 4 Bug categories in four software build systems.

Category	Number	Percentage
algorithm/method	36	4.49%
assignment/initialization	67	8.36%
checking	61	7.62%
data	100	12.48%
logic	146	18.23%
non-functional defects	12	1.50%
timing/optimization	13	1.62%
internal interface	74	9.24%
external interface	171	21.35%
configuration	103	12.86%
others	17	2.12%

Fig. 2 An example of external interface bug.

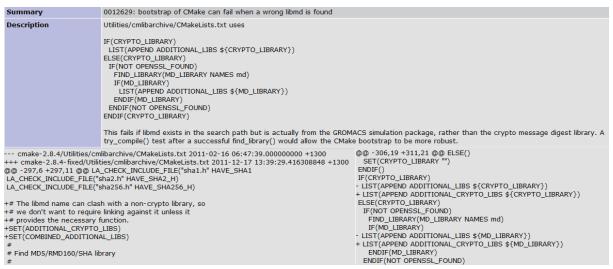

Fig. 3 An example of logic bug.

interface category are the most. Figure 2 shows an example of an external interface bug. It describes a bug about integrating Maven with glassfish plugin.

Logic category corresponds to bugs of incorrect expression appearing in conditional statements. For software build systems, since we need to make them platform-independent, we have to consider different conditions, such as the variable assignment in different platform conditions, which make the logic bugs appear more than many other bug categories. To identify logic bugs, we need to check the source code modification logs. Figure 3 shows an example of a logic bug and one patch file. It describes the failure to bootstrap affecting CMake. Only after we read its patch file, we can identify that it is a logic bug.

Configuration category corresponds to bugs in non-code files (e.g., configuration files). The whole build process can be simply viewed as reading from a configuration file (e.g., Makefile, build.xml), and building the system according to the command in the configuration file. For some software build systems, such as CMake and QMake, they are based on low-level build systems such as make, thus they

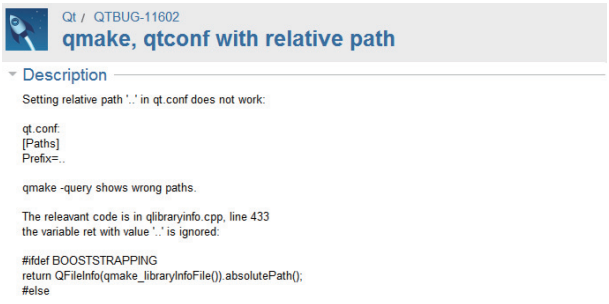

Fig. 4 An example of configuration bug.

need to parse their configuration files and generate low-level configuration files. This makes configuration bugs appear more than many other bug categories. Figure 4 shows an example of a configuration bug. It describes a problem about qtconf file.

The most common categories of bugs in Ant, Maven, CMake and QMake are: external interface (21.35%), logic (18.23%), and configuration (12.86%), while the least common category of bugs aside from others is non-functional.

3.4 RQ3: Bug Severity

Next, we investigate the relationship between bug category and bug severity. We study the same five severity levels[†] as [9], i.e., Block, Critical, Major, Minor, and Trivial. Block is the most severe category while Trivial is the least severe category. Table 5 presents the relationship between bug category and bug severity in Ant, Maven, CMake and QMake.

We notice that major and minor severities dominate all the bug categories. For example, in internal interface, checking, logic, and non-functional categories, major severity dominates, while in algorithm/method, timing/optimization, assignment/initialization, external interface, data, and configuration categories, minor severity dominates. It is notable that in JIRA (Maven, QMake), the default severity level when a user creates a new bug report is major, while in Bugzilla (Ant), the default severity level is normal, and in MantisBT (CMake), the default severity level is minor. Past studies have shown that the number of bug reports with default severity level is the largest [19], [22]–[24]. Herraiz et al. state that the many severity levels can confuse bug reporters which are often unable to distinguish the meaning of the different severity levels [25]. Lamkanfi

[†]We notice in JIRA (Maven and QMake), the word priority is used instead of severity. In Bugzilla (Ant), there are 7 severity levels (blocker, critical, major, normal, minor, trivial, and enhancement). And in MantisBT (CMake), there are 8 severity levels (blocker, crash, major, minor, tweak, text, trivial, and feature). To make the severity level consistent with a previous study [9], we assign them into 5 severity levels, i.e., we assign normal to minor, enhancement to trivial for Ant, and we assign crash to critical, and tweak, text, feature to trivial for CMake.

