BugBuilder: An Automated Approach to Building Bug Repository

Yanjie Jiang, Hui Liu, Xiaoqing Luo, Zhihao Zhu, Xiaye Chi, Nan Niu,
Yuxia Zhang, Yamin Hu, Pan Bian, and Lu Zhang

Abstract—Bug-related research, e.g., fault localization, program repair, and software testing, relies heavily on high-quality and large-scale software bug repositories. The importance of such repositories is twofold. On one side, real-world bugs and their associated patches may inspire novel approaches for finding, locating, and repairing software bugs. On the other side, the real-world bugs and their patches are indispensable for rigorous and meaningful evaluation of approaches to software testing, fault localization, and program repair. To this end, a number of software bug repositories, e.g., iBUGS and Defects4J, have been constructed recently by mining version control systems and bug tracking systems. However, fully automated construction of bug repositories by simply taking bug-fixing commits from version control systems often results in inaccurate patches that contain many bug-irrelevant changes. Although we may request experts or developers to manually exclude the bug-irrelevant changes (as the authors of Defects4J did), such extensive human intervention makes it difficult to build large-scale bug repositories. To this end, in this paper, we propose an automatic approach, called *BugBuilder*, to construct bug repositories from version control systems. Different from existing approaches, it automatically extracts complete and concise bug-fixing patches and excludes bug-irrelevant changes. It first detects and excludes software refactorings involved in bug-fixing commits. *BugBuilder* then enumerates all subsets of the remaining part, and discards invalid subsets by compilation and software testing. If exactly a single subset survives the validation, this subset is taken as the complete and concise bug-fixing patch for the associated bug. In case multiple subsets survive, BugBuilder employs a sequence of heuristics to select the most likely one. Evaluation results on 809 real-world bug-fixing commits in Defects4J suggest that *BugBuilder* successfully extracted complete and concise bug-fixing patches from forty-three percent of the bug-fixing commits, and its precision (99%) was even higher than human experts. We also built a bug repository, called GrowingBugs, with the proposed approach. The resulting repository serves as evidence of the usefulness of the proposed approach, as well as a publicly available benchmark for bug-related research.

Index Terms—Bug, Defect, Testing, Patch, Repository, Dataset, Refactoring

1 Introduction

High-quality and large-scale bug repositories are urgently needed by research in the software engineering community for various reasons, e.g., fault localization, software testing, program repair, and prediction of bugs. The benefits of bug repositories are twofold. On one side, real-world bugs and their concise patches are indispensable for rigorous evaluation of numerous automatic or semi-automatic approaches to localizing faulty statements [1], [2], [3], [4], [5], to predicting the number of bugs in software applications [6], [7], [8], [9], and to repairing faulty applications [10], [11], [12], [13], [14], [15]. Because we expect such approaches to work well on real-world applications, it is critical to evaluate such approaches with a large number of real-world bugs (and their corresponding patches) from real-world applications before they could be widely applied in the wild [16], [17]. Although automatically generated mutants or manually injected bugs could also be exploited for the evaluation [18], they could be essentially different from real-world bugs, and thus conclusions drawn on them may not hold on real-world bugs. On the other side, real-world bugs and their concise patches may also inspire researchers to propose novel approaches to finding, localizing, and repairing software bugs. For example, by analyzing a large number of real-world bugs, researchers may figure out what kind of statements are more error-prone, and thus they could try to repair such statements first during automatic program repair [19] to improve the efficiency of program repair. Another example is that researchers have discovered many common fix patterns by reading human-written patches [20]. Leveraging such patterns in turn has significantly increased the performance of automatic program repair [20]. Finally, data-driven and learning-based approaches in automatic program repair [21], [22], [23] and bug detection [24] usually depend on a large number of diverse real-world bugs and their concise patches. Notably, the quality of these bugs, e.g., the diversity of the bugs and the accuracy of the patches, could significantly influence the performance of such data-driven approaches.

To this end, a few bug repositories have been constructed to facilitate bug-related research [16]. According to their construction, we divide such repositories into the following

- *Y.J. Jiang, H. Liu, X.Q. Luo, Z.H. Zhu, X.Y. Chi, Y.X. Zhang, and Y.M. Hu are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: yanjiejiang@bit.edu.cn, liuhui08@bit.edu.cn, ccluo33@163.com, 2899329254@qq.com, chixiaye@icloud.com, yuxiazh@bit.edu.cn,ymhu@bit.edu.cn.*
- *N. Niu is with Department of EECS, University of Cincinnati. Cincinnati, OH 45221. E-mail: nan.niu@uc.edu.*
- *P. Bian is with Huawei Technologies Co., Ltd, Beijing, China. E-mail: bianpan@huawei.com*
- *L. Zhang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education (Peking University), Beijing 100871. E-mail: zhanglu@sei.pku.edu.cn*
- *Corresponding Author: Hui Liu*

three categories. The first category of bug repositories is constructed manually. Typical examples of this category include SIR [18], BugBench [25], IntroClass [26], Codeflaws [27], QuixBugs [28], DroixBench [29], and DBGBench [30]. These repositories are constructed manually, and thus all of them are quite limited in scale and diversity. The second category of bug repositories is constructed automatically. For example, iBUGS [17] and ManyBugs [26] were constructed by automatically extracting bug-fixing commits as bug-fixing patches. Although such automated construction could result in large bug repositories with great diversity, the quality of the patches is questionable. Existing studies [16], [31] suggest that bug-fixing commits often contain bug-irrelevant changes, e.g., refactorings. Consequently, if we take all of the changes in a bug-fixing code commit as the patch of the associated bug report, the resulting patch could be inconcise, and contain code changes irrelevant to the bug. A code change is bug-irrelevant if it changes/adds/removes functionalities that are not associated with the bug report or it does not change the external behaviors of the system (e.g., software refactorings). A complete and concise bug-fixing patch should include all bug-relevant changes (complete) and exclude all bug-irrelevant changes (concise). The third category of bug repositories is constructed semi-automatically. A typical example is the well-known Defects4J [16]. To extract complete and concise bug-fixing patches from bug-fixing commits, Defects4J automatically takes all changes in a bug-fixing commit as an initial patch, and then manually excludes bug-irrelevant changes from it. As a result, the resulting patch is highly accurate, often both complete and concise. However, such extensive human intervention makes it difficult and expensive to increase the size of the repository. Consequently, the scale of Defects4J and the diversity of its patches remain limited.

To automate the construction of large-scale and high-quality bug repositories, in this paper, we propose an automatic approach, called *BugBuilder*, to extracting concise and complete bug-fixing patches from human-written patches in version control systems. For each bug-fixing commit, it works as follows. First, it identifies refactorings within the bug-fixing commit by an existing tool (i.e., *RefactoringMiner* [32]), and removes refactorings from human-written patches by reapplying the identified refactorings to the buggy version of the application. Second, it automatically generates all potential patches by enumerating all possible combinations of the remaining non-refactoring changes. Third, it validates the potential patches on test cases. If all but one potential patch are invalid, the remaining one is deemed as a complete and concise patch. If multiple ones pass the validation, BugBuilder leverages a sequence of highly accurate heuristics to select the most likely patch. Notably, if the human-written patch is composed of both refactorings and bug-fixing changes, BugBuilder splits it into two ordered patches: a refactoring patch and a following bug-fixing patch. It is highly similar to Defects4J which splits a human-written patch into a bug-irrelevant patch and a following bug-fixing patch.

The evaluation of the proposed approach is composed of two parts. In the first part, we applied the proposed approach (BugBuilder) to the 809 real-world bug-fixing commits collected by Defects4J. On each of the evaluated commits, we leveraged BugBuilder to extract concise patches automatically. If a patch was successfully generated, we compared it against the manually constructed patch provided by Defects4J. On the 809 bug-fixing commits, BugBuilder automatically generates 350 patches where 334 were identical to manually constructed patches in Defects4J. For the other 16 patches that are different from manually constructed patches, we manually analyzed the associated bug reports as well as the code commits. Our evaluation results suggest that out of the 16 pairs of mismatched patches, 12 were caused by incomplete patches in Defects4J whereas the generated patches were complete and concise. Only four out of the 350 generated patches were inaccurate (complete but not concise), and all of them were caused by incomplete detection of refactorings. In the second part of the evaluation, we constructed a large bug repository by applying the proposed approach to open-source applications. The resulting repository, called *GrowingBugs*, is composed of 1,491 real-world bugs and their concise patches, automatically collected from 169 well-known and widely used Java applications.

This is an expanded version of our previous conference paper [33]. Compared against the conference version, this paper makes the following expansions:

- We further improve the approach by proposing a sequence of heuristics to select the correct patch from multiple candidate patches. In the conference version, the proposed approach gives up when there are more than one candidate patch for a single bug-fixing commit, which has a significant and negative influence on the recall of the proposed approach. Selecting patches with the strategies proposed in this paper improves the recall of the proposed approach by 8% whereas the precision keeps untouched. The implementation of the improved approach is published on GitHub[1].
- We build a high-quality bug repository, called GrowingBugs[2], with the proposed approach. On one side, successfully building the repository with the proposed approach further validates the usefulness of the proposed approach. On the other side, the resulting repository can serve as a benchmark to facilitate future bug-related research, especially automated program repair and fault localization. To the best of our knowledge, it is the largest Java bug repository composed of real-world bugs and concise patches.

The rest of this paper is structured as follows. Section 2 introduces related work. Section 3 motivates the research with a motivating example, and Section 4 presents the details of the proposed approach. Section 5 presents the evaluation of the proposed approach, and Section 6 builds a new bug repository with the proposed approach. Section 7 discusses some closely related issues, and Section 8 concludes the paper.

