An Empirical Study on Real Bug Fixes

Hao Zhong*† Zhendong Su†
*Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
†University of California, Davis, USA
zhonghao@sjtu.edu.cn, su@ucdavis.edu

***Abstract*—Software bugs can cause significant financial loss and even the loss of human lives. To reduce such loss, developers devote substantial efforts to fixing bugs, which generally requires much expertise and experience. Various approaches have been proposed to aid debugging. An interesting recent research direction is *automatic program repair*, which achieves promising results, and attracts much academic and industrial attention. However, people also cast doubt on the effectiveness and promise of this direction. A key criticism is to what extent such approaches can fix real bugs. As only research prototypes for these approaches are available, it is infeasible to address the criticism by evaluating them directly on real bugs. Instead, in this paper, we design and develop BUGSTAT, a tool that extracts and analyzes bug fixes. With BUGSTAT's support, we conduct an empirical study on more than 9,000 real-world bug fixes from six popular Java projects. Comparing the nature of manual fixes with automatic program repair, we distill 15 findings, which are further summarized into four insights on the two key ingredients of automatic program repair: fault localization and faulty code fix. In addition, we provide indirect evidence on the size of the search space to fix real bugs and find that bugs may also reside in non-source files. Our results provide useful guidance and insights for improving the state-of-the-art of automatic program repair.**

I. INTRODUCTION

Over the past decades, software has permeated into almost every economic activity, and is boosting economic growth from many perspectives. At the same time, like any other man-made artifacts, software suffers from various bugs which lead to incorrect results, deadlocks, or even crashes of the entire system. When this happens in a critical application, it can cause great loss of money or even human lives. For example, Zhivich and Cunningham [40] report that the software of a hospital miscalculated the proper dosage of radiation for patients. In this accident, at least eight patients died. To improve the quality of software, it is desirable to fix as many bugs as possible. The long battle with software bugs began ever since software existed. It requires much effort to fix bugs, *e.g.* Kim and Whitehead [14] report that the median time for fixing a single bug is about 200 days.

It has been actively studied to reduce the effort of fixing bugs. A recent research direction is to investigate automatic approaches to fixing bugs (see Section V for a detailed discussion). A typical approach in this direction first locates faults of a program, and then mutates the located faulty code with predefined operators until the program passes all the test cases. In this paper, we refer to this line of research as *automatic program repair*, which complements traditional approaches (*e.g.* [18]), since it has the potential to deal with a variety of different bugs. Research in this direction has already produced promising results. For example, Le Goues *et al.* [17] reported that their approach was able to automatically fix 55 out of 105 bugs. However, many people question the positive results. Existing approaches (*e.g.* [17], [12]) seem to be able to fix only simple bugs, due to various limitations. For example, Table 2 of Kim *et al.* [12] lists their ten templates to fix bugs, which are all quite simple. As existing empirical studies (*e.g.* [14]) show that it requires much expertise and time to fix bugs, many researchers doubt the reported positive results. For example, at ICSE 2014, Monperrus [22] criticized Kim *et al.*'s recent work [12] and discussed a number of issues concerning this research direction.

The preceding criticism shows that the research community has limited knowledge on the nature of bug fixes. Although empirical studies exist to understand bug fixes (see Section V for details), only one [21] analyzed the links between the nature of bug fixes and automatic program repair. Furthermore, the empirical study focuses on only one aspect of automatic program repair, namely the search space of fixing bugs; most questions raised by Monperrus [22] are still open. For example, how many bugs could be fixed by automatic program repair? Which parts should be focused upon to further improve the state-of-the-art? It is important to carefully design an empirical study to answer these questions, and its results will benefit future research in this direction:

Benefit 1. The results will provide insights on the potential of automatic program repair. For example, the results will reveal how many bugs can be fixed by simple changes. As pointed out by Monperrus [22], existing approaches are effective in fixing this type of bugs. As another example, the results will reveal the essential operators to fix bugs. If we compare these operators with an existing approach, we may estimate the potential of the approach for fixing bugs.

Benefit 2. The results will provide insights on how to improve existing approaches. For example, the results will reveal the distribution of bug fixes, and the required knowledge to fix each type of bugs. Future research may leverage such knowledge to fix more bugs. As another example, the results will reveal the nature of bug fixes, and future work may be able to tune existing approaches to achieve their best performance.

Benefit 3. The results will provide insights on new research directions of fixing bugs. For example, the results will reveal how many bug fixes require modifications on only non-source files. Follow-up work may explore how to locate bugs in non-source files and how to fix them with advanced techniques.

Despite the preceding benefits, it is difficult to conduct such an empirical study, due to the following challenges:

Challenge 1. It is difficult to collect the bug fixes for the empirical study. First, it needs a large number of bug fixes to ensure the representativeness of the results, but many projects do not provide adequate data for analysis. For example, many projects on SourceForge[1] do not have a long active period, so they provide limited bug fixes for analysis. Second, Kim *et al.* [15] point out that it needs high quality bug fixes to reduce superficial conclusions, but many bug fixes are polluted. For example, although Linux has a long active period, its bug fixes are intertwined with other types of commits (*e.g.* new features). Although Tian *et al.* [31] propose an approach to identifying bug fixes for Linux, its precision is relatively low, due to the difficulty in accurately identifying bug fixes.

Challenge 2. It is challenging to implement the support tool. It is infeasible to analyze a large amount of bug fixes manually, so it is desirable to implement a support tool for analysis. To save space, a bug fix typically consists of only buggy files and modified files, instead of the whole project. As a result, the tool should be able to compare and analyze partial code.

To address the two preceding challenges, we collect high-quality bug fixes, and implement a tool, called BUGSTAT, that automatically extracts, compares and classifies bug fixes. With the support of the tool, we conduct the first empirical study to investigate the aforementioned research questions. This paper identifies the following key insights:

- **Fault localization.** Our results show that it is reasonable to assume that it needs to fix only several files to fix a bug (Findings 2 and 14). Current fault localization approaches can deal with 30% of source files at the most (Finding 3). To deal with more source files, researchers should consider multiple faulty lines (Findings 4 and 5) and the data dependence among faulty lines (Finding 6).
- **Faulty code fix.** Our results show that it is reasonable for automatic program repair to focus on modified source files (Findings 12 and 15). The existing approaches can potentially fix 30% of source files (Finding 3), but their effectiveness in practice may be further reduced since it is difficult to decide the faulty lines of a bug, and the interference among multiple bugs is serious (Finding 4). To fix more bugs, researchers can focus on mutation operators of several most common modified code elements (Finding 8), API knowledge (Finding 11), the frequency of repair actions (Finding 9), multiple faulty lines (Finding 5), their data dependence (Finding 6), and multi-language programming (Finding 14).
- **The search space.** Combining the results of Martinez and Monperrus [21] with our results in Figure 2, we find that automatic program repair can potentially fix only half of the buggy files, due to the huge space of searching correct repair shapes (Finding 4). The potential is further reduced, since even after such a shape is found, the effort to find its concrete edits is nontrivial (Finding 11).
- **Non-source bugs.** Our results show that about 10% of bugs reside in non-source files (Finding 1), and the bug prediction models should consider non-source files (Findings 1 and 2). Most of these files are configuration files and natural language documents (Finding 13). Even in source files, there are many bug fixes on non-code elements such as comments (Finding 7).