Table 5 Relationship between bug category and bug severity in four software build systems.

Category	Severity	Number	Proportion	Category	Severity	Number	Proportion
algorithm/method	Block	0	0.00%	timing/optimization	Block	0	0.00%
	Critical	0	0.00%		Critical	0	0.00%
	Major	8	22.22%		Major	2	15.38%
	Minor	28	77.78%		Minor	10	76.92%
	Trivial	0	0.00%		Trivial	1	7.69%
assignment/initialization	Block	0	0.00%	internal interface	Block	9	12.16%
	Critical	14	20.90%		Critical	10	13.51%
	Major	16	23.88%		Major	36	48.65%
	Minor	36	53.73%		Minor	14	18.92%
	Trivial	1	1.49%		Trivial	5	6.76%
checking	Block	1	1.64%	external interface	Block	8	4.68%
	Critical	8	13.11%		Critical	42	24.56%
	Major	36	59.02%		Major	53	30.99%
	Minor	15	24.59%		Minor	57	33.33%
	Trivial	1	1.64%		Trivial	11	6.43%
data	Block	0	0.00%	configuration	Block	3	2.91%
	Critical	13	13.00%		Critical	17	16.50%
	Major	28	28.00%		Major	25	24.27%
	Minor	47	47.00%		Minor	48	46.60%
	Trivial	12	12.00%		Trivial	10	9.71%
logic	Block	3	2.05%	others	Block	1	5.88%
	Critical	56	38.36%		Critical	4	23.53%
	Major	68	46.58%		Major	7	41.18%
	Minor	14	9.59%		Minor	5	29.41%
	Trivial	5	3.42%		Trivial	0	0.00%
non-functional defects	Block	4	33.33%				
	Critical	2	16.67%				
	Major	4	33.33%				
	Minor	2	16.67%				
	Trivial	0	0.00%				

et al. argue that bug reporters who do not assess bug severity level well often assign these bugs to the default severity level [22]. As a future work, we could explore automated methods (e.g., [19], [22]–[24]) to recommend more accurate severity levels to bugs before performing a more detailed analysis.

Following the definition of the various severity levels, block bug refers to a bug that causes system crash, data corruption, irreparable harm, etc., and critical bug refers to a bug that affects an important function and it has no reasonable workaround. Analyzing block and critical bugs can provide insight towards developing a more robust application. From Table 5, we notice all the bug categories except algorithm/method and timing/optimization contain bugs of block or critical severity levels. For external interface and logic categories, they contain the most of block and critical bugs, i.e., with 50 (29.24%) and 59 (40.41%) bugs, respectively. In Sect. 3.3, we report that most of the bugs are categorized as external interface and logic. This results show that external interface and logic categories not only take the majority of bugs, but also contain the most of block and critical bugs. To develop a new software build system, we should pay special attention to there two categories of bugs. Moreover, block bugs are much less than critical bugs, for example, in the external interface category, there are 8 block bugs compared to 42 critical bugs, and in the logic category, there are 3 block bugs compared to 56 critical bugs.

Considering the trivial severity level, the proportion of such bugs is small. Category data contains the highest proportion of trivial bugs (12.00%), followed by configuration (9.71%).

External interface and logic categories not only takes the majority of bugs (21.35% and 18.23%), but also contain the most of block and critical bugs (29.24% and 40.41%). We should pay special attention to these two categories of bugs when developing a new software build system. Trivial bugs takes the smallest proportion.

3.5 RQ4: Bug Fixing Time

Next, we investigate the relationship between bug categories and bug fixing times. For each closed bug report, we collect its creation date and closed date (last update date), and we measure bug fixing time as the difference of these two date. Table 6 presents the bug fixing times in term of days for various bug categories. We record the minimum, maximum, mean, and median number of days for each category.