2 Related Work

2.1 Bug Repositories

The importance of large-scale and high-quality bug repositories is well-recognized, and thus a number of bug repos-

1. https://github.com/liuhuigmail/BugBuilder
2. https://github.com/liuhuigmail/GrowingBugRepository

itories have been constructed recently. To the best of our knowledge, the software-artifact infrastructure repository (SIR) [18] is the first attempt in the construction of bug repositories. It is composed of bugs from 17 C programs and 7 Java programs. Each of the programs has several different faulty versions, and each faulty version contains several known bugs and associated test suites to reveal the bugs. However, the bugs in SIR are different from real-world bugs because such bugs were hand-seeded or obtained from mutation [34]. It may reduce the usability of the bug repository.

Some bug repositories were constructed by collecting real-world bugs from assignments or competitions. For example, Spacco et al. [35] collected real-world bugs made by students during programming tasks. Their resulting bug repository contains hundreds of faulty projects accompanied by test cases. IntroClass [26] proposed by Le et al., Codeflaws [27] proposed by Tan et al., and QuixBugs [28] proposed by Lin et al. contain real-world bugs from programming competitions/challenges. However, such bugs made in programming assignments or competitions could be significantly different from real-world bugs in industry, and thus bug-related approaches validated on such repositories may not work in industry.

Some bug repositories were constructed by manually collecting real-world bugs from real-world applications. For example, Lu et al. [25] manually collected 19 real-world bugs from 17 programs, Tan et al. [29] manually collected 24 reproducible crashes from 15 open-source Android Apps, and Böhme et al. [30] requested twelve experts to collect 27 real-world bugs from open-source C projects. Péter et al. [36] manually validated 453 real-world JavaScript bugs from JavaScript server-side programs. Although such real-world bugs are of high quality, manual collection of real-world bugs is tedious and time-consuming, and thus the sizes of such bug repositories are often limited. As a result, the diversity of the bugs is limited.

Automatic and semi-automatic approaches have been proposed to collect real-world bugs. For example, iBUGS [17] extracted 369 bugs automatically from version control systems, and took the whole bug-fixing commits as bug-fixing patches with the assumption that all changes in the bug-fixing commits are bug-related. However, existing studies (e.g., [16], [31]) suggested that bug-fixing commits often contain bug-irrelevant changes, e.g., implementation of new features and software refactorings that do not change software functionalities. Consequently, taking all changes in bug-fixing commits as bug-fixing patches may result in *unconcise patches*: A patch is deemed unconcise if it contains bug-irrelevant changes. The authors of ManyBugs [26] also took the whole bug-fixing commit as a patch, and thus the patches in the repository are potentially unconcise. Bugs.jar [37], containing 1,158 real-world bugs and patches automatically collected from open-source applications (Apache@GitHub), is another large-scale bug repository that took the whole bug-fixing commits as bug-fixing patches. Evaluating bug-related approaches with such unconcise patches could result in misleading conclusions, and the unconcise patches may make it harder to assess patch correctness in automated program repair.

Defects4J [16] isolated bug-fixing changes from bug-irrelevant changes manually. Each bug in Defect4J is associated with three versions: the original buggy version (called V_{n-1}), the fixed version (called V_n), and an intermediate version (called V_{bug}). Applying the bug-irrelevant changes to the original buggy version (V_{n-1}) results in the intermediate version V_{bug}. Applying the concise bug-fixing patch in Defects4J to V_{bug} fixes the bug and results in the fixed version V_n. Notably, both the original buggy version and the fixed version were retrieved from the version control systems. However, the intermediate version was created manually by experts to contain all bug-irrelevant changes. As a result of the manual isolation, bugs and patches in Defects4J are highly accurate. Consequently, Defects4J becomes one of the most frequently used bug repositories in the software engineering community. To evaluate automated program repair tools/algorithms (called APR tools for short) with Defects4J, we should apply the APR tools to V_{bug} (instead of the original buggy version V_{n-1}) to generate patches, and compare the generated patches against the patch in Defects4J. Besides APR tools, fault localization tools/algorithms may also leverage the patches in Defects4J for quantitative evaluation (taking V_{bug} as the buggy program to be fixed). Another significant advantage of Defects4J is that it provides an extensible framework to enable reproducible studies in software testing research. However, the manual intervention requested by Defects4J prevents it from being fully automatic, and thus the repository remains limited in scale and diversity.

Some bugs were collected automatically according to Continuous Integration (CI). For example, BEARS proposed by Madeiral et al. [38] finds potential pairs of buggy and patched program versions from open-source projects according to commit building state from Continuous Integration (CI). The core step of its bug collection is the execution of the test suite of the program on two consecutive versions. If a test failure is found in the faulty version and no test failure is found in its patched version, the authors of BEARS take the two versions as the faulty and the fixed versions respectively whereas their difference is taken as the associated patch. Notably, the difference may contain some bug-irrelevant changes, e.g., software refactorings. However, BEARS does not distinguish bug-fixing changes from bug-irrelevant changes within the same source code commit. Similar to BEARS, Tomassi et al. [39] also collected bugs/patches according to CI and did not exclude bug-irrelevant changes from the collected patches, either. As a result, all such patches could be unconcise.

Based on the preceding analysis, we conclude that existing bug repositories are limited in either scale or quality. Manually or semi-automatically constructed repositories are limited in scale because they request extensive human intervention. In contrast, automatically constructed ones could be inaccurate because the automatically extracted patches often contain bug-irrelevant changes [31].

2.2 Distinguishing Special Changes in Code Commits

Approaches have been proposed to distinguish some special changes in code commits. For example, DiffCat proposed by Kawrykow and Robillard [40] identifies non-essential changes (especially refactorings) in version histories, highly

similar to RefactoringMiner [32] that our approach leverages to identify refactorings. It leverages CHANGEDISTILLER [41] to detect changes between two successive versions, and then it identifies non-essential changes by comparing the changes to a set of predefined types of fine-grained non-essential changes. RefactoringMiner [32] exploits an AST based approach to match software entities in different versions of the same application, and then identifies refactorings based on the entity matching by a sequence of heuristic rules. Notably, RefactoringMiner was specially designed to detect refactorings whereas DiffCat is not confined to refactorings, and thus the latter may identify some non-refactoring and non-essential changes, e.g., changes in code format (like removing white spaces). RefactoringMiner, after a long-term evolution, covers more refactoring types than DiffCat. DiffCat and RefactoringMiner at best may serve as only the first step for concise patch extraction (see Fig.1 for more details). Simply recommending refactoring-excluded patches (output of DiffCat or RefactoringMiner) would result in numerous unconcise patches when commits contain non-refactoring bug-irrelevant changes. As a result, developers need to manually check/clean all of the recommended patches to guarantee the quality.

Thung et al. [42] proposed an approach to identifying root causes of bugs. Root causes of a bug refer to the *lines of code in the buggy version that are responsible for the bug*. Notably, root causes are essentially different from concise patches. Consequently, these approaches [40], [42] do not address the same issue as we do, i.e., automatically constructing bug-fixing patch repositories. Notably, neither Thung et al. [42] nor Kawrykow and Robillard [40] leverage off-the-shelf refactoring mining tools, and neither of them reapply discovered refactorings as we do.

```
1  diff --git a/src/main/java/com/fasterxml/jackson/
       databind/deser/DeserializerCache.java b/src/main/
       java/com/fasterxml/jackson/databind/deser/
       DeserializerCache.java
2  index bdb2144..ded6b68 100644
3  --- a/src/main/java/com/fasterxml/jackson/databind/deser
       /DeserializerCache.java
4  +++ b/src/main/java/com/fasterxml/jackson/databind/deser
       /DeserializerCache.java
5  @@ -204,7 +204,7 @@ public final class DeserializerCache
6     if (type == null) {
7        throw new IllegalArgumentException("Null JavaType
             passed");
8     }
9  -  if (_hasCustomValueHandler(type)) {
10 +  if (_hasCustomHandlers(type)) {
11       return null;
12    }
13       return _cachedDeserializers.get(type);
14 @@ -274,7 +274,7 @@ public final class DeserializerCache
15 - boolean addToCache = !_hasCustomValueHandler(type)
       && deser.isCachable();
16 + boolean addToCache = !_hasCustomHandlers(type)
       && deser.isCachable();
17
18 @@ -531,13 +531,23 @@ public final class
       DeserializerCache
19 -  private boolean _hasCustomValueHandler(JavaType t) {
20 +  private boolean _hasCustomHandlers(JavaType t) {
21    if (t.isContainerType()) {
22       JavaType ct = t.getContentType();
23       if (ct != null) {
24 - return (ct.getValueHandler() != null) ||
       (ct.getTypeHandler() != null);
25 + if ((ct.getValueHandler() != null) ||
       (ct.getTypeHandler() != null)) {
26 +       return true;
27 +    }
28 + }
29 +  if (t.isMapLikeType()) {
30 +     JavaType kt = t.getKeyType();
31 +     if (kt.getValueHandler() != null) {
32 +        return true;
33 +     }
34    }
35 }
36    return false;
```

Listing 1. A Bug-Fixing Commit from JacksonDatabind

3 Motivating Example

This section explains why it is challenging to extract concise patches from bug-fixing commits automatically. Listing 1 presents a motivating example for illustration. It is a bug-fixing commit extracted from JacksonDataind [43], and the associated bug report is publicly available online [44]. The changes involved in the commit are highlighted in standard *diff* style. Red lines beginning with '-' are removed by the commit whereas green lines beginning with '+' are inserted by the commit. Other lines are untouched.