The rest of the paper is structured as follows. Section II presents our research methodology, and Section III presents our empirical results. We discuss possible extensions to our study in Section IV. Section V surveys related work, and Section VI concludes.

II. METHODOLOGY

This section describes the dataset used in our empirical study and our research questions. To answer these research questions, we should analyze thousands of bug fixes. To reduce the effort of manual inspection, we have developed the BUGSTAT tool to identify and classify bug fixes.

A. Dataset

Table I lists the subject projects used in our study. Aries[2] is a set of Java components that enable an enterprise OSGi application programming model. Cassandra[3] is a distributed database management system that handles large amounts of data across commodity servers. Derby[4] is a relational database. Lucene[5] is an information retrieval library, and Solr[6] is an enterprise search platform that is built on Lucene. As Lucene and Solr share the same source code repository, it is difficult to determine whether a commit belongs to Lucene or Solr. We put the results of the two projects into a single row. Mahout[7] is a machine learning library. All the projects are from the Apache software foundation[8], and most Apache projects carefully maintain the links between bug reports and bug fixes. Wu *et al.* [35] reported that even simple heuristics achieved almost the same precision and recall with their proposed sophisticated technique, when they identified bug fixes for the Apache projects. We select the Apache projects, so that we can focus on the study, rather than the techniques to identify bug fixes. Column "LOC" lists lines of code. To ensure the reliability of our results, we selected both median and large projects. The total lines of code add up to more than one million. We collected the dataset in February 2014. Cassandra changed its source code repository from SVN[9] to Git[10] in December 2011. As BUGSTAT retrieves commits from only SVN repositories, for Cassandra it retrieved the data before the repository was changed.

[1] http://sourceforge.net/

[2] https://aries.apache.org
[3] http://cassandra.apache.org
[4] http://db.apache.org/derby
[5] https://lucene.apache.org
[6] https://lucene.apache.org/solr
[7] https://mahout.apache.org
[8] http://www.apache.org
[9] https://svn.apache.org/repos/asf/cassandra/
[10] https://git-wip-us.apache.org/repos/asf?p=cassandra.git

TABLE I
DATASET

Name	LOC	Bug			Commit			
		F	I	%	T	N	K	%
Aries	142,110	497	490	98.6%	5,318	839	226	20.0%
Cassandra	121,170	1,374	1,236	90.0%	6,825	1,708	281	29.1%
Derby	659,426	2,433	2,022	83.1%	10,285	4,624	346	48.3%
Lucene/Solr	677,873	3,145	2,226	70.8%	26,890	4,234	2,019	23.3%
Mahout	121,084	457	407	89.1%	3,632	553	289	23.2%
Total	1,721,663	7,906	6,381	80.7%	52,950	11,958	3,161	28.6%

F: fixed bugs in bug reports. I: bugs whose fixes are identified by issue number; T: total commits; N: bug-fix commits that are identified by issue number; K: bug-fix commits that are identified by keywords.

All the projects in Table I have source code repositories. For each project in Table I, BUGSTAT retrieves all the commits from its source code repository. Each commit has a message. We inspected the messages, and found two types of bugs: (1) bugs reported through issue trackers, which we call *reported bugs*, and (2) those not reported to issue trackers, which we call *on-demand* bugs. To identify fixes of the two types, we define the following two criteria:

1. Issue number. All the projects in Table I have their issue trackers to track various issues (*e.g.* bugs, improvements, new features, tasks, and sub-tasks). Each reported issue has an associated issue number. If a change of an issue is committed, programmers often write its issue number to the message of the commit. For example, in Cassandra, a commit's message says "implement multiple index expressions. patch by jbellis; reviewed by Nate McCall for CASSANDRA-1157". The benefit of the practice is that it is easy to track the issue. In the issue tracker, the page of the issue[11] lists useful information (*e.g.*, its description, the discussions among programmers, and the relations to other issues). In the Apache projects, when writing issue number to messages, programmers typically use the "name-number" pattern, where *name* denotes the name of a project, and *number* denotes the issue number. BUGSTAT uses the pattern to extract issue number, and checks the issue tracker to determine whether a commit is a bug fix. In the above example, as CASSANDRA-1157 is a sub-task, BUGSTAT determines that the commit is not a bug fix.

In Table I, column "Bug" lists the results with the issue number criterion. Initially, the resolution of a reported issue is *unresolved*, and is changed to *fixed* after programmers fix the issue. Programmers may not fix some issues, since they resolve these reports as *invalid* and *duplicate*. Programmers may have different opinions on some reported issues, and change them to other categories. For example, a reported bug can be changed as a new feature request. For the "Bug" column, subcolumn "F" lists the number of bug reports that are resolved as *fixed*; subcolumn "I" lists the number of bug reports whose bug fixes are identified by issue number; and subcolumn "%" is calculated as $\frac{D}{F}$. The result shows that the issue number criterion identifies fixes for 80% of the reported bugs, so our study reflects the nature of the majority. BUGSTAT is similar to existing approaches (*e.g.* [35]). The difference is that BUGSTAT checks the issue tracker to determine whether an issue number indicates a bug, while other approaches assume that every issue number indicates a bug.

2. Keyword. Issue trackers do not store all the bug fixes. In some cases, programmers may bypass issue trackers, especially when they believe that a change is trivial. When they commit a change, programmers may write a message to describe the fix. For example, in Aries, the message of a commit says "Fix broken service registration listener". BUGSTAT determines a commit as a bug fix, if its message contains words such as "bug" or "fix". The preceding commit was identified as a bug fix, since its message contains the keyword "fix". The heuristic is simple, and a number of previous studies (*e.g.* [16]) used the same technique to extract bug fixes.

In Table I, column "Commit" lists the results using the two criteria. For this column, subcolumn "T" lists the number of retrieved commits; subcolumn "N" lists the number of bug fixes that are identified by issue number; subcolumn "K" lists the number of bug fixes that are identified by keywords; and subcolumn "%" is calculated as $\frac{N+K}{T}$. Our result shows that about 30% of commits are bug fixes. In addition, as shown in subcolumns "I" and "N", a *reported bug* has two commits on average, and an *on-demand bug* has one commit by definition. If it is necessary, we treat the two types of bugs differently, when we investigate our research questions.

B. Research Questions

We consider the following research questions:

RQ1. To what extent are bugs localized (Section III-A)?
As the first step, automatic program repair leverages fault localization techniques to locate the faulty line. Typically, a fault localization approach compares passing and failing traces. The suspiciousness of each line is calculated by different formulae, according to how many times the line appears in failing and passing traces. Wong and Debroy [34] show that most fault localization approaches assume that each buggy source file has exactly one line of faulty code. However, if a bug has multiple lines of faulty code, the impacts of these lines may depend on each other. In this study, we analyze the fault distribution of real bugs. The results provide insights on locating faults.