We notice that the minimum period a bug is fixed is just a few seconds for all the categories. We check these bug reports manually, and find that most of them are reported and fixed by the same developers. This phenomenon follows the observation by Lamkanfi and Demeyer [26]. The maximum fixing time can take a few years. Three categories with the highest maximum bug fixing time are data (4,040.0257 days), checking (3,473.0556 days) and configuration (2,617.8535 days). Categories non-functional and

Table 7 Relationship between bug category and bug fixing time in four software build systems.

Category	Fixing Time	Number	Proportion	Category	Fixing Time	Number	Proportion
algorithm/method	Within A month	11	30.56%	timing/optimization	Within A month	5	38.46%
	Within A Year	7	19.44%		Within A Year	3	23.08%
	More than A Year	18	50.00%		More than A Year	5	38.46%
assignment/initialization	Within A month	21	31.34%	internal interface	Within A month	54	72.97%
	Within A Year	32	47.76%		Within A Year	13	17.57%
	More than A Year	14	20.90%		More than A Year	7	9.46%
checking	Within A month	43	70.49%	external interface	Within A month	49	28.65%
	Within A Year	12	19.67%		Within A Year	60	35.09%
	More than A Year	6	9.84%		More than A Year	62	36.26%
data	Within A month	27	27.00%	configuration	Within A month	27	26.21%
	Within A Year	24	24.00%		Within A Year	27	26.21%
	More than A Year	49	49.00%		More than A Year	49	47.57%
logic	Within A month	83	56.85%	others	Within A month	9	52.94%
	Within A Year	33	22.60%		Within A Year	3	17.65%
	More than A Year	30	20.55%		More than A Year	5	29.41%
non-functional defects	Within A month	7	58.33%				
	Within A Year	2	16.67%				
	More than A Year	3	25.00%				

Table 6 Bug fixing times in terms of days in four software build systems.

Category	Min	Max	Mean	Median
algorithm/method	0.0014	2544.2750	861.8832	328.9979
assignment/ initialization	0.0118	2542.1111	767.0298	319.8646
checking	0.0007	3473.0556	315.5635	2.8306
data	0.0014	4040.0257	783.1057	339.0691
logic	0.0014	2562.8535	210.3875	11.7160
non-functional defects	0.0472	748.1382	121.9941	5.1135
timing/ optimization	0.6688	859.3326	233.5380	117.3604
internal interface	0.0021	2405.3354	127.1160	5.0056
external interface	0.0042	2600.9215	458.1231	173.8674
configuration	0.0062	2617.8535	739.9159	318.7201
others	0.0062	2545.1285	708.0220	28.2229

timing/optimization have the smallest maximum bug fixing time.

We further investigate the mean and median bug fixing time. We notice the mean bug fixing times for all the categories are quite high compared to the fixing times reported in machine learning systems [9]. For categories algorithm/method, assignment/initialization, data, configuration, and others, the mean fixing times are around 2 years. For categories non-functional and internal interface, the mean fixing times are around 4 months. For the remaining categories, the mean fixing times are around 1 year. Although the mean fixing time is high, the median fixing time is not always high. For category checking, its mean fixing time is 315.5635 days, but its median fixing time is only 2.8306 days. For checking bugs, some bugs needs much longer time to fix, while most of them just need a short time to fix. For categories non-functional and internal interface, their mean fixing time and median fixing times are lower than those of the other categories.

Table 7 investigates the relationship between bug categories and bug fixing times bucketized into: less than a month, less than a year, and more than a year. For categories internal interface, checking, non-functional, logic, and others, most the bugs are fixed in less than a month, i.e., 72.97%, 70.49%, 58.33%, 56.85%, 52.94%, respectively. For category assignment/initialization, most of the bugs are fixed in less than a year. For categories algorithm/method, data, and configuration, most of the bugs are fixed in more than a year, i.e., 50.00%, 49/00%, and 47.57%, respectively. For category external interface, we notice that the number of bugs fixed in less than a month, less than a year, and more than a year are almost the same, i.e., 28.65%, 35.09%, and 36.26%, respectively.