The changes within such bug-fixing commit is composed of two parts. The first part is to rename method "*_hasCustomValueHandler*" as "*_hasCustomHandlers*" (Lines 19-20), and to update the method invocation of the renamed method (Lines 9-10 and 15-16). The second part of the commit is to fix the reported bug: "*Using org.apache.logging.log4j.core.jackson.Log4jJsonObjectMapper to deserialize the appended JSON object is throwing an exception with 2.9.2 but worked with 2.9.1.*". When the parameter *t* is a *map*, the method should return *true* if the Java type of the keys in the *map* has value handler. The buggy version decides to return *true* or *false* based on the type of the content only (i.e., *ct* on Line 22) and thus it would result in incorrect return value when *t* is a *map*. To this end, the commit deletes the *return* statement on Line 24 and inserts two *if* statements on Lines 25-28 and Lines 29-33 to handle *map* as a special case.

For the motivation example in Listing 1, it would result in an unconcise patch that contains bug-irrelevant changes (i.e., the *rename method* refactoring on Lines 9-10, 15-16, and 19-20) if we simply extract all changes in the bug-fixing commit as the patch. BugBuilder successfully extracts the complete and concise patch (i.e., the changes on Lines 24-33) from the bug-fixing commit. It works as follows.

- First, it discovers the *rename method* refactoring by analyzing the changes made in the commit.
- Second, it reapplies the identified refactoring to the faulty version, resulting in a new version V'_{n-1}, called *refactoring-included* version.
- Third, it computes all changes between V'_{n-1} and the fixed version (V_n). The resulting changes are noted as $Chgs$. Notably, $Chgs$ does not include the *rename method* refactoring because it has already been done on V'_{n-1}. Consequently, $Chgs$ is composed of the changes on Lines 24-33 only.
- Fourth, BugBuilder enumerates and validates all possible subsets of $Chgs$. Such subsets are called potential patches. Only a single potential patch (*ptch*=$Chgs$) passes the validation whereas applying any of the other

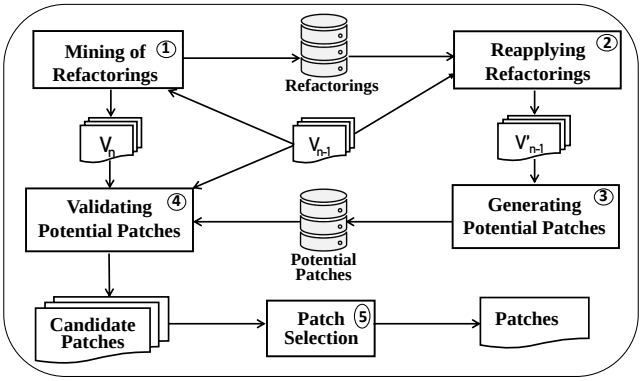


Fig. 1. Overview of BugBuilder

potential patches to V'_{n-1} results in compiler errors or fails to pass any new test cases in the fixed version. Consequently, *ptch* is the only candidate patch that survives the validation.

- Finally, BugBuilder outputs *ptch* as the patch for the associated bug because it is the only candidate patch.

4 Approach

4.1 Overview

Fig. 1 presents an overview of the proposed approach BugBuilder. It takes as input two consecutive versions of a software application, i.e., V_{n-1} and V_n. The latter version V_n is called V_{fix} or the *fixed version* whereas V_{n-1} is called the *faulty version*. The two consecutive versions are accompanied by two test suites, noted as T_{n-1} and T_n, respectively. T_{n-1} and T_n exclude broken test cases that fail on their associated version of the application. The evolution from V_{n-1} to V_n is driven by a bug-fixing commit whose commit message contains an ID of a validated bug report. With such input, BugBuilder extracts the concise patch from the commit as follows:

- First, it identifies refactorings that have been applied to V_{n-1}. The identification is conducted by *RefactoringMiner* [32], a state-of-the-art approach to mining software refactorings by comparing two consecutive versions of the same application. The first step results in a list of refactorings, noted as R.
- Second, if R is not empty, BugBuilder applies all of the discovered refactorings to V_{n-1} via Eclipse refactoring APIs. The applications result in a new version V'_{n-1} (called *refactoring-included version*) that is different from both V_{n-1} and V_n.
- Third, BugBuilder distinguishes the difference between the *refactoring-included version* (V'_{n-1}) and the fixed version (V_n). The difference is represented as a sequence of changes, noted as $Chgs$.
- Fourth, BugBuilder enumerates all possible subsequences of $Chgs$, and validates whether the subsequences represent candidate patches. For a subsequence $schg \subseteq Chgs$, BugBuilder applies all changes in $schg$ to V'_{n-1}, resulting in a new version V''_{n-1}. The subsequence $schg$ represents a candidate patch if and only if V''_{n-1} passes all test cases in T_{n-1} and passes some test cases in T_n that fail on V_{n-1}.
- Finally, BugBuilder selects the most likely patch from the validated candidate patches. If only a single candidate patch is generated by BugBuilder, it is deemed as the concise and complete patch for the associated bug report. However, when BugBuilder generates multiple candidate patches, it selects the most likely one from the candidate patches according to a series of heuristics. If none of the heuristics works, no patch would be recommended for the given commit.

It should be noted that the patches generated by BugBuilder should be applied to V'_{n-1} (refactoring-included version). Applying such patches to the original buggy version (V_{n-1}) may result in compilation errors and may not fix the bugs. Although we may revise BugBuilder to generate patches that could be directly applied to the original buggy version, we decide to follow the widely used Defects4J: In Defects4J, patches are also intended to be applied to V'_{n-1} (called V_{bug} in Defects4J [16]). Following the same pattern may facilitate the users of Defects4J to make advantage of BugBuilder.

4.2 Mining and Reapplying Refactorings

Software refactoring is to restructure software applications without changing their external behaviors. It is commonly used to improve software quality, especially the readability and maintainability [45], [46]. Notably, software refactoring is frequently conducted with other development activities, e.g., bug fixing. Consequently, to extract concise bug-fixing patches, we should identify and exclude refactorings within the commits. Our strategy is to discover refactorings involved in bug-fixing commits (by data mining) and remove such refactorings (by reapplication of refactorings) before patches are extracted.

Automated identification of software refactorings from version control systems has been extensively studied, and some automatic, highly accurate approaches [47], [48], [49], [50], [51] have been proposed. The discovered refactorings have been exploited to facilitate the evaluation of automatic refactoring recommendation algorithms [52], empirical studies on code evolution [53], and library API migration [54]. However, to the best of our knowledge, such approaches have not yet been applied to automatic extraction of patches as what we do in this paper.

To make the paper self-contained, we present here a brief introduction to automatic refactoring detection, and more details are referred to related work [32], [55]. An automatic refactoring detection algorithm takes as input two consecutive versions (noted as V_{n-1} and V_n, respectively) of the same application. It first matches elements (e.g., classes, methods, and variables) across versions. With the matched elements, it identifies which elements in the former version (i,e., V_{n-1}) have been removed, which elements in the latter version (i.e., V_n) have been added, and which elements are kept untouched. It then infers refactorings based on the removed, added, and untouched elements according to a list of pre-defined heuristic rules. For example, if a method m in class C_1 (of version V_{n-1}) matches a method m' in class C_2 (of version V_n) and C_1 does not match C_2, the algorithm recognizes the changes as a *move method* refactoring that moves method m from class C_1 to class C_2. The performance

of the algorithms depends on the accuracy of the employed element matching algorithm and the quality of the heuristic rules. In this paper, we leverage *RefactoringMiner* [55] to discover refactorings in bug-fixing code commits because existing study suggests that it is highly accurate (precision=99.6% and recall=94%) [32]. A significant advantage of RefactoringMiner is that it leverages an AST-based highly accurate matching algorithm, and this algorithm does not require any user-defined thresholds [55].

BugBuilder excludes the discovered refactorings by reapplying such refactorings to the faulty version V_{n-1}, and employs the resulting version (called V'_{n-1}) instead of the original faulty version V_{n-1} to generate the patch. The rationale is that we can divide the revision (bug-fixing commit) into two steps: (i) applying refactorings on V_{n-1}, which results in an intermediate version V'_{n-1}; and (ii) fixing bugs and implementing new features (if there is any) on V'_{n-1}. For convenience, we call the intermediate version V'_{n-1} *refactoring-included version*. Notably, reapplication of the discovered refactorings is accomplished by calling Eclipse refactoring APIs [56]. Such APIs are widely used and well-established. For example, if a bug-fixing commit contains a *rename* refactoring that changes the name of method m from *oldMethodName* to *newMethodName*, we can reapply the refactoring by calling the method *Rename* in Listing 2.

```
1  Rename(m,newMethodName);
2    ...
3   public void Rename (IJavaElement element, String
        newName){
4   ...
5    RenameJavaElementDescriptor rnDescriptor=
6    createRenameDescriptor(element, newName);
7    RenameSupport rnSupport=RenameSupport
8    .create(rnDescriptor);
9   Shell shell=PlatformUI.getWorkbench().
        getActiveWorkbenchWindow().getShell();
10   rnSupport.perform(shell, PlatformUI.getWorkbench()
11   .getActiveWorkbenchWindow());
```

Listing 2. Reapplication of Rename Refactoring

Notably, the method *Rename* in Listing 2 depends on a sequence of Eclipse refactoring APIs. Invocations of such APIs are highlighted with green background in the code snippet. We have to customize the code snippet for different categories of refactorings to forward refactoring information from *RefactoringMiner* to refactoring APIs because the required refactoring information and refactoring APIs vary significantly among different categories of refactorings. Currently, we have customized the code snippet for eight most common refactorings, including *rename classes*, *rename methods*, *rename variables*, *rename fields*, *rename parameters*, *rename packages*, *extract methods*, and *extract variables*. According to our analysis, such refactorings account for the majority (72%) of the refactorings in the bug-fixing commits in Defects4J. The empirical study conduced by Murphy et al. [57] also suggests that "rename" is by far the most popular refactoring common and "extract" is on the third place. The most challenging part in implementing the *replication of refactorings* is to figure out how Eclipse implements the refactorings, and how to invoke the related APIs to automate the refactorings without any human intervention. Notably, refactoring APIs in Eclipse are complex and hard to understand. There are a large number of refactoring-related classes in JDT. One of the most simple refactoring *extract variable* involves more than ten classes distributed in different packages. Heavy coupling with UI elements is also preventing readers to find out the clear map for API invocation without activating UI elements.