RQ2. How complicated is it to fix bugs (Section III-B)?
After a faulty line is located, automatic program repair mutates the faulty line to generate candidates, and uses genetic algorithm (*e.g.* [17]) or random search (*e.g.* [26]) to select candidates, until a candidate passes all the test cases. These approaches are effective in fixing a faulty line, but may be

[11] https://issues.apache.org/jira/browse/CASSANDRA-1157

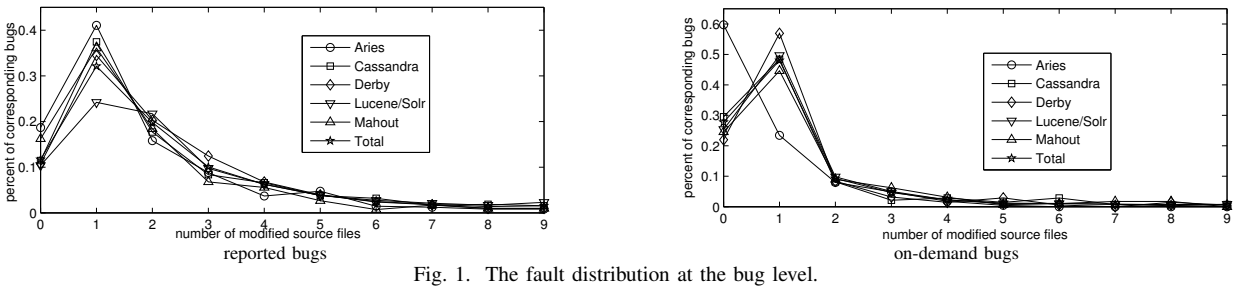

Fig. 1. The fault distribution at the bug level.

ineffective in fixing multiple faulty lines, especially when such lines are relevant. For example, for a specific bug, two lines of code have data dependence, and it is important to maintain the dependence during the bug fix process. If the two lines of code are mutated independently, their data dependence can be easily broken. In this study, we analyze data dependence among faulty lines to investigate the complexity of fixing bugs; the results provide insights on fixing bugs.

RQ3. What operators are essential for fixing bugs (Section III-C)?

When fixing bugs, automatic program repair uses predefined operators to mutate faulty code. The operators are quite important, since they decide how many and which bugs can be fixed. Current automatic program repair uses quite limited operators. For example, although Kim *et al.* [12] achieved better results than previous approaches, their approach relies only ten templates to mutate code. Although it is widely known that existing approaches use incomplete operators, it is challenging to make improvements. In this study, we analyze the operations to fix real bugs. The results provide insights on designing more comprehensive operators.

RQ4. What is the importance of API knowledge to fix bugs (Section III-D)?

API knowledge is useful in various programming tasks (*e.g.*, coding [43] and migrating code [42]). Our previous work [44] shows that careless programmers may introduce API-related bugs. However, current automatic program repair does not leverage API knowledge, and thus is ineffective in fixing API-related bugs. In this study, we analyze how many bug fixes are related to APIs. The results provide insights on the importance of API knowledge for fixing bugs.

RQ5. What kinds of files are necessary to be modified to fix bugs (Section III-E)?

We notice that many bugs are not related to source files, and such bugs could not be fixed by current automatic program repair techniques. It is desirable to understand where such bugs reside, so we could investigate their nature and explore corresponding repair approaches. In this study, we analyze the types and the distribution of modified files when programmers fix real bugs. The results provide insights on fixing bugs that do not reside in source files.

RQ6. How many files are necessary to be added or deleted to fix bugs (Section III-F)?

Automatic program repair only modifies source files, but programmers may also add or delete files to fix real bugs. Thus existing approach may be insufficient in fixing certain bugs. In this study, we analyze the distribution of added and deleted files when programmers fix real bugs. The results provide insights on fixing the corresponding bugs.

We implement BUGSTAT to reduce the manual effort, and it uses ChangeDistiller [7] to compare source files and PPA [4] to parse partial code.

III. EMPIRICAL RESULTS

A. RQ1: Fault Distribution

We calculate the number of modified files for each bug fix, and Figure 1 shows the distribution. Its horizontal axes show the number of modified source file, and its vertical axes show the percentage of the corresponding bugs. The results lead to the following findings:

Finding 1. In total, programmers did not modify any source files to fix about 10% of reported bugs and about 20% of on-demand bugs. Yin *et al.* [38] show that both open-source and commercial projects contain many errors in configuration files, and Xiong *et al.* [36] propose an approach to fix such errors. In this study, we find bug fixes in configuration files. For example, a bug report of Solr[12] says that the released code did not compile, since the compiler's configuration file did not set the paths correctly. The modified lines are as follows:

```
+       <tarfileset dir="../lucene"
+         prefix="lucene"
+         excludes="**/build/" /> ...
```

Our previous work [41] detects errors in documents. In this study, we find bug fixes in documents. For example, a bug report of Lucene[13] says that the online user guide did not display correctly, and the modified line is as follows:

```
-<!DOCTYPE..."..../xhtml1/DTD/xhtml1-transitional.dtd">
+<!DOCTYPE..."..../html4/loose.dtd">
```

Researchers [45], [13], [37] have proposed models to predict buggy source files for a bug report. As source files are quite different from non-source files, it needs nontrivial extension to predict buggy non-source files. In addition, as non-source files are typically not executable, it needs nontrivial extension for fault localization to locate faulty lines for non-source files. Section III-E shows that some modified files are in programming languages other than Java. As BUGSTAT parses Java code, it considers only Java code as source files.

Finding 2. In total, programmers modified one or more source files to fix about 90% of reported bugs and more than 70%

[12]https://issues.apache.org/jira/browse/SOLR-1989

[13]https://issues.apache.org/jira/browse/LUCENE-4302

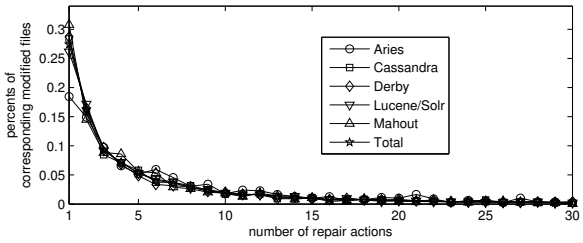

Fig. 2. The fault distribution at the file level.

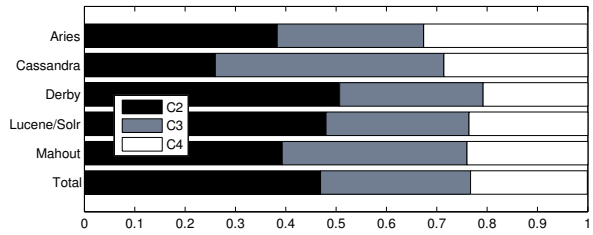

Fig. 3. The category for fault complexity.

of on-demand bugs. The number of buggy files for a bug is an important parameter for fault prediction models (*e.g.* [45], [13]). Our results show that the percentage of bugs decreases rapidly with the increasing number of modified source files, and the percentage of on-demand bugs decreases even faster. It is reasonable to set the value of the parameter to be four or less, since more than 80% of bugs have fewer than four modified source files.