In terms of median bug fixing time, bugs in checking (2.8306 days), internal interface (5.0056 days), and non-functional (5.1135 days) categories take the shortest time to fix, while bugs in data (339.0691 days), algorithm/method (328.9979 days), and assignment/initialization (319.8646 days) take the longest time to fix.

3.6 RQ5: Number of Comments

Finally, we investigate the relationship between bug categories and number of bug comments. Table 8 presents the number of bug comments for various bug categories. We record the minimum, maximum, mean, and median number

Table 8 Number of comments in four software build systems.

Category	Min	Max	Mean	Median
algorithm/method	1	13	3.39	3.00
assignment/initialization	0	27	4.16	2.00
checking	0	13	2.87	2.00
data	0	22	2.59	2.00
logic	0	25	4.31	3.00
non-functional defects	1	6	2.08	2.00
timing/optimization	1	8	3.77	4.00
internal interface	0	12	3.12	2.00
external interface	0	38	5.42	4.00
configuration	0	25	4.31	3.00
others	1	6	1.82	1.00

Table 9 Relationship between bug category and number of bug comments in four software build systems.

Category	Fixing Time	Number	Proportion	Category	Fixing Time	Number	Proportion
algorithm/method	0 or 1	9	25.00%	timing/optimization	0 or 1	1	7.69%
	Less than 5	23	79.62%		Less than 5	10	23.08%
	More than 5	4	15.38%		More than 5	2	38.46%
assignment/initialization	0 or 1	25	37.31%	internal interface	0 or 1	32	43.24%
	Less than 5	28	41.79%		Less than 5	27	36.49%
	More than 5	14	20.90%		More than 5	15	20.27%
checking	0 or 1	26	42.62%	external interface	0 or 1	28	16.37%
	Less than 5	28	45.90%		Less than 5	98	57.31%
	More than 5	7	11.48%		More than 5	45	26.32%
data	0 or 1	33	33.00%	configuration	0 or 1	31	30.10%
	Less than 5	40	40.00%		Less than 5	42	40.78%
	More than 5	27	27.00%		More than 5	30	29.13%
logic	0 or 1	70	47.59%	others	0 or 1	11	64.71%
	Less than 5	59	40.41%		Less than 5	5	29.41%
	More than 5	17	11.64%		More than 5	1	5.88%
non-functional defects	0 or 1	6	50.00%				
	Less than 5	5	41.67%				
	More than 5	1	8.33%				

of comments for each category.

We notice bugs in others and non-functional categories have the smallest mean number of comments, and as shown in Table 4, these two categories also have the smallest proportion of bugs. For bugs in external interface category, the mean number of comments is 5.42, which is the highest number of comments, followed by logic (4.31) and configuration (4.31). Table 9 further investigates the relationship between bug categories and number of bug comments bucketized into: 0 or 1 comment, less than 5 comments, and more than 5 comments. We notice that the proportion of bugs which have more than 5 comments is low, while most of the bugs have less than 5 comments.

In terms of mean number of bug comments, bugs in external interface (5.34 comments), logic (4.31 comments), and configuration (4.31 comments) categories have the most number of comments, which correspond to the same 3 categories which have the most number of bugs (c.f., Table 4). Bugs in others (1.82 comments) and non-functional (2.08 comments) categories have the least number of comments, which corresponding to the same 2 categories which have the least number of bugs.

3.7 Discussion

Comparison to Findings in Prior Studies: In the above 5 research questions, we first investigate the overall bug distribution across the 4 software build systems. Next, we manually analyze each bug and assign them to different categories proposed by Seaman et al. [15]. For each bug report category, we investigate its severity, bug fix time, and number of comments distribution.

Some of the research questions asked by ours are also investigated in other empirical studies for other types of software systems (e.g., machine learning systems [9], mobile platforms [27], and Google chrome project [28]). Various bug categorization schemes have been investigated, e.g., performance vs. non-performance bugs [11], [17], security vs. non-security bugs [11], and configuration vs. non-configuration bugs [16].