4.3 Generating Potential Patches

Given the *refactoring-included version* V'_{n-1} and its associated bug-fixing version V_n, BugBuilder should generate all possible bug-fixing patches. To this end, it computes the difference between V'_{n-1} and V_n (excluding their differences in test cases). The difference is represented as a sequence of token-level changes (e.g., removing or inserting a token), noted as $Chgs =< chg_1, chg_2, \ldots, chg_k >$. Each of the token-level changes is composed of three parts: position, token, and edition type where edition type is either "*remove*" or "*insert*".

BugBuilder generates potential bug-fixing patches by enumerating all subsequences of $Chgs$. Each subsequence $schg \subseteq Chgs$ represents a potential bug-fixing patch that makes all of the token-level changes in $schg$ on V'_{n-1}, and ignores other changes in $Chgs$. To reduce the number of potential patches, we also introduce coarse-grained changes: line-level changes. If a whole line of source code has been removed from V'_{n-1}, we represent it as a line-level change instead of a sequence of token-level changes. Insertion of a new line of source code is handled in the same way as a line-level change. Consequently, a potential patch is finally represented as a sequence of token-level and/or line-level changes.

4.4 Validating Potential Patches

BugBuilder validates a potential patch pt as follows. First, it applies this potential patch to the refactoring-included version V'_{n-1}, resulting in a new version V''_{n-1}. If the resulting version V''_{n-1} could not compile successfully, the potential patch pt is discarded as an illegal patch and its validation terminates.

If V''_{n-1} passes the preceding validation (i.e., compilation), BugBuilder further validates it with test cases associated with the faulty version (noted as T_{n-1}) and test cases associated with the fixed version (noted as T_n) as follows:

- The potential patch pt is not a valid bug-fixing patch and its validation terminates if any test case in T_{n-1} fails on V''_{n-1};
- BugBuilder collects all test cases in T_n that fail on V_{n-1}, and notes such test cases as *potential triggering test cases* that may expose the associated bug;
- The potential patch pt is deemed invalid if V''_{n-1} fails to pass any potential triggering test cases. Otherwise, it is taken as a candidate patch.

4.5 Selecting from Candidate Patches

If BugBuilder generates exactly a single candidate patch from a bug-fixing code commit, it is deemed a concise patch for the reported software bug associated with the bug-fixing commit. Consequently, the approach recommends the candidate patch for the associated bug report.

However, it is challenging, if not impossible, to select automatically the correct one from a number of candidate patches because none of them could be filtered out by the associated test cases: All of them can pass the associated test cases, including all of the potential triggering test cases. To this end, in this paper, we only make recommendations for some special cases if there are multiple candidates.

First, it employs heuristic rule H_1 in Section 4.5.1 to exclude such candidate patches that contain only part of repeated changes. Second, if multiple candidate patches survive the heuristic rule H_1, BugBuilder employs the second heuristic rule H_2 in Section 4.5.2 to exclude candidate patches that contain optional changes. If exactly a single candidate patch survives both H_1 and H_2, it is recommended by BugBuilder as the correct concise and complete patch. Details are presented in the following subsections.

4.5.1 Repeated Changes

Heuristics 1 (H_1). Suppose that a candidate patch $cand_i$ contains a sequence of changes $chgs =< chg_1, chg_2, \ldots, chg_k >$, and the whole sequence of changes have been repeated (for n times) in other places by the enclosing bug-fixing code commit. If adding all such repeated changes to candidate patch $cand_i$ results in another candidate patch $cand_j$, it is likely that candidate patch $cand_i$ is incomplete and thus invalid.

The rationale for the heuristic is that developers may fix the reported bug with a sequence of changes to pass the triggering test case, and fix the same bug in other places not covered by the triggering test case. In this case, a validated patch should contain all of the bug-fixing changes to make the patch *complete*.

```
1  diff --git a/src/main/java/org/apache/commons/compress/
       archivers/tar/TarArchiveInputStream.java b/src/main/
       java/org/apache/commons/compress/archivers/tar/
       TarArchiveInputStream.java
2  index c557007..41acf2a 100644
3  --- a/src/main/java/org/apache/commons/compress/
       archivers/tar/TarArchiveInputStream.java
4  +++ b/src/main/java/org/apache/commons/compress/
       archivers/tar/TarArchiveInputStream.java
5  @@ -498,11 +498,11 @@ public class TarArchiveInputStream
        extends ArchiveInputStream {
6  } else if ("linkpath".equals(key)){
7  currEntry.setLinkName(val);
8  } else if ("gid".equals(key)){
9  -          currEntry.setGroupId(Integer.parseInt(val));
10 +          currEntry.setGroupId(Long.parseLong(val));
11 } else if ("gname".equals(key)){
12 currEntry.setGroupName(val);
13 } else if ("uid".equals(key)){
14 -          currEntry.setUserId(Integer.parseInt(val));
15 +          currEntry.setUserId(Long.parseLong(val));
16 } else if ("uname".equals(key)){
17 currEntry.setUserName(val);
18 } else if ("size".equals(key)){
```

Listing 3. Repeated Changes

A typical example is presented in Listing 3. The changes on Lines 9-10 alone, i.e., changing '`curEntry.setGroupId(Integer.parseLong(val))`' into '`curEntry.setGroupId(Long.parseLong(val))`' can pass the triggering test cases associated with the commit. However, the same bug also appears on Line 14 that is not covered by the triggering test cases. Consequently, the developer repeated exactly the same bug-fixing actions on Line 14 to fix the bug completely. The changes on Line 9 and Line 14 together, as suggested by Defects4J, compose the concise and complete bug-fixing patch. Notably, the incomplete candidate patch (i.e., changes on Line 9 alone) can pass the associated triggering test cases because the triggering test cases are insufficient: additional test cases is required to reveal uncovered buggy statements (i.e., Line 14). To facilitate the comprehension of bug-fixing commits, we strongly encourage developers to create triggering test cases that are able to distinguish incomplete candidate patches from complete ones.

BugBuilder employs the heuristics H_1 to exclude invalid candidate patches that contain only part of the repeated changes. If it excludes all but one candidate patch, BugBuilder recommends the remaining one as the validated patch for the associated bug report.

4.5.2 Optional Changes

Heuristics 2 (H_2). Suppose that candidate patch $cand_i$ is a superset of another candidate patch $cand_j$, it is likely that $cand_i$ contains optional changes (i.e., this patch is not concise) if all of the following preconditions hold:

1) The difference is not empty, i.e., $diff_{i,j} = cand_i - cand_j \neq \emptyset$,
2) $diff_{i,j}$ is not simply repeating changes in $cand_j$, and
3) All source code (in the buggy version) modified or deleted by $diff_{i,j}$ and all source code (in the fixed version) inserted by $diff_{i,j}$ are covered by the associated test cases.

```
1  diff --git a/src/java/org/apache/commons/cli/PosixParser
       .java b/src/java/org/apache/commons/cli/PosixParser.
       java
2  index 12d3146..f3c9d0d 100644
3  --- a/src/java/org/apache/commons/cli/PosixParser.java
4  +++ b/src/java/org/apache/commons/cli/PosixParser.java
5  @@ -99,7 +99,7 @@ public class PosixParser extends
       Parser {
6    // an iterator for the command line tokens
7      Iterator iter = Arrays.asList(arguments).iterator();
8  -   String token = null;
9  +   String token;
10     // process each command line token
11     while (iter.hasNext())
12 @@ -302,7 +302,8 @@ public class PosixParser extends
       Parser {
13     }
14     else
15        {
16 -        tokens.add("-" + ch);
17 +        tokens.add(token);
18 +         break;
19        }
20     }
21 }
```

Listing 4. Optional Changes

An illustrating example is presented in Listing 4. This example comes from open-source application Apache Commons CLI [58]. This commit is to fix bug report #CLI-51 [59]: parameter value "-something" misinterpreted as a parameter. This commit makes two changes: replacing Line 16 with Lines 17-18, and replacing Line 8 with Line 9. The first change (on Line 16-18) is directly related to the associated bug, and fixes it completely.

The other change (on Line 8-9), however, is not directly related to the bug fixing. In the original version, the string variable `token` is explicitly initialized with constant "null". The bug-fixing commit deletes the initialization (as shown

on Line 9). However, the deletion is optional and is not indispensable for the bug fixing. In fact, this change does not change any functionality of the software because string variables would be initialized with "null" by default. Consequently, this change should be taken as software refactoring.

Notably, RefactoringMiner [55] fails to recognize this refactoring because it is not a typical or popular software refactoring, no special heuristics rules have been proposed to recognize such rare refactorings.