We further analyzed the modified locations of each modified source file. The underlying tool, ChangeDistiller [7], produces a set of repair actions from two compared source files. A repair action is a pair $\langle a, e \rangle$, where a is an action such as *add*, *delete*, *update*, and *move*, and e is a code element. For actions, BUGSTAT considers *add* as additions, *delete* as deletions, and *update* and *move* as modifications. To understand bug fixes, we often need statement-level changes. For example, it requires different knowledge to fix `if` statements and `return` statements, although modified internal code elements are the same (*e.g.* variables). For each repair action inside a statement, BUGSTAT replaces its code element with the statement, and it ignores repair actions on the same statement. A code element may not have a parent statement (*e.g.*, `Modifier` and `Javadoc`). BUGSTAT does not change repairs on these code elements. In addition, when programmers fix a bug, they often modify the test code to reproduce the bug. As repair actions on test code are more like implementing new features, BUGSTAT ignores these repair actions. For each of the remaining modified files, we calculate its number of repair actions, and Figure 2 shows the results. Its horizontal axis shows the number of repair actions, and its vertical axis shows the percentage of the corresponding modified files. We have the following finding:

Finding 3. In total, programmers made a repair action to fix less than 30% of source files. Wong and Debroy [34] claim that most fault localization approaches assume that each buggy file has exactly a faulty line. As these source files fit the assumption, existing approaches have the potential to locate their faulty lines. However, their effectiveness in practice may be limited for two reasons. First, a faulty line may not appear in traces, so fault localization approaches cannot locate it. For example, code comments are not executable and will not appear in traces, although they may contain errors. As another example, fixing bugs sometimes requires changing modifiers. In particular, a bug report of Solr[14] says that the visibility of a method should be changed. The original code is as follows:

```
46: public abstract Query parse()...
```

The fixed code is as follows:

```
46: protected abstract Query parse()...
```

As many tools instrument only method bodies, Line 46 will not appear in their traces. Second, an added or deleted code element may have multiple lines (see Section III-C for details). For example, adding a method is considered a single repair action. Current automatic program repair cannot add a method effectively, since a method may contain many lines.

Finding 4. In total, programmers made at least two repair actions to fix more than 70% of source files.

DiGiuseppe and Jones [5] conducted an empirical study to explore the influence of multiple faults on fault localization approaches. Their results show that fault localization approaches are effective in locating at least one faulty line, but the ranks of the remaining faulty lines are low. They suggest that programmers may use fault localization to locate and fix faulty lines one bye one. The suggestion works for humans, but may not apply to automatic program repair for two reasons. First, automatic program repair needs to rerun test cases to guide the bug fix process. DiGiuseppe and Jones [5] show that the interference among multiple faults is serious, and such interference may negatively impact the search algorithms. Second, it may be insufficient to mutate a single line to fix bugs. For example, a bug report of Solr[15] says that an exception was thrown due to the misuse of Java reflection. In particular, three methods use Java reflection to call the methods of `clob`, and one method has the faulty lines as the following:

```
110: Method m = clob.getClass().
          getDeclaredMethod("getCharacterStream");
111: if (Modifier.isPublic(m.getModifiers())) {
       ...
116:    return (Reader) m.invoke(clob);
117: }
```

If the `getCharacterStream` method is defined in the superclass of `clob`, Line 110 will throw an exception, since `clob` does not define the method. The fixed code is as follows:

```
110:  return clob.getCharacterStream();
```

It is reasonable for programmers to fix the bug from a faulty line, but it can be difficult for automatic program repair to fix the bug, since fault localization approaches typically do not report the line range of a bug. Qi *et al.* [27] use different fault localization approaches to locate the faulty lines for automatic program repair. Their results are consistent with our results, since they find that the best technique for humans is not the best technique for automatic program repair. Still, more research is needed to fully understand such influence, since their study [27]

[14]https://issues.apache.org/jira/browse/SOLR-601

[15]https://issues.apache.org/jira/browse/SOLR-1794

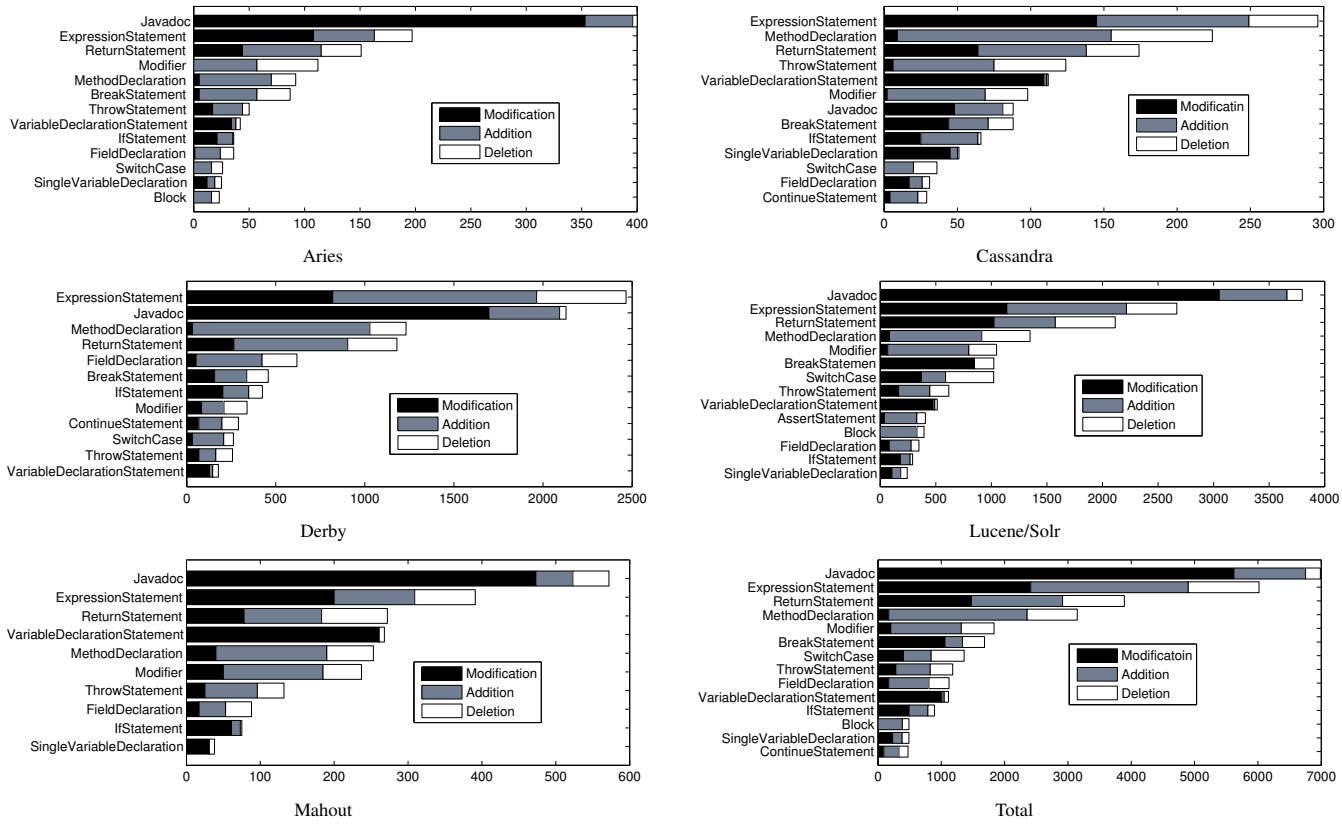

Fig. 4. The distribution of repair actions.

does not analyze low-level details (*e.g.* the inference of multiple faults). Research exists (*e.g.* [19]) to recover the links between test cases and multiple bugs, but additional research is needed to locate multiple faults and fault ranges, especially for automatic program repair.