The closest to our work, is the study by Thung et al. which also categorize bugs into the bug categorization scheme proposed by Seaman et al. [15]. They analyze machine learning bugs while we analyze build system bugs. Our findings are different from Thung et al. in the following respects:

1. The densities of bugs in software build systems are much high than these of machine learning systems [9]. For example, the average number of bugs per kLOC for Maven (a build system) is 57.02 bugs/kLOC, while the average number of bugs per kLOC for Lucene (a machine learning system) is 2.77 [9]. This difference is attributed mainly to the popularity of build system – almost every system of substantial size and complexity needs to use software build systems.
2. For bugs in software build systems, external interface, logic, and configuration bugs appear more often than the others, while in machine learning systems [9], algorithm/method, non-functional, and assignment/initialization bugs are the 3 main categories of bugs. Understanding the categories of frequently occurring bugs could help developers plan quality improvement efforts. Different from machine learning systems, for build systems, developers need to pay more attentions to external interface, logic, and configuration bugs.

Implications: Our findings (RQ1) show that compared with other systems (e.g., Lucene which is a popular information retrieval system), a lot more bugs are reported for build systems. This highlights the need for build system developers to employ more advanced techniques to better manage bug reports. Techniques like duplicate bug report de-

tection [29]–[31], bug severity/priority prediction [22], [23], and automated bug categorization [18] can be used to help developers manage the mass of bug reports. Furthermore, our findings (RQ2, RQ3, and RQ5) suggest that build system developers need to give more attention to the external interface of build systems as most of build system bugs appear in this category, many of these bugs are critical and blocker bugs, and these bugs are among the hardest ones to fix (in terms of the number of comments needed to resolve the bug). Furthermore, our results (in RQ4) can be used by developers and project managers to estimate the amount of time needed to fix different kinds of build system bugs.

3.8 Threats to Validity

There are several threats that may potentially affect the validity of our empirical study. Threats to internal validity relates to experimenter bias and errors. Our study involves manual inspection of bugs. This process is potentially error-prone. To reduce this threat, one of the authors collects the bug reports, and each bug report is labeled by one author and is checked by at least another. Finally, we discuss to decide the final bug category.

Threats to external validity relates to the generalizability of our study. We have analyzed four software build systems: Ant, Maven, CMake, and QMake. To improve the generalizability of our findings we intentionally pick four software build systems; Ant and Maven are used mainly for Java application, CMake works as a substitute of make, and QMake is a component published with the QT toolkit. In these systems, bugs are managed using different bug tracking systems: JIRA, Bugzilla, and MantisBT. However, there are still many other software build systems we have not analyzed. Also, we only randomly pick 800 bug reports for the four software build systems. Although this is not a very large number, we believe it is a good sample size as past studies, such as [9], [32]–[35], investigate similar numbers of manually labeled data. We plan to reduce this threat to external validity in the future by analyzing more software build systems and bug reports.

4. Related Work

In this section, we survey the related work in the field of software build system maintenance in the software engineering literature. We first briefly introduce some empirical studies on software build system maintenance. Next, we briefly introduce some empirical studies on bugs and fixes. Finally, we review the tools for build script analysis.

4.1 Empirical Study on Build System Maintenance

McIntosh et al. investigate version histories of one proprietary and nine open source projects [36]. They conclude that build maintenance incurs up to a 27% overhead on source code development and a 44% overhead on test development. Adams et al. [37] analyze the changes to the Linux kernel build system from its inception up to version 2.6 using MAKAO [3]. They conclude that a good balance between obtaining a fast, correct build system and migrating in a step by step way is the general approach followed by developers maintaining the Linux build system. A similar conclusion is also observed for ant-based build systems [38].

Suvorov et al. provide an empirical study for build system migration; they analyze two cases: KDE and Linux kernel [5]. They describe 4 types of major challenges for build system migration, i.e., requirements gathering, communication issues, balance between performance and the complexity of build code, and effective evaluation of build system.