The rationale of heuristics H_2 is that some optional changes within bug-fixing commits (e.g., atypical software refactorings) should be excluded from concise bug-fixing patches. However, the employed refactoring-mining tool [55] cannot detect such changes and thus we cannot exclude them by detecting-and-reapplying refactorings (as we do in Section 4.2).

Notably, the preconditions (especially the third one) for the heuristics H_2 are indispensable. If the difference $diff_{i,j}$ is not covered by test cases, it is risky to tell that such changes are optional. We employ the bug-fixing commit in Listing 5 to illustrate the necessity of the precondition. This commit comes from open-source application Closure Compiler to fix bug report #issues-1144 [60]. In this example, BugBuilder generates two candidate patches. The first one (called `theBigger`) is composed of the changes on Lines 8, 16, 24, 32, and 33. The second one (called `theSmaller`) is composed of the changes on Lines 8, 24, 32, and 33. The only difference is that `theBigger` contains changes on Line 16 whereas `theSmaller` does not. Excluding the third condition of heuristics H_2, i.e., (does not request the differences between the patches to be covered by test cases), the proposed approach would exclude `theBigger` by taking changes on Line 16 as optional changes by mistake. However, changes on Line 16 are indispensable for the bug fixing. Without such changes, the updated `IF` statement on Line 33 may not work correctly when the given document contains more than one scope. The smaller candidate patch `theSmaller` passes the validation of the proposed approach (as introduced in Section 4.4) because the associated test cases are insufficient, failing to cover the changes on Line 16. By enabling the preconditions of H_2, however, the proposed approach avoids such kinds of mistakes.

BugBuilder employs the heuristics H_2 to exclude inconcise candidate patches. If it excludes all but one candidate patch, BugBuilder recommends the remaining one as the validated patch for the associated bug report. If more than one candidate patch survives, BugBuilder does not recommend any patch for the given bug-fixing commit.

```
1  diff --git a/src/com/google/javascript/jscomp/
       ScopedAliases.java b/src/com/google/javascript/
       jscomp/ScopedAliases.java
2  index 8235052..b2a1690 100644
3  --- a/src/com/google/javascript/jscomp/ScopedAliases.
       java
4  +++ b/src/com/google/javascript/jscomp/ScopedAliases.
       java
5  @@ -255,6 +255,9 @@ class ScopedAliases implements
       HotSwapCompilerPass {
6  private final Map<String, Var> aliases = Maps.newHashMap
       ();
7
8  +   private final Set<Node> injectedDecls =
       Sets.newHashSet();
9
10 // Suppose you create an alias.
11 // var x = goog.x;
12 @@ -311,6 +314,7 @@ class ScopedAliases implements
       HotSwapCompilerPass {
13
14 if (t.getScopeDepth() == 2) {
15 renameNamespaceShadows(t);
16 +      injectedDecls.clear();
17 aliases.clear();
18 forbiddenLocals.clear();
19 transformation = null;
20 @@ -427,6 +431,7 @@ class ScopedAliases implements
       HotSwapCompilerPass {
21 } else {
22       grandparent.addChildBefore(newDecl, varNode);
23 }
24 +         injectedDecls.add(newDecl.getFirstChild());
25 }
26
27 @@ -573,8 +578,10 @@ class ScopedAliases implements
       HotSwapCompilerPass {
28 aliasUsages.add(new AliasedNode(aliasVar, n));
29 }
30
31 JSDocInfo info = n.getJSDocInfo();
32 -      if (info != null) {
33 +      if (info != null && !injectedDecls.contains(n)) {
34       for (Node node : info.getTypeNodes()) {
35             fixTypeNode(node);
36       }
```

Listing 5. Bug-Fixing Changes Not Covered by Test Cases

5 Evaluation

In this section, we evaluate the proposed approach (BugBuilder) on bug-fixing commits collected by well-known Defects4J.

5.1 Research Questions

The evaluation aims to investigate the following research questions:

- **RQ1**: How often do bug-fixing commits contain bug-irrelevant changes and what percentage of the changes in bug-fixing commits are bug-irreverent?
- **RQ2**: How accurate is BugBuilder in extracting complete and concise bug-fixing patches from bug-fixing commits? How often could real-world patches in version control systems be extracted accurately and automatically by BugBuilder?
- **RQ3**: To what extent does the refactoring detection and replication affect the precision and recall of BugBuilder?
- **RQ4**: To what extent do the candidate selection strategies influence the performance of BugBuilder?
- **RQ5**: How long does it take BugBuilder to extract a patch from a bug-fixing commit, and how is BugBuilder's performance influenced by the size of commits?

5.2 Dataset

In this section, we evaluate BugBuilder with the raw data in Defects4J that contains 835 real-world bugs collected from real-world applications. For each bug, Defects4J provides the bug-fixing code commit, the versions immediately following/preceding the bug-fixing commit (called V_n and V_{n-1}, respectively), and the manually confirmed patch for

the bug. V_n and V_{n-1} were taken from version control history by Defects4J without any modification. V_{n-1} is different from the faulty version (V_{bug}) provided by Defects4J. Applying bug-irrelevant change in the commit to V_{n-1} results in V_{bug} whereas applying the bug-fixing patch to V_{bug} should result in the fixed version V_n.

Only V_n and V_{n-1} were leveraged as the input of BugBuilder, whereas the manually constructed patches provided by Defects4J were leveraged only to assess the performance of BugBuilder, i.e., whether the automatically generated patches are identical to the manually constructed ones. BugBuilder does not depend on the reference patches (in Defects4J) to generate bug-fixing patches.

In total, 809 bug-fixing commits from Defects4J were leveraged for the evaluation. Although Defect4J contains 835 bug-fixing commits from 17 projects, we failed to retrieve the V_{n-1} version for project *Chart* because the version IDs for this project are invalid. Consequently, this project was excluded from our evaluation. We did not exploit other bug-patch datasets, like iBUGS and ManyBugs, for the evaluation because they do not exclude bug-irrelevant changes from the final bug-fixing patches.

5.3 Experiment Design

5.3.1 RQ1: Popularity of Bug-irrelevant Changes Within Bug-fixing Commits:

If bug-fixing commits often contain bug-irrelevant changes, it could be risky to take the whole bug-fixing commits as bug-fixing patches. This assumption serves as the basis of the proposed approach. To validate the assumption, we compared the manually constructed bug-fixing patches in Defects4J against their associated bug-fixing commits. The comparison was conducted in two steps. First, we investigated how often the bug-fixing patches are identical to their associated code commits:

$$P_{\text{same}} = \frac{\textit{number of commits identical to associated patches}}{\text{number of bug-fixing commits}} \tag{1}$$

Assuming that patches in Defects4J are complete and concise, bug-fixing commits that are not identical to the associated patches must contain bug-irrelevant changes. Consequently, $P_{\text{diff}} = 1 - P_{\text{same}}$ is the percentage of bug-fixing commits that contain bug-irrelevant changes.

Second, we investigated what percentage of changes in bug-fixing commits are bug-fixing changes and what percentage of changes are bug-irrelevant changes. Because patches in Defects4J have been manually constructed to exclude bug-irrelevant changes [16], we took the size of the patches in Defects4J as the size of bug-fixing changes in the associated bug-fixing commits.

5.3.2 RQ2: Precision and Recall of BugBuilder

To investigate the precision and recall of BugBuilder, we evaluated it on each of the bug-fixing commits in Defects4J as follow:

- First, we retrieved its associated V_n and V_{n-1} versions as well as the manually constructed patch pt_{4j} associated with the bug-fixing commit;
- Second, we leveraged BugBuilder to generate patches, taking V_{n-1} and V_n as input;
- Third, if BugBuilder resulted in a patch pt, we compared it against the manually constructed patch pt_{4j} to reveal whether the automatically generated patch was identical to the manually constructed one. In case they were identical, we called the generated patch a *matched patch*. Notably, the comparison between generated patches and the ground truth was a pure textual comparison of the patches, and it was fully automatic.

An automatically generated patch was taken as a complete and concise patch if and only if it was a matched patch, i.e., it was identical to the manually constructed patch (provided by Defects4J) associated with the same bug-fixing commit.

Based on the preceding process, we computed the precision and recall of BugBuilder as follows:

$$Precision = \frac{\textit{number of matched patches}}{\text{number of generated patches}} \tag{2}$$

$$Recall = \frac{\textit{number of matched patches}}{\text{number of patches in Defects4J}} \tag{3}$$

5.3.3 RQ3: Impact of Refactoring Detection and Reapplication

BugBuilder excludes refactorings from generated patch by discovering refactorings contained in the bug-fixing commit and reapplying the discovered refactorings to the associated faulty version (see Section 4.2 for details). To investigate to what extent the leveraged refactoring detection and reapplication may affect the precision and recall of BugBuilder, we disabled refactoring detection and reapplication, and repeated the evaluation (as specified in Section 5.4.3).

5.3.4 RQ4: Effect of Patch Selection

As specified in Section 4.5, BugBuilder employs a sequence of heuristics to select the correct (both concise and complete) bug-fixing patch when BugBuilder generates more than one candidate patch for a single bug-fixing commit. To investigate how accurate the employed heuristics are and how they affect the performance (both precision and recall) of BugBuilder, we computed how often the heuristics were employed and how often the selection was correct/incorrect. We also disabled the selection (i.e., generating a bug-fixing patch for a given commit only if there was exactly a single candidate patch for the commit), to quantitatively assess the effect of the heuristics.

5.3.5 RQ5: Scalability of BugBuilder

The performance of BugBuilder is important because we should apply it to a large number of bug-fixing commits to construct a large-scale and high-quality bug repository. To investigate the performance and the scalability of BugBuilder, we depicted the quantitative relation between the run time of BugBuilder and the size of involved commits.