Automatic program repair uses machine learning algorithms such as genetic algorithm [17] and random search [26] to guide the bug fix process. Martinez and Monperrus [21] mine two repair models (CT and CTET) from bug fixes. The two mined repair models are the probability distributions of repair actions (*e.g.*, inserting a statement) at different granularity. Their results show that the space of searching repair shapes explodes when the length of repair actions is more than four or eight for the two models, respectively. Here, a repair shape is a set of repair actions. The two repair models are built on ChangeDistiller, and the granularity of ChangeDistiller is between CT and CTET. If we adjust the repair models to the granularity of ChangeDistiller, the maximum number may be between four and eight. Figure 2 shows that about half of the buggy files have more repair actions. In addition, even after a correct repair shape is found, it is still nontrivial to synthesize the concrete fix. As a result, we estimate that current automatic program repair cannot fix more than half of the buggy files due the large search space to synthesize fixes.

B. RQ2: Fault Complexity

Based on the fault complexity, we classify modified source files into four categories: a single repair action (*C1*), non-data dependent repair actions (*C2*), data dependent repair actions (*C3*), and mixture repair actions (*C4*). BUGSTAT puts a repair action into a set of data dependent repair actions, if its code element contains a variable, and the variable has data dependence on variables in that set.

As shown in Figure 2, programmers made a single repair action to fix about 30% of source files. These source files fall into the *C1* category. Current automatic program repair is effective in fixing bugs in this category. We calculated the number for the other three categories, and Figure 3 shows the results. Its vertical axis shows the subject projects, and its horizontal axis shows the percentages of the three categories. The results lead to the following findings:

Finding 5. As shown in Figures 2 and 3, programmers made multiple non-data dependent repair actions to fix about 40% of source files (the *C2* category). As these repair actions are not data dependent, it may be feasible to apply the repair actions one by one when fixing bugs. For example, a bug report of Lucene[16] says that an exception was thrown if a resource was closed, and a submitted patch is as follows:

```
@@ -152,7 +152,15 @@
-          swapSearcher(newSearcher);
+          boolean success = false;
+          try {
+            swapSearcher(newSearcher);
+            success = true;
+          } finally {
               ...
+          }
```

[16]https://issues.apache.org/jira/browse/LUCENE-3476

```
@@ -204,7 +212,12 @@
-  public void close() throws IOException {
-    swapSearcher(null);
+  public synchronized void close() throws IOException {
+    if (currentSearcher != null) {
       ...
+      swapSearcher(null);
+    }
```

In the first diff hunk of the patch, programmers placed a statement inside a `try` statement. In the second diff hunk of the patch, programmers put a statement into an `if` statement. As the two faulty lines have no data dependency, it may be feasible for existing approaches to fix them one by one.

Finding 6. In total, as shown in Figures 2 and 3, programmers made data dependent repair actions to fix more than 40% of source files (the *C3* and *C4* categories). For example, a bug report of Mahout[17] says that a function got broken over time, and the faulty lines were as follows:

```
130: if (r == 0) {
131:   if (index < o.index) {
132:     return -1;
133:   } else if (index > o.index) {
134:     return 1;
135:   }
136:   return 0;
137: } else {
138:   return r;
139: }
```

When `r` is zero, the preceding code returns a wrong value. To calculate the correct value, programmers should understand the relation among `r`, `o.index`, and `index`. The modified lines are as follows:

```
130: if (r != 0) {
131:   return r;
132: } else {
133:   return o.index - index;
134: }
```

It should be more difficult for programmers to fix source files in this category than the *C1* and *C2* categories, since they have to consider the nodes that depend on each other. However, the added complexity may be an opportunity for automatic approaches. For example, a fault localization approach may use the data dependencies to locate relevant faulty lines of a located faulty line, and automatic program repair could use the data dependencies to prune its search space.

C. RQ3: Mutation Operator

As the underlying tool, ChangeDistiller [7], is built on the Eclipse's Java model[18], the code elements of extracted repair actions follow the APIs of Eclipse[19]. We find that the underlying tool has special strategies to extract repair actions on `Block` and `MethodDeclaration`. For example, it counts adding or deleting a method as an addition or a deletion on `MethodDeclaration`, but it counts only modifying a method name as a modification on `MethodDeclaration`. It does not count modifications inside a method (*e.g.*, the modifier, the parameters, and the statements in the method body) as a modification on `MethodDeclaration`, but counts them at finer levels. Here, as a method may have complicated structures, current program repair may not add a method effectively, although it is counted as a repair action.

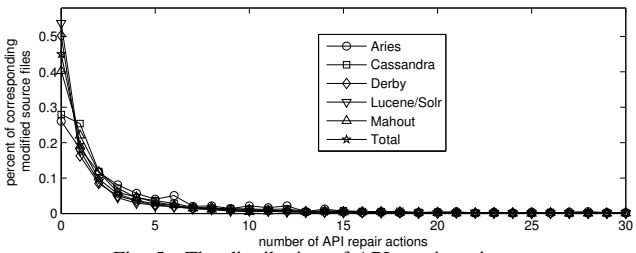

Fig. 5. The distribution of API repair actions.

As discussed in Section III-B, non-data dependent repair actions follow quite different patterns from data dependent repair actions, and current automatic program repair is effective to apply only non-data dependent repair actions. To eliminate the interference between the two types of repair actions and to provide valuable findings for the state-of-the-art, this study focuses only on non-data dependent repair actions. By definition, `Javadoc`, `Modifier`, `BreakStatement`, and `ContinueStatement` have only non-data dependent repair actions. All the other code elements have both data dependent and non-data dependent repair actions. The nature of code elements affects their ranks. For example, Pan *et al.* [25] show that a large portion of bug-fix patterns are related to `if` statements. In our study, we find that most repair actions on `if` statements are data dependent. As a result, its rank is low. The results show that automatic program repair should leverage data-flow analysis to effectively apply repair actions on this code element.

We calculated the number of repair actions on each code element, and Figure 4 shows the results. Its vertical axis shows the names of code elements. To save space, we do not present code elements whose repair actions are less than 1%. Its horizontal axis shows the number of repair actions with the categories such as additions, deletions, and modifications. The results lead to the following findings:

Finding 7. In total, repair actions on `Javadoc` rank the first. In the Eclipse's Java model, `Javadoc` denotes the comments between code. Programmers modified these code comments, since they contain errors or become inconsistent with the implementation. There exists research to address this problem. For example, our previous work [41] detects documentation errors (*e.g.* outdated code names in comments). As another example, Tan *et al.* [29] and Zhong *et al.* [44] infer formal rules from code comments and check the rules against the implementation to detect inconsistencies. Although the problem has been explored, there may still be space for improvement, since there are many modifications on `Javadoc`.

Finding 8. The repair actions on a code element typically increase with its complexity. For example, in Eclipse's Java model, `Expression` is complicated, since it has a rich set of subclasses[20]. `MethodInvocation` is one of its subclasses, and

[17]https://issues.apache.org/jira/browse/MAHOUT-591

[18]http://www.vogella.com/tutorials/EclipseJDT/article.html

[19]http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/reference/api/overview-summary.html

[20]http://help.eclipse.org/indigo/topic/org.eclipse.jdt.doc.isv/reference/api/org/eclipse/jdt/core/dom/Expression.html

TABLE II
THE DISTRIBUTION OF MODIFIED FILES

Aries	Cassandra	Derby	Lucene/Solr	Mahout	Total
java	java	java	java	java	java
xml	txt	out	txt	xml	txt
mdtext	py	properties	xml	CHANGELOG	xml
	xml	sql	html		out
		xml			properties
					sql

it may invoke complicated API methods. By definition, each `ExpressionStatement` has at least an `Expression` node, and each `ReturnStatement` can have an `Expression` node. As the inside nodes are complicated, the repair actions on the two types of statements rank the second and the third, in total. In contrast, `BreakStatement` and `ContinueStatement` are relatively simple, so the repair actions on the two types of statements are much fewer.