Neitsch et al. perform an empirical study of the build systems for programs that are developed in multiple programming languages [4]. They identify the major issues for building multi-language software, and explore the reasons why these issues occur. Tu and Godfrey perform case studies on build-time software architecture, and introduce the "code robot" architectural style [39].

4.2 Empirical Study on Bugs and Fixes

There are various empirical studies on bugs and fixes. Seaman et al. make use of NASA historical data by creating model to guide future software development, and propose a set of bug categories [15]. Our study extend the set of bug categories, and label build system bugs into the bug categories. Thung et al. perform an empirical study of bugs in machine learning systems [9]. They investigate three open source machine learning system: Apache Mahout, Lucene, and OpenNLP. Our study is similar to their study, but we focus on another type of systems: software build systems. We believe our result can help developers understand bugs in software build systems better.

Chou et al. investigate bugs in operating systems in the Linux and OpenBSD kernels [12]. They analyze the root cause of bugs, bug distribution, bug life cycle, bug clusters, and the difference between operating system bugs and other bugs. Pan el al. investigate bug fix patterns in a number of systems, and categorize the bug fix types based on the syntax of code changes [40]. Zaman et al. perform a study on security and performance bugs in Firefox [11]. Lu et al. investigate concurrency bugs in MySQL, Apache Web Server, Mozilla, and OpenOffice [7]. Maji et al. study bugs in Android and Symbian [10]. Bhattacharya et al. perform an empirical study on bug reports and bug fixing in open source Android applications [27].

4.3 Tools for Build Script Analysis

To our best knowledge, only two tools, i.e., MAKAO [3] and SYMAKE [2], have been developed for build script analysis. MAKAO is a visualization and smell detection tool for make-based build scripts. MAKAO generates a build graph from a Makefile, and based on this, it support various functionalities such as querying build-related data, and viewing the build architecture in different perspectives. MAKAO

works on concrete build graphs [41].

Tamrawi et al. [2], [42] propose SYMAKE to analyze the dynamic Makefile. It proposes a symbolic execution algorithm to process Makefiles and produces a symbolic build graph (SDG). SYMAKE has been used in several applications, such as the detection of several types of code smell and errors in build system. It could also be used to support build code refactoring.

5. Conclusions and Future Work

To better understand software build systems, we perform an empirical study on bugs in four such systems. We analyze four software build systems: Apache Ant, Apache Maven, CMake, and QMake. We first download their bug repositories and code repositories. Next, we randomly pick 800 bugs (199, 250, 200, and 151 bug reports for Ant, Maven, CMake and QMake, respectively), and manually assign them into different categories. We further investigate the relationship between bug categories and bug severities, bug fixing time, and number of bug comments. We find that among the 800 bug reports, 21.35% of the bugs belong to the external interface category, 18.23% of the bugs belong to logic category, and 12.86% of the bugs belong to configuration category.

In the future, we plan to investigate more software build systems, and analyze more bug reports. We also plan to design an automatic bug categorization tool for build systems.

Acknowledgments

This research is sponsored in part by NSFC Program (No.61103032) and National Key Technology R&D Program of the Ministry of Science and Technology of China (No2013BAH01B01).