5.4 Results and Analysis

5.4.1 RQ1: Bug-fixing Commits Often Contain Bug-irrelevant Changes

For each commit in Defects4J, we counted the size of the commit (in lines) and the size of the associated concise bug-fixing patch. The latter represents the size of bug-fixing changes whereas the former represents the size of

whole changes, including both bug-fixing changes and bug-irrelevant changes. Evaluation results are presented on Table 1. From this table, we make the following observations:

- Bug-irrelevant changes are common in bug-fixing commits. On average, bug-fixing changes account for only 63% of the changes made in bug-fixing commits. In other words, 37%=1-63% of the changes in bug-fixing commits are bug-irrelevant. All such bug-irrelevant changes should be excluded from bug-fixing patches. Consequently, taking the whole bug-fixing commits as bug-fixing patches would result in unconcise patches. Such unconcise patches, if employed by the evaluation of bug-related approaches (e.g., fault localization and program repair), could be misleading.
- The ratio of bug-fixing changes to all changes in bug-fixing commits varies significantly from project to project. As suggested by the last column of Table 1, the ratio varies from 96% (on project Jackson Dataformat XML) to 47% (on project Commons CSV). One possible reason for the variation is that different projects often pose different guidelines on how patches should be committed.

To investigate how often bug-fixing commits contain bug-irrelevant changes, we computed P_{diff} in Section 5.3.1. Our evaluation results suggest that 379 out of the 809 bug-fixing commits contain bug-irrelevant changes, resulting in a $P_{\text{diff}} = 47\% = 379/809$. The results suggest that simply taking the whole bug-fixing commits as bug-fixing patches may frequently result in unconcise patches.

> We conclude based on the preceding analysis that bug-fixing commits often contain a large percentage of bug-irrelevant changes. Consequently, we should exclude such bug-irrelevant changes from bug-fixing patches to guarantee the quality of bug-fixing patches.

TABLE 1
Bug-fixing Changes within Bug-fixing Commits

Project	Size of Commits (N_1)	Size of Bug-fixing Changes (N_2)	N_2/N_1
Jackson Dataformat XML	119	114	96%
Joda Time	264	242	92%
Commons Collections	38	29	76%
Commons Lang	689	516	75%
Commons JXPath	582	430	74%
Jackson Databind	2,104	1,508	72%
Gson	239	168	70%
Commons Codec	275	193	70%
Commons CLI	473	325	69%
Jsoup	1,163	777	67%
Jackson Core	485	307	63%
Commons Compress	602	372	62%
Commons Math	1,246	763	61%
Mockito	503	277	55%
Closure Compiler	4,108	2,111	51%
Commons CSV	119	56	47%
TOTAL	13,009	8,188	63%

5.4.2 RQ2: BugBuilder Is Accurate

To answer RQ2, we applied BugBuilder to each of the bug-fixing code commits in Defects4J and compared its generated patches against the manually constructed patches in Defects4J. If the generated patch is identical to the corresponding patch provided by Defects4J, we call it a *matched patch*.

Table 2 presents the evaluation results. The first two columns of the table specify the project names and the number of bug-fixing commits in the projects. The third column presents the number of patches generated by BugBuilder. The fourth column presents the number of the matched patches, i.e., generated patches that are identical to the manually constructed ones in Defects4J. The precision and recall of BugBuilder are presented in the last two columns. From Table 2, we make the following observations:

- BugBuilder succeeded frequently. In total, it generated 350 bug-fixing patches from 809 bug-fixing commits. Among them, 334 are identical to manually constructed patches in Defects4J, which results in a recall 41%=334/809.
- BugBuilder was highly accurate. Among the 350 automatically generated patches, 95%=334/350 are identical to manually constructed ones in Defects4J. Notably, on 6 out of the 16 projects, BugBuilder achieves 100% precision, i.e., all patches generated from such projects are both complete and concise. Such a high precision (95%) guarantees that the resulting bug repositories built with BugBuilder could be highly reliable.

From the table, we also observe that 16 (=350-334) out of the 350 patches generated by BugBuilder are different from their corresponding patches in Defects4J. We call them *mismatched patches* because they do not match the benchmark (manually constructed patches provided by Defects4J). To investigate why they are different from the benchmark, we manually analyzed such mismatched patches, referring to the corresponding patches in Defects4J, associated bug reports, and the associated code commits. Based on the manual analysis, we observed that all of the 16 mismatched patches are supersets of their corresponding reference patches in Defects4J. Consequently, the reason for the mismatch should be either (or both) of the following:

1) The patches generated by BugBuilder are complete but not concise. In other words, they include some bug-irrelevant changes as well as all bug-fixing changes;
2) The manually constructed patches in Defects4J are incomplete, i.e., they miss some bug-fixing changes that appear in the patches generated by BugBuilder.

It is surprising that the patches automatically generated by BugBuilder are often even better than manually constructed patches in Defects4J. Twelve out of the 16 mismatched patches were manually confirmed as correct

TABLE 2
Precision and Recall of BugBuilder

Project	Bug-fixing Commits	Generated Patches	Matched Patches	Precision	Recall
Commons CLI	39	19	19	100%	49%
Closure Compiler	174	70	68	97%	39%
Commons Codec	18	12	10	83%	56%
Commons Collections	4	1	1	100%	25%
Commons Compress	47	25	24	96%	51%
Commons CSV	16	10	9	90%	56%
Gson	18	9	8	89%	44%
Jackson Core	26	10	10	100%	38%
Jackson Databind	112	36	35	97%	31%
Jackson Dataformat XML	6	2	2	100%	33%
Jsoup	93	42	38	90%	41%
Commons JXPath	22	7	6	86%	27%
Commons Lang	64	31	29	94%	45%
Commons Math	106	46	45	98%	42%
Mockito	38	19	19	100%	50%
Joda Time	26	11	11	100%	42%
TOTAL	809	350	334	95%	41%

(i.e., both complete and concise) whereas their corresponding patches in Defects4J missed some bug-fixing changes (i.e., they were incomplete). Counting in such 12 complete and concise patches, the precision of BugBuilder increases to 99%=(334+12)/350, and its recall increases to 43%=(334+12)/809. We will take these updated precision and recall as the baseline in the following evaluation (e.g., in Section 5.4.3 and Section 5.4.4). We observe that its precision (99%) is even higher than the experts who manually constructed the Defects4J patches: On the 350 commits where BugBuilder generates patches, BugBuilder generates only 4 unconcise patches whereas the experts resulted in 12 incomplete patches.

The first and the foremost reason for incomplete patches in Defects4J is that fixing a bug may require duplicate (or highly similar) changes in multiple places (e.g., multiple documents) but human experts missed some places. Listing 6 presents a typical example. This is a bug-fixing commit from Apache Commons Codec [61], and the associated bug report is publicly available online [62]. As the bug report explains, the return statements *return new String(bytes, Charsets.xxx)* in a sequence of *newStringxxx* methods (Lines 7, 13, 19, 25, and 31) could not handle null input, and thus they should be replaced with *return newString(bytes, Charsets.xxx)*. However, the patch in Defects4J [63] contains only the changes in one of the methods (i.e., the first method in Listing 6), and thus it is incomplete. In contrast, our approach successfully generates the complete patch containing all of the similar changes in all *newStringxxx* methods.

Another reason for the incomplete patches in Defects4J is that they ignore the required changes in method declarations and/or variable declarations. Listing 7 presents a typical example. The associated commit comes from Apache Commons CSV [64], and the associated bug report is publicly available online [65]. The manually constructed patch provided by Defects4J is publicly available at [66]. As the bug report explains, *CSVFormat* with header does not work with *CSVPrinter*. To fix the bug, the developers added the whole *if* statement (Lines 25-27) to print the header if it is not null.

```
1  diff --git a/src/main/java/org/apache/commons/codec/
       binary/StringUtils.java b/src/main/java/org/apache/
       commons/codec/binary/StringUtils.java
2  index 84a2a72..7bb15e3 100644
3  --- a/src/main/java/org/apache/commons/codec/binary/
       StringUtils.java
4  +++ b/src/main/java/org/apache/commons/codec/binary/
       StringUtils.java
5  @@ -336,7 +336,7 @@ public class StringUtils {
6  public static String newStringIso8859_1(final byte[]
       bytes) {
7  -       return new String(bytes, Charsets.ISO_8859_1);
8  +       return newString(bytes, Charsets.ISO_8859_1);
9  }
10
11 @@ -352,7 +352,7 @@ public class StringUtils {
12 public static String newStringUsAscii(final byte[] bytes
       ) {
13 -       return new String(bytes, Charsets.US_ASCII);
14 +       return newString(bytes, Charsets.US_ASCII);
15 }
16
17 @@ -368,7 +368,7 @@ public class StringUtils {
18 public static String newStringUtf16(final byte[] bytes)
       {
19 -       return new String(bytes, Charsets.UTF_16);
20 +       return newString(bytes, Charsets.UTF_16);
21 }
22
23 @@ -384,7 +384,7 @@ public class StringUtils {
24 public static String newStringUtf16Be(final byte[] bytes
       ) {
25 -       return new String(bytes, Charsets.UTF_16BE);
26 +       return newString(bytes, Charsets.UTF_16BE);
27 }
28
29 @@ -400,7 +400,7 @@ public class StringUtils {
30 public static String newStringUtf16Le(final byte[] bytes
       ) {
31 -       return new String(bytes, Charsets.UTF_16LE);
32 +       return newString(bytes, Charsets.UTF_16LE);
33 }
```