Finding 9. The actions on code elements follow two patterns. First, the modifications on a code element increase with its complexity. For example, `ExpressionStatement` is more complicated than `BreakStatement` as discussed before. Thus as shown in Figure 4, the former statement has more modifications than the latter statement. In fact, all the modifications of the latter statement are to move from one line to another, since it has no internal structures. Second, additions on a code element are more than deletions. When bugs are introduced, a careless programmer may forget some statements, and may also add unwanted statements. Our results show that the former case is more common than the latter case. Automatic program repair could use the two patterns to prune its search space, when they mutate the faulty code.

D. RQ4: API Knowledge

The underlying tool, PPA [4], parses and resolves full names of code elements. BUGSTAT considers a repair action as an API repair action, if the full name of its code element indicates that the code element is declared by third-party API libraries. If PPA fails to resolve the full name for a code element, BUGSTAT conservatively classifies the corresponding repair action into the non-API category. Figure 5 shows the distribution. Its horizontal axis shows the number of API repair actions, and its vertical axis shows the percentage of corresponding modified source files. Based on the results, we have the following findings:

Finding 10. In total, programmers did not make any API repair actions to fix half of the source files. The results explain why current automatic program repair is able to fix many bugs without API knowledge. By definition, some code elements (*e.g.*, `Javadoc`, `Modifier`, `BreakStatement`, and `ContinueStatment`) do not contain API elements, so repair actions on these code elements are not related to APIs. The repair actions on some other code elements have limited relation to APIs. For example, in Figure 4, `SwithCase` requires constants. If a constant in API libraries passes the type check, it may be used to fix bugs in `SwithCase`.

Finding 11. In total, programmers made at least one API repair action to fix the other half of source files. We find that complicated code elements tend to have more API repair actions. For example, as discussed before, `ExpressionStatement` is complicated, and we find many API repair actions on this code element. In particular, a bug report of Cassandra[21] says that it fails to read saved files, and the faulty line is as follows:

```
249:  in = new ObjectInputStream(...);
```

Line 249 is an `ExpressionStatement`. This statement calls `ObjectInputStream` to read files, but the files are saved with incompatible APIs. To fix the bug, programmers choose another API, and the modified line was as follows:

```
249:  in = new DataInputStream(...);
```

Robillard *et al.* [28] show that various approaches have been proposed to mine specifications for API libraries. As mined specifications describe legal API usage, they may be useful to fix API-related bugs. An interesting point to note is that most mined specifications describe the usage of only several API elements, according to our previous work (*e.g.* [43]). In Figure 5, a file has fewer than five API repair actions on average. The results indicate that an API usage typically involves only several API elements, regardless whether it is legal or illegal.

E. RQ5: File Type for Modifications

We calculated the file types of all the modified files, and Table II shows the distribution. In Table II, we use the suffices of file names to denote file types. To save space, we do not present file types that are less than 1%. Based on the results, we have the following findings.

Finding 12. The most common modified files are Java source files, and the other files are much fewer. The result is not surprising, since all the studied projects are in Java. However, we have noticed something interesting when we compare the results with results in Figure 1. Figure 1 shows that programmers did not modify Java source files to fix at least 10% of the bugs. As a result, on average, programmers modified fewer files when a bug did not involve Java source files than when it did. The result indicates that the dependency among Java source files may be higher than that among other files (*e.g.* configuration files).

Finding 13. The two most common modified non-source files are configuration files and natural language documents. The names of the most found configuration files end with "xml" or "properties", and we find three typical types of such files. The first type defines the parameters of build tools (*e.g.*, Ant[22]), and one such example is presented in Finding 1. The second type defines runtime parameters. For example, a bug report of

[21] https://issues.apache.org/jira/browse/CASSANDRA-2174
[22] http://ant.apache.org

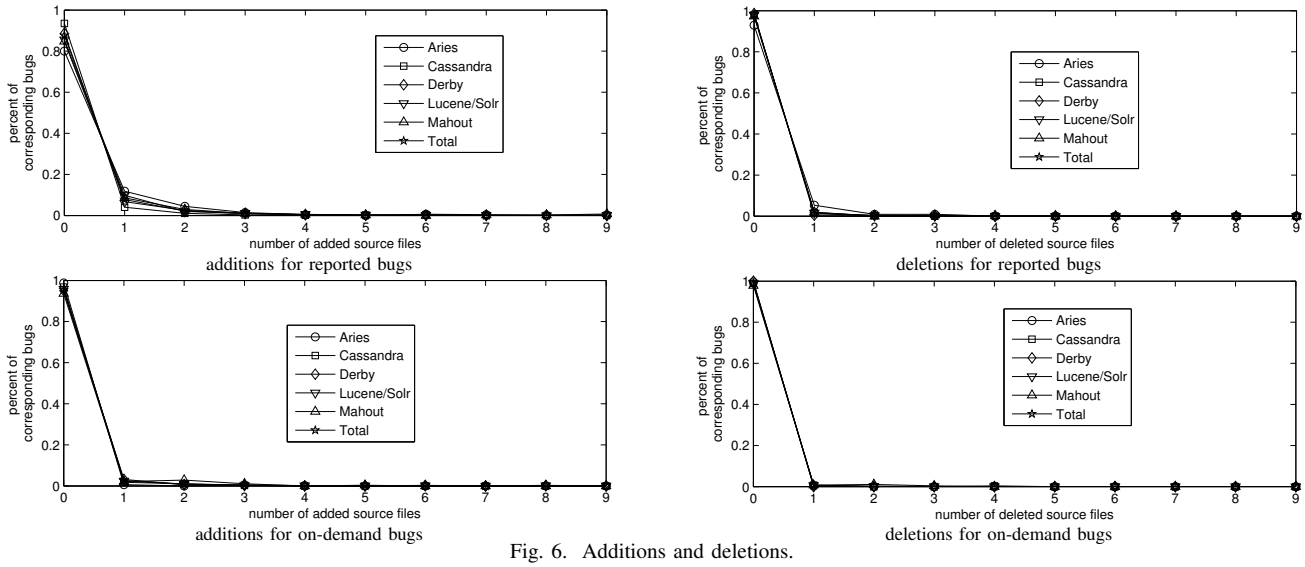

Fig. 6. Additions and deletions.

Cassandra[23] says that committing log can cause write pauses. To fix the bug, programmers modified code and enlarged the following old parameter:

```
312:<CommitLogSyncPeriodInMS>1000</CommitLogSyncPeriodInMS>
```

The modified line was as follows:

```
312:<CommitLogSyncPeriodInMS>10000</CommitLogSyncPeriodInMS>
```

The third type defines parameters of third-party tools. For example, many programmers use Findbugs[24] to detect bugs, and the tool has a configuration file to enable specific checks. We find that the programmers of Mahout added several lines to the Findbugs' configuration file to enable a check:

```
40:<Match>
41: <Bug pattern="SE_NO_SUITABLE_CONSTRUCTOR"/>
42:</Match>
```

The names of most found natural language documents end with "html" or "txt". These documents are manuals, tutorials and change logs. The results highlight the importance of fixing bugs in configuration files and natural language documents.