References

[1] P. Smith, Software Build Systems: Principles and Experience, Addison-Wesley Professional, 2011.
[2] A. Tamrawi, H. Nguyen, H. Nguyen, and T. Nguyen, "Build code analysis with symbolic evaluation," 2012 34th International Conference on Software Engineering (ICSE), pp.650–660, 2012.
[3] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter, "Design recovery and maintenance of build systems," IEEE International Conference on Software Maintenance, ICSM 2007, pp.114–123, 2007.
[4] A. Neitsch, K. Wong, and M. Godfrey, "Build system issues in multilanguage software," IEEE International Conference on Software Maintenance, ICSM 2012, 2012.
[5] R. Suvorov, M. Nagappan, A. Hassan, Y. Zou, and B. Adams, "An empirical study of build system migrations in practice: Case studies on KDE and the linux kernel," IEEE International Conference on Software Maintenance, ICSM 2012, 2012.
[6] R. Chillarege, I.S. Bhandari, J.K. Chaar, M.J. Halliday, D.S. Moebus, B.K. Ray, and M.-Y. Wong, "Orthogonal defect classification — A concept for in-process measurements," IEEE Trans. Softw. Eng., vol.18, no.11, pp.943–956, 1992.
[7] S. Lu, S. Park, E. Seo, and Y. Zhou, "Learning from mistakes: A comprehensive study on real world concurrency bug characteristics," ACM Sigplan Notices, vol.43, no.3, pp.329–339, 2008,
[8] L. Ma and J. Tian, "Web error classification and analysis for reliability improvement," J. Systems and Software, vol.80, no.6, pp.795–804, 2007.
[9] F. Thung, S. Wang, D. Lo, and L. Jiang, "An empirical study of bugs in machine learning systems," 23rd IEEE International Symposium on Software Reliability Engineering, 2012.
[10] A.K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile OSes: A case study with android and symbian," 2010 IEEE 21st International Symposium on Software Reliability Engineering (ISSRE), pp.249–258, 2010.
[11] S. Zaman, B. Adams, and A.E. Hassan, "Security versus performance bugs: A case study on firefox," Proc. 8th Working Conf. on Mining Soft. Rep., pp.93–102, 2011.
[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," Operating Systems Review, vol.35, no.5, pp.73–88, 2001.
[13] Z. Li, L. Tan, Y. Zhou, and C. Zhai, "Have things changed now?," An empirical study of bug characteristics in modern open source software, In ICSE: Proc. 29th International Conference on Software Engineering, 2007.
[14] X. Xia, X. Zhou, D. Lo, and X. Zhao, "An empirical study of bugs in software build systems," 2013 13th International Conference on Quality Software (QSIC), pp.200–203, 2013.
[15] C.B. Seaman, F. Shull, M. Regardie, D. Elbert, R.L. Feldmann, Y. Guo, and S. Godfrey, "Defect categorization: Making use of a decade of widely varying historical data," Proc. Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, pp.149–157, 2008.
[16] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L.N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," Proc. Twenty-Third ACM Symposium on Operating Systems Principles, pp.159–172, 2011.
[17] S. Zaman, B. Adams, and A.E. Hassan, "A qualitative study on performance bugs," 2012 9th IEEE Working Conference on Mining Software Repositories (MSR), pp.199–208, 2012.
[18] F. Thung, D. Lo, and L. Jiang, "Automatic defect categorization," 2012 19th Working Conference on Reverse Engineering (WCRE), pp.205–214, 2012.
[19] T. Menzies and A. Marcus, "Automated severity assessment of software defect reports," IEEE International Conference on Software Maintenance, ICSM 2008, pp.346–355, 2008.
[20] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," ACM SIGPLAN Notices, vol.46, no.3, pp.305–318, 2011.
[21] I. Herraiz, E. Shihab, T.H. Nguyen, and A.E. Hassan, "Impact of installation counts on perceived quality: A case study on debian," 2011 18th Working Conference on Reverse Engineering (WCRE), pp.219–228, 2011.
[22] A. Lamkanfi, S. Demeyer, E. Giger, and B. Goethals, "Predicting the severity of a reported bug," 2010 7th IEEE Working Conference on Mining Software Repositories (MSR), pp.1–10, 2010.
[23] A. Lamkanfi, S. Demeyer, Q.D. Soetens, and T. Verdonck, "Comparing mining algorithms for predicting the severity of a reported bug," 2011 15th European Conference on Software Maintenance and Reengineering (CSMR), pp.249–258, 2011.
[24] Y. Tian, D. Lo, and C. Sun, "Information retrieval based nearest neighbor classification for fine-grained bug severity prediction," 2012 19th Working Conference on Reverse Engineering (WCRE), pp.215–224, 2012.
[25] I. Herraiz, D.M. German, J.M. Gonzalez-Barahona, and G. Robles, "Towards a simplification of the bug report form in eclipse," Proc. 2008 International Working Conference on Mining Software Repositories, pp.145–148, 2008.
[26] A. Lamkanfi and S. Demeyer, "Filtering bug reports for fix-time analysis," 2012 16th European Conference on Software Maintenance and Reengineering (CSMR), pp.379–384, 2012.
[27] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S.C. Koduru, "An em-