Listing 6. Duplicate Changes in Multiple Places Ignored by Human Experts

Notably, the method declaration of *printRecord* explicitly

specifies that it has the potential to throw *IOException*. Consequently, inserting an invocation of this method (Line 16) forces the enclosing method (and its caller, method *print* on Line 6) to explicitly specify the *IOException* in their method declarations (Line 7 and Line 17). Otherwise, the revision would result in compiler errors. However, the patch in Defects4J ignores such changes in method declarations, and thus it is incomplete. In contrast, our approach generated the complete patch including the changes in method declarations.

```
1  diff --git a/src/main/java/org/apache/commons/csv/
       CSVFormat.java b/src/main/java/org/apache/commons/
       csv/CSVFormat.java
2  index a6fde0a..5963edb 100644
3  --- a/src/main/java/org/apache/commons/csv/CSVFormat.
       java
4  +++ b/src/main/java/org/apache/commons/csv/CSVFormat.
       java
5  @@ -605,8 +605,10 @@ public final class CSVFormat
       implements Serializable {
6  - public CSVPrinter print(final Appendable out) {
7  + public CSVPrinter print(final Appendable out)
       throws IOException {
8  return new CSVPrinter(out, this);
9  }
10
11 diff --git a/src/main/java/org/apache/commons/csv/
       CSVPrinter.java b/src/main/java/org/apache/commons/
       csv/CSVPrinter.java
12 index d048f89..d2968b5 100644
13 --- a/src/main/java/org/apache/commons/csv/CSVPrinter.
       java
14 +++ b/src/main/java/org/apache/commons/csv/CSVPrinter.
       java
15 @@ -45,24 +45,31 @@ public final class CSVPrinter
       implements Flushable, Closeable {
16 - public CSVPrinter(final Appendable out,
       final CSVFormat format) {
17 + public CSVPrinter(final Appendable out,
       final CSVFormat format) throws IOException {
18 Assertions.notNull(out, "out");
19 Assertions.notNull(format, "format");
20
21 this.out = out;
22 this.format = format;
23 this.format.validate();
24
25 +  if (format.getHeader() != null) {
26 +    this.printRecord((Object[]) format.getHeader());
27 +   }
28 }
```

Listing 7. Throw Statements Ignored by Human Experts

To investigate why BugBuilder generated four unconcise patches, we manually analyzed such bug-fixing commits, the associated bug reports, and the generated patches. Our analysis results suggest that all of the 4 unconcise patches were created because the leveraged refactoring-mining tool missed some refactorings in the involved code commits. As a result, the uncovered refactorings were taken as a part of the bug-fixing patches generated by BugBuilder, which in turn resulted in unconcise patches.

Listing 8 presents a typical example. This example comes from Google Gson [67]. The associated bug report is publicly available online [68]. The bug report complains that the method (more specifically, the *return* statement on Line 19) would result in null pointer exceptions when *typeAdapter* is null. To fix the bug, developers inserted an *if* statement (Line 20) to validate that *typeAdapter* is not null. The patch provided by Defects4J [69] is composed of two changes only: Line 20 and Line 22. Other changes are ignored. In contrast, our approach takes all of the changes in the listing as bug-fixing changes. One possible rationale for Defects4J to exclude other changes from the patch is that they could be taken as refactorings: decomposing statement *return typeAdapter.nullSafe();* (Line 19) into two statements *typeAdapter=typeAdapter.nullSafe();* on Line 21 and *return typeAdapter* on Line 23. Because variable *typeAdapter* would not be used anywhere after the *return* statement (Line 23), it could be used as a temporary variable safely. As a result of the usage, the keyword *final* (Line 9) should be removed from the declaration of variable *typeAdapter* because it is assigned/changed on Line 21 as a temporary variable. We will not argue that such changes should not be taken as refactorings. However, it is a rather complex and unusual *extract variable* refactoring (if it is) because an *extract variable* refactoring usually defines a new variable instead of employing an existing variable temporarily. Such unusual refactoring is beyond the capability of the state-of-the-art refactoring mining tools. Consequently, BugBuilder failed to recognize this refactoring and thus took all of the changes as bug-fixing changes.

```
1  diff --git a/gson/src/main/java/com/google/gson/internal
       /bind/JsonAdapterAnnotationTypeAdapterFactory.java b
       /gson/src/main/java/com/google/gson/internal/bind/
       JsonAdapterAnnotationTypeAdapterFactory.java
2  index 3801cfd..b52e157 100644
3  --- a/gson/src/main/java/com/google/gson/internal/bind/
       JsonAdapterAnnotationTypeAdapterFactory.java
4  +++ b/gson/src/main/java/com/google/gson/internal/bind/
       JsonAdapterAnnotationTypeAdapterFactory.java
5  @@ -51,7 +51,7 @@ public final class
       JsonAdapterAnnotationTypeAdapterFactory implements
       TypeAdapte
6  static TypeAdapter<?> getTypeAdapter(
       ConstructorConstructor constructorConstructor, Gson
       gson,
7  TypeToken<?> fieldType, JsonAdapter annotation) {
8  Class<?> value = annotation.value();
9  -   final TypeAdapter<?> typeAdapter;
10 +   TypeAdapter<?> typeAdapter;
11 if (TypeAdapter.class.isAssignableFrom(value)) {
12 Class<TypeAdapter<?>> typeAdapterClass = (Class<
       TypeAdapter<?>>) value;
13 typeAdapter = constructorConstructor.get(TypeToken.get(
       typeAdapterClass)).construct();
14 @@ -64,7 +64,9 @@ public final class
       JsonAdapterAnnotationTypeAdapterFactory implements
       TypeAdapte
15 throw new IllegalArgumentException(
16 "@JsonAdapter value must be TypeAdapter or
       TypeAdapterFactory reference.");
17 }
18
19 -   return typeAdapter.nullSafe();
20 +   if (typeAdapter != null) {
21 +     typeAdapter = typeAdapter.nullSafe();
22 +   }
23 +   return typeAdapter;
24 }
25 }
```

Listing 8. Imperfect Patch Caused by Undiscovered Refactorings

While the example in Listing 8 illustrates how unusual refactorings affect BugBuilder, the example in Listing 9 illustrates how BugBuilder is affected by unsupported refactorings. The bug-fixing commit in Listing 9 comes from Google Closure Compiler [70] and the associated bug report is available at [71]. The bug-fixing changes include the changes on the *while* condition (Lines 11-13) and the *if* statement (Lines 18-21). Such changes are included in both the automatically generated patch and the manually constructed Defects4J patch. However, other changes, i.e., moving the declaration of local variables (*parameter* and *argument*) from the interior of the *while* iteration (Lines 15

and 16) to the outside of the *while* iteration (Lines 7-8), are not taken by Defects4J as bug-fixing changes because they should be taken as refactorings: The movement would not change the functionality of the method but improves its performance by avoiding the repeating definition of the same variables. However, this kind of refactorings is not yet supported by the refactoring mining tool that is leveraged by BugBuilder. Consequently, BugBuilder failed to remove such refactorings from its generated patches. Notably, it remains controversial whether the refactorings should be excluded from the patch because without such refactorings it is impossible to use variable *parameter* in the *while* condition (as the patch does). However, in this paper, we conservatively take it as a false positive of BugBuilder to avoid controversies.

```
1  diff --git a/src/com/google/javascript/jscomp/TypeCheck.
       java b/src/com/google/javascript/jscomp/TypeCheck.
       java
2  index 8198efc..b05fbc4 100644
3  --- a/src/com/google/javascript/jscomp/TypeCheck.java
4  +++ b/src/com/google/javascript/jscomp/TypeCheck.java
5  @@ -1403,9 +1403,17 @@ public class TypeCheck implements
         NodeTraversal.Callback, CompilerPass {
6  int ordinal = 0;
7  +   Node parameter = null;
8  +   Node argument = null;
9
10    while (arguments.hasNext() &&
11 -     parameters.hasNext()) {
12 +     (parameters.hasNext()||
13 +     parameter != null && parameter.isVarArgs())) {
14
15 -     Node parameter = parameters.next();
16 -     Node argument = arguments.next();
17
18 +     if (parameters.hasNext()) {
19 +       parameter = parameters.next();
20 +     }
21 +     argument = arguments.next();
22        ordinal++;
23
24    validator.expectArgumentMatchesParameter(t, argument,
```

Listing 9. Imperfect Patch Caused by Unsupported Refactorings

In Section 4.4, we exclude such potential patches that fail all potential triggering test cases. To validate whether it is practical to exclude all potential patches that fail at least one potential triggering test case, we change the filtering condition and repeat the evaluation. Our evaluation results suggest that this new setting reduces the performance of the proposed approach: The number of generated correct patches is reduced from 346 to 335, and the precision is reduced slightly from 98.9% to 98.5%.

> Based on the preceding analysis, we conclude that BugBuilder is highly accurate with a precision of 99%. It also achieved a reasonable recall of 43%, which suggests that it can automatically and accurately extract both complete and concise bug-fixing patches from around half of the bug-fixing commits.

5.4.3 RQ3: Refactoring Detection and Reapplication Improves Recall by 10%

Refactoring detection and reapplication as introduced in Section 4.2 is an important part of the proposed approach. To investigate its importance, we disabled it and repeated the evaluation as specified in Section 5.3.2 and Section 5.4.2. Our evaluation results are presented in Fig. 2 where *default setting* means that all components (including refactoring detection and reapplication) of the proposed approach were enabled.