Finding 14. Some modified source files are in programming languages other than Java for two reasons. First, a project may be implemented in multiple programming languages. For example, Cassandra is a database, and its programmers implement a Python driver. A bug report[25] says that the driver did not parse queries correctly. To fix the bug, programmers modified one faulty line of the `cursor.py` file:

```
39:_cfamily_re = re.compile("...", re.I | re.M)
```

The modified line was as follows:

```
39:_cfamily_re = re.compile("...",
      re.IGNORECASE | re.MULTILINE | re.DOTALL)
```

Second, a project may implement an interface for a programming language. For example, Derby is a database that supports queries in SQL, and its programmers use SQL queries as test cases. A bug report of Derby[26] says that when an error code was returned when a query casted `DATE` to `TIMESTAMP`. To reproduce the error, a programmer added a line to the `cast.sql` file:

```
476: select cast(t as timestamp) from tab1;
```

On this line, `t` is a column of table `tab1`, and the type of `t` is `time`, and the `cas.out` file recorded the output:

```
ij> select cast(t as timestamp) from tab1;
ERROR 42846: Cannot convert types 'TIME' to 'TIMESTAMP'.
```

The two situations highlight the importance of fixing bugs in multiple programming languages. As automatic program repair has been evaluated on only a limited number of programming languages, such as C and Java, it may require significant improvement to fix bugs in other programming languages. For example, it is challenging to instrument SQL queries to collect their executed traces. Without such traces, it is infeasible to use fault localization to locate faulty lines in such queries.

F. RQ6: Additions and Deletions

We calculated the number of added and deleted files for each bug fix, and Figure 6 shows the distribution. Its horizontal axes show the number of added or deleted source files, and its vertical axes show the corresponding percentage. Based on the results, we have the following finding:

Finding 15. In total, programmers did not add any files to fix more than 80% of the bugs, and they did not delete any files to fix more than 90% of the bugs. Hattori and Lanza [9] analyze the nature of commits, and they find that the size of a commit is associated with the type of the commit. In particular, tiny commits are more related to bug fixes, and large commits are more related to new features. Our result reflects another nature of commits, namely at the file level, modifications may be more related to bug fixes. As most bug fixes require only modifying files, it is reasonable for automatic program repair to focus on mutating files to fix bugs.

G. Threats to Validity

The threat to internal validity includes the defects in the underlying tools that we use. ChangeDistiller may produce

[23]https://issues.apache.org/jira/browse/CASSANDRA-668
[24]http://findbugs.sourceforge.net
[25]https://issues.apache.org/jira/browse/CASSANDRA-2993
[26]https://issues.apache.org/jira/browse/DERBY-896

infeasible edit scripts, and PPA may wrongly resolve code elements. To reduce this threat, we reported our found defects, and if they are not fixed, we tried to fix them by ourselves. The threat could be further reduced by developing more advanced tools. The threat to external validity includes our selected subjects. Although we analyzed thousands of bug fixes in total, the selected projects may still be limited and are all for Java. It is likely that most our findings still hold in other programming languages, but the specific numbers may be different. For example, although the repair actions on a code element increase with the complexity of its usage (Finding 8) in other programming languages, the ranks of code elements will be different, since other programming languages may define quite different code elements. To reduce this validity, our study should be replicated in future work by using subjects in other programming languages.

IV. Discussions and Future Work

Bug definitions. Practitioners and researchers can have different definitions of bugs. Herzig *et al.* [10] claimed that at least 30% of reported bugs are not bugs, but features. Even researchers may have different definitions of bugs. For example, in our previous work [41], we reported our detected documentation errors as bugs, and programmers accepted and fixed them as bugs. However, by the definition of Herzig *et al.*, our detected errors are documentation requests, but not bugs. The different definitions lead to quite different results. In our study, we follow the pragmatic definition of practitioners, *i.e.*, those issues that are reported and fixed as bugs. The benefit is that the results reflect the reality of practice, and practitioners do not need to read all the definitions to understand our results.

More factors of automatic program repair. We have focused on the major factors of current automatic program repair, and due to space limit, we certainly miss some factors that are also relevant. For example, the domain knowledge of compilers, tool chains, programming models, multi-programming and concurrent, and the low-level knowledge of operating systems and hardware are essential to fix some bugs. Murphy-Hill *et al.* [23] present various factors, when programmers manually fix bugs. In addition, Luo *et al.* [20] show that bugs can not be easily reproducible or verifiable, which introduces extra barriers to automatic program repair. In future work, we plan to conduct studies to investigate the importance of these factors.

Manual fixes vs. automatic fixes. Neither manual fixes nor automatic fixes are perfect. Although Yin *et al.* [39] show that manual fixes can be incorrect, our results are still reliable, since most fixes are correct. At the same time, a program fix may pass test cases, but does not fix the real problem. Bird *et al.* [3] show that even manual fixes should be carefully examined, so it is likely that the correctness of program fixes also needs to be verified. Although we agree with Monperrus [22] that understandability of patches is inessential in certain situations, in most cases, generated patches should be readable to humans or other programs for verification. In addition, although it is a practical way to mimic humans and it is a good way to understand the challenges by analyzing manual fixes, we agree that there could be alternative ways to solve the problem. For example, computers can already beat the best human players in playing chess, and the algorithm is quite different from humans, since computers have much larger (and more reliable) memories and much stronger computational capabilities than most humans [1]. With advanced techniques, it is conceivable that a computer algorithm may produce better patches than we humans do.

V. Related Work

Automatic program repair. Weimer *et al.* [33] proposed GenPro, a seminal work on automatic program repair. Le Goues *et al.* [17] extended GenPro, and proposed new mutation and crossover operators. Kim *et al.* [12] manually inspected thousands of bug fixes and summarized ten templates as new mutation operators. Wei *et al.* [32] used mined invariants to fix bugs. Jin *et al.* [11] proposed new mutation operators and selectors that are designed to fix concurrent bugs. All the preceding approaches define a limited number of simple repair shapes, and rely on genetic algorithm to generate complicated repair shapes to fix complicated bugs. Qi *et al.* [26] show that random search is more effective than genetic algorithm to guide the bug fix process. Martinez and Monperrus [21] mine repair models from manual fixes, and the mined repair models improve random search. Our study provides findings and insights to better understand and improve these approaches.

Fault localization. Fault localization has been an extensive studied topic. Wong and Debroy [34] provided a comprehensive survey on this line of research. Most of the approaches (*e.g.* [8]) assume that each program has exactly one faulty location, and a few recent approaches (*e.g.* [19]) start to explore the links between traces and multiple faulty locations. Our study shows that future research should further improve existing approaches to locate bugs in multiple locations.