pirical analysis of bug reports and bug fixing in open source android apps," 2013 17th European Conference on Software Maintenance and Reengineering (CSMR), pp.133–143, 2013.
[28] S. Lal and A. Sureka, "Comparison of seven bug report types: A case-study of google chrome browser project," Proc. 2012 19th Asia-Pacific Software Engineering Conference, vol.01, pp.517–526, 2012.
[29] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," Proc. 32nd ACM/IEEE International Conference on Software Engineering, vol.1, pp.45–54, 2010.
[30] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," Proc. 30th International Conference on Software Engineering, pp.461–470, 2008.
[31] A.T. Nguyen, T.T. Nguyen, T.N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," 2012 Proc. 27th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp.70–79, 2012.
[32] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc, "Is it a bug or an enhancement?: A text-based approach to classify change requests," Proc. 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds, p.23, 2008.
[33] C. Bird, A. Bachmann, F. Rahman, and A. Bernstein, "Linkster: Enabling efficient manual inspection and annotation of mined data," Proc. Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.369–370, 2010.
[34] Y. Tian, J. Lawall, and D. Lo, "Identifying linux bug fixing patches," 2012 34th International Conference on Software Engineering (ICSE), pp.386–396, 2012.
[35] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: Recovering links between bugs and changes," SIGSOFT FSE, pp.15–25, 2011.
[36] S. McIntosh, B. Adams, T. Nguyen, Y. Kamei, and A. Hassan, "An empirical study of build maintenance effort," 2011 33rd International Conference on Software Engineering (ICSE), pp.141–150, 2011.
[37] B. Adams, K. De Schutter, H. Tromp, and W. De Meuter, "The evolution of the linux build system," Electronic Communications of the EASST, vol.8, pp.1–16, 2008.
[38] S. McIntosh, B. Adams, and A. Hassan, "The evolution of ant build systems," 2010 7th IEEE Working Conference on Mining Software Repositories (MSR), pp.42–51, 2010.
[39] Q. Tu and M. Godfrey, "The build-time software architecture view," Proc. IEEE International Conference on Software Maintenance, pp.398–407, 2001.
[40] K. Pan, S. Kim, and E.J. Whitehead, Jr., "Toward an understanding of bug fix patterns," Empirical Software Engineering, vol.14, no.3, pp.286–315, 2009.
[41] C. Gunter, "Abstracting dependencies between software configuration items," ACM Trans. Software Engineering and Methodology (TOSEM), vol.9, no.1, pp.94–131, 2000.
[42] A. Tamrawi, H. Nguyen, H. Nguyen, and T. Nguyen, "Symake: A build code analysis and refactoring tool for makefiles," Proc. 27th IEEE/ACM International Conference on Automated Software Engineering, pp.366–369, 2012.

Xin Xia is a Ph.D. candidate in College of Computer Science and Technology, Zhejiang University, China. His research interests include software mining, empirical study. He is a student member of Institute of Electrical and Electronics Engineers.

Xiaozhen Zhou is a Ph.D. candidate in College of Computer Science and Technology, Zhejiang University, China. His research interests include software engineering and system monitor.

David Lo received his Ph.D. degree from the School of Computing, National University of Singapore in 2008. He is currently an assistant professor in the School of Information Systems, Singapore Management University. He works in the areas of software engineering and data mining. He is particularly interested in specification mining, debugging, software text analytics, frequent pattern mining, and social network mining. He is a member of the Institute of Electrical and Electronics Engineers and Association for Computing Machinery.

Xiaoqiong Zhao is a Ph.D. candidate in College of Computer Science and Technology, Zhejiang University, China. His research interests include empirical software engineering.

Ye Wang received the Ph.D. degree in Computer Science from Zhejiang University in 2013. She is a lecturer in School of Computer and Information Engineering, Zhejiang Gongshang University. Her research interests include software engineering and service computing.