Empirical study on bug fixes. Various empirical studies exist to understand the nature of bug fixes. Yin *et al.* [39] show that bug fixes could introduce new bugs. Nguyen *et al.* [24] show that repetitiveness is common in small size bug fixes. Eyolfson *et al.* [6] show that the bugginess of a commit is correlated with the time to make the commit. Bird *et al.* [2] show that many projects did not carefully maintained the links between bug reports and bug fixes. Our empirical study has analyzed bug fixes to gain insights on automatic program repair, providing a different research angle than previous studies. Thung *et al.* [30] manually examined bug fixes; their results show that faults are not localized. Our results are largely consistent with their findings. However, our results are built on much larger samples, since we analyzed bug fixes automatically. In addition, different from theirs, our study is not limited to fault localization.

VI. Conclusion

To reduce the effort of fixing bugs, various approaches have been proposed to automatically repair programs. These approaches show promising results, but their effectiveness on real-world bugs has also been questioned and criticized. Directly evaluating tools from these efforts on real bugs may

produce many trivial and negative results, since they are still immature research prototypes. Instead, we have conducted a large-scale empirical study on thousands of real-world bug fixes from five popular projects. We have compared how programmers fix bugs and how bug-fix approaches are designed, and distilled 15 findings and four insights that we believe are useful to guide future research in this direction.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. This research was sponsored in part by the National Basic Research Program of China (973) No. 2015CB352203, the National Science Foundation of China No. 61100071, 61272102, and 91118004, and by United States NSF Grants 1117603 and 1319187. The information presented here does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.

REFERENCES

[1] H. J. Berliner. Backgammon computer program beats world champion. *Artificial Intelligence*, 14(2):205–220, 1980.

[2] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. Devanbu. Fair and balanced?: bias in bug-fix datasets. In *Proc. ESEC/FSE*, pages 121–130, 2009.

[3] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proc. 19th FSE*, pages 4–14, 2011.

[4] B. Dagenais and L. J. Hendren. Enabling static analysis for partial Java programs. In *Proc. 23rd OOPSLA*, pages 313–328, 2008.

[5] N. DiGiuseppe and J. A. Jones. On the influence of multiple faults on coverage-based fault localization. In *Proc. ISSTA*, pages 210–220, 2011.

[6] J. Eyolfson, L. Tan, and P. Lam. Correlations between bugginess and time-based commit characteristics. *Empirical Software Engineering*, 19(4):1009–1039, 2014.

[7] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering*, 33(11):725–743, 2007.

[8] D. Hao, L. Zhang, Y. Pan, H. Mei, and J. Sun. On similarity-awareness in testing-based fault localization. *Automated Software Engineering*, 15(2):207–249, 2008.

[9] L. P. Hattori and M. Lanza. On the nature of commits. In *Proc. 23rd ASE-Workshops*, pages 63–71, 2008.

[10] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proc. 35th ICSE*, pages 392–401, 2013.

[11] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu. Automated concurrency-bug fixing. In *Proc. 10th OSDI*, pages 221–236, 2012.

[12] D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proc. 35th ICSE*, pages 802–811, 2013.

[13] D. Kim, Y. Tao, S. Kim, and A. Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE Transactions on Software Engineering*, 39(11):1597–1610, 2013.

[14] S. Kim and E. J. Whitehead Jr. How long did it take to fix bugs? In *Proc. 3rd MSR*, pages 173–174, 2006.

[15] S. Kim, H. Zhang, R. Wu, and L. Gong. Dealing with noise in defect prediction. In *Proc. 33rd ICSE*, pages 481–490, 2011.

[16] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proc. 29th ICSE*, pages 489–498, 2007.

[17] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proc. 34th ICSE*, pages 3–13, 2012.

[18] G. Liang, Q. Wang, T. Xie, and H. Mei. Inferring project-specific bug patterns for detecting sibling bugs. In *Proc. ESEC/FSE*, pages 565–575, 2013.

[19] C. Liu and J. Han. Failure proximity: a fault localization-based approach. In *Proc. 14th FSE*, pages 46–56, 2006.

[20] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov. An empirical analysis of flaky tests. In *Proc. 22nd FSE*, pages 643–653, 2014.

[21] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, pages 1–30, 2013.

[22] M. Monperrus. A critical review of "automatic patch generation learned from human-written patches": Essay on the problem statement and the evaluation of automatic software repair. In *Proc. 36th ICSE*, pages 234–242, 2014.

[23] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design of bug fixes. In *Proc. 35th ICSE*, pages 332–341, 2013.

[24] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan. A study of repetitiveness of code changes in software evolution. In *Proc. 28th ASE*, pages 180–190, 2013.

[25] K. Pan, S. Kim, and E. J. Whitehead Jr. Toward an understanding of bug fix patterns. *Empirical Software Engineering*, 14(3):286–315, 2009.

[26] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. The strength of random search on automated program repair. In *ICSE*, pages 254–265, 2014.

[27] Y. Qi, X. Mao, Y. Lei, and C. Wang. Using automated program repair for evaluating the effectiveness of fault localization techniques. In *Proc. ISSTA*, pages 191–201, 2013.

[28] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford. Automated API property inference techniques. *IEEE Transactions on Software Engineering*, 39(5):613–637, 2013.

[29] L. Tan, D. Yuan, G. Krishna, and Y. Zhou. /* iComment: Bugs or Bad Comments?*/. In *Proc. 21st SOSP*, pages 145–158, 2007.

[30] F. Thung, D. Lo, and L. Jiang. Are faults localizable? In *Proc. 9th MSR*, pages 74–77, 2012.

[31] Y. Tian, J. Lawall, and D. Lo. Identifying linux bug fixing patches. In *Proc. 34th ICSE*, pages 386–396, 2012.

[32] Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proc. 19th ISSTA*, pages 61–72, 2010.

[33] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proc. 31st ICSE*, pages 364–374, 2009.

[34] W. E. Wong and V. Debroy. A survey of software fault localization. Technical report, University of Texas at Dallas, 2009.

[35] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung. Relink: recovering links between bugs and changes. In *Proc. ESEC/FSE*, pages 15–25, 2011.

[36] Y. Xiong, A. Hubaux, S. She, and K. Czarnecki. Generating range fixes for software configuration. In *Proc. 34th ICSE*, pages 58–68, 2012.

[37] X. Ye, R. Bunescu, and C. Liu. Learning to rank relevant files for bug reports using domain knowledge. In *Proc. FSE*, pages 66–76, 2014.

[38] 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. In *Proc. 23rd SOSP*, pages 159–172, 2011.

[39] Z. Yin, D. Yuan, Y. Zhou, S. Pasupathy, and L. Bairavasundaram. How do fixes become bugs? In *Proc. ESEC/FSE*, pages 26–36, 2011.

[40] M. Zhivich and R. K. Cunningham. The real cost of software errors. *IEEE Security & Privacy*, 7(2):87–90, 2009.

[41] H. Zhong and Z. Su. Detecting API documentation errors. In *Proc. SPASH/OOPSLA*, pages 803–816, 2013.

[42] H. Zhong, S. Thummalapenta, T. Xie, L. Zhang, and Q. Wang. Mining API mapping for language migration. In *Proc. 32nd ICSE*, pages 195–204, 2010.

[43] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending API usage patterns. In *Proc. 23rd ECOOP*, pages 318–343, 2009.

[44] H. Zhong, L. Zhang, T. Xie, and H. Mei. Inferring resource specifications from natural language API documentation. In *Proc. 24th ASE*, pages 307–318, 2009.

[45] 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 *Proc. 34th ICSE*, pages 14–24, 2012.