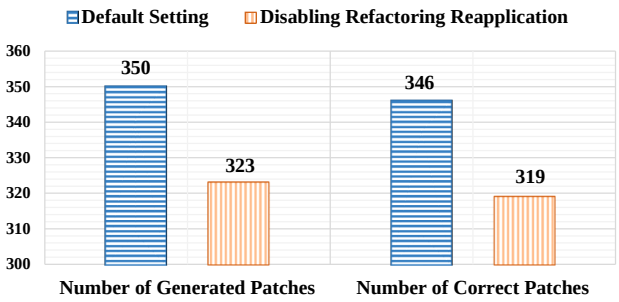


Fig. 2. Impact of Refactoring Detection and Replication

From Fig. 2, we make the following observations:

- First, refactoring detection and reapplication has a significant positive impact on the recall of BugBuilder. Enabling it substantially improves both the number of generated patches and the number of correct patches. The number of correct patches generated by the proposed approach was increased from 319 to 346. As a result, enabling the refactoring detection and reapplication improved the recall of BugBuilder from 39% to 43%, resulting in a substantial increase of 10%=(43%-39%)/39%.
- Second, refactoring detection and reapplication has little impact on the precision of BugBuilder. We notice that the precision keeps stable ($99\% \approx 346/350 \approx 319/323$) regardless of the changes in the setting. A possible reason for the stable precision is that the other components of the approach can exclude incomplete and unconcise patches. As a result, the proposed approach can reach a high precision regardless of the refactoring detection and reapplication.

Detecting and reapplying refactoring can improve the recall of the proposed approach because bug-fixing commits often contain refactorings. With the help of RefactoringMiner, we have discovered refactorings from 192 out of the involved 809 bug-fixing commits. We call such commits *refactoring-containing commits*. Eighty-three out of the 192 refactoring-containing commits contain no refactorings except for the *supported refactorings* that the current implementation of our approach can identify and reapply. From these 83 commits, BugBuilder successfully generated 27 complete and concise patches. Disabling the detection and reapplication of refactoring, however, made the proposed approach miss all such patches.

A simple and intuitive alternative approach for BugBuilder is to take the whole bug-fixing commit as a patch if and only if the bug-fixing commit does not contain any refactorings (not limited to the eight categories of refactorings supported by the current implementation of BugBuilder). We call this alternative approach `refactoring-based approach`. It would generate 617=809-192 patches from 809 bug-fixing commits

TABLE 3
Patch Selection

Metrics	Value
Number of Activations	66
Number of Selected Patches	26
Number of Correct Selection	26
Number of Incorrect Selection	0
Accuracy	100%

in Defects4J. However, up to 217 out of the 617 patches are unconcise, i.e., containing non-refactoring bug-irrelevant changes, like implementation of new features. As a result, its precision 65%=1-217/617 is significantly lower than that (99%) of BugBuilder. Such a low precision makes it unsuitable for automated construction of high-quality bug repositories.

Based on the preceding analysis, we conclude that detecting and reapplying refactorings improved recall of BugBuilder substantially by 10% whereas its precision was unchanged.

5.4.4 RQ4: Patch Selection is Highly Accurate

To answer RQ4, we identified all cases where the patch selection was activated, and compared the selected patches against the benchmark in Defects4J. Notably, the patch selection was activated if and only if BugBuidler generated more than one candidate patches for a single commit and BugBuidler exploited the heuristics (as specified in Section 4.5) to select the correct patch from them.

Our evaluation results are presented in Table 3. From this table, we observe that the patch selection had been activated on 66 bug-fixing commits, resulting in 26 selected bug-fixing patches. We also notice that all of the selected patches were identical to those in Defects4J, suggesting that the accuracy of the patch selection had a high accuracy of 100%. All such results suggest that the employed selection strategies are highly reliable.

We also disabled the patch selection and repeated the evaluation on bug-fixing commits collected by Defects4J. Notably, when patch selection was disabled, BugBuilder generated bug-fixing patch for a bug-fixing commit if and only if it generated exactly a single candidate patch from the commit. Our evaluation results are presented in Fig. 3 where *default setting* enabled the patch selection. From this figure, we make the following observations:

- First, the patch selection substantially improved the success rate (i.e., recall) of BugBuilder by 8%. Without the patch selection, BugBuilder generated 320 correct patches. Enabling the patch selection improved the number to 346. As a result, the number of the correct patches (and the recall of BugBuilder as well) was improved substantially by 8%=(346-320)/320.
- Second, the patch selection did not reduce the precision of BugBuilder. The patch selection was highly accurate, and all of the selected patches were correct. Consequently, it did not result in any negative impact on the overall precision of the proposed approach.

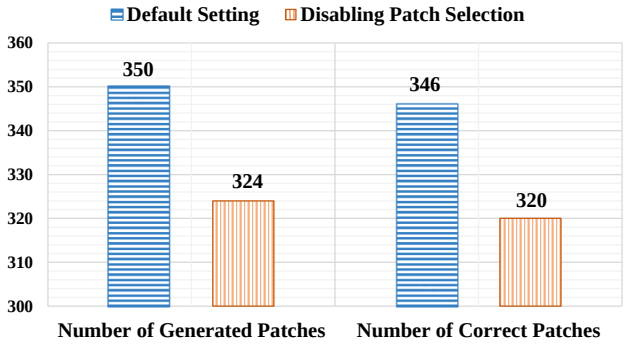


Fig. 3. Impact of Patch Selection

We conclude based on the preceding analysis that patch selection substantially improved the recall of BugBuilder by 8% without any negative effect on its precision.

5.4.5 RQ5: Scalability

Fig. 4 (histogram with a trendline) depicts the relationship between the size of bug-fixing commits and the run time of BugBuilder on such commits. Notably, BugBuilder terminates when its run time reaches the upper limit (40 minutes on a single commit) to avoid extensive execution on a few big commits. The evaluation is conducted on a personal computer with Intel Core i9, 16GB RAM, and Mac OS.

From Fig. 4, we observe that the run time increased significantly when the commit size increased. Around 34% of the commits ran out of the maximal time slot. We also notice that BugBuilder efficiently handled bug-fixing commits that contain up to thirty lines of changes. By increasing the maximal time slot (40 minutes at present) for each commit, BugBuilder has the potential to handle larger commits in the future.

Detecting and reapplying refactorings improved not only the recall of the proposed approach, but also the efficiency of the approach. In total, the proposed approach (BugBuilder) reapplied refactorings on 83 commits, and its average run time on such commits was 21 minutes. We disabled the detection and reapplication of refactorings, and reapplied BugBuilder to such commits. The average run time on such commits was not reduced. Instead, it was surprisingly increased by 24%=(26-21)/21. A possible reason is that removing refactorings from the commit (via refactoring detection and reapplication) would reduce the size of search space for potential patches, and thus reduce the time in searching for the correct patches.

Based on the preceding analysis, we conclude that BugBuilder is scalable, and most of the commits could be handled within 40 minutes. Detecting and reapplying refactorings have a substantial contribution to the performance of the proposed approach.

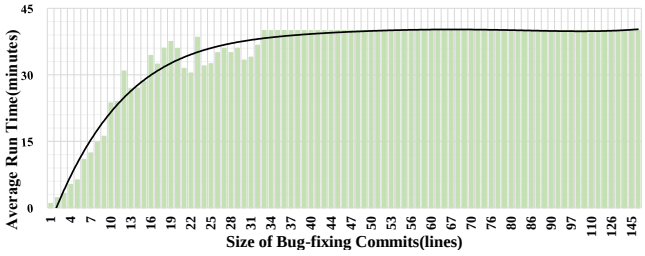


Fig. 4. Scalability of the Approach

5.5 Threats To Validity

A threat to external validity is the limited size of the evaluation data, i.e., bug-fixing commits. In the evaluation, we evaluated BugBuilder on 809 bug-fixing commits collected by Defects4J. Special characteristics of such commits may have biased the conclusions of the evaluation. The commits were selected because Defects4J provided manually constructed concise patches that exclude bug-irrelevant changes. As a result, we could leverage such patches as the ground truth to evaluate the quality of the patches generated by the proposed approach. To the best of our knowledge, Defects4J was the only bug repository that provided manually constructed concise patches for real-world bugs in open-source applications. That is the reason why the evaluation was confined to the bug-fixing commits in Defects4J. To reduce the threat, however, we should evaluate the proposed approach with more bug-fixing commits in the future.

A threat to construct validity is that the evaluation requested manual checking of the generated patches (and patches in Defects4J) whereas manual checking could be inaccurate. During the evaluation, we manually checked the generated patches and their corresponding patches in Defects4J when they did not match each other, to figure out which of them were incorrect. Such manual checking could be biased and inaccurate. Herbold et al. [31] conducted a large-scale empirical study on tangled bug-fixing commits. They requested multiple participants to manually identify bug-fixing changes within such bug-fixing commits. However, their evaluation results suggest that more than ten percentages of the changed lines are hard to label: Participants failed to reach consensus on such changes. It may suggest that sometimes manual labeling of the patches could be debatable. To reduce the threat, we presented typical examples in Section 5.4.2, and made all of the manually checked patches publicly available at [72].

Another threat to construct validity is that the evaluation was based on an unverified assumption that a generated patch is correct if it is identical to that constructed by experts (stored in Defects4J). However, as discussed in Section 5.4.2, human experts may also make incorrect (especially incomplete) patches occasionally, and thus it could be risky to say that a generated patch is deemed correct if it is identical to the manually constructed patch.

6 GrowingBugs: A Bug Repository Built with BugBuilder

In this section, we build a bug repository with BugBuilder. On one side, it may further validate the usefulness of the proposed approach. On the other side, it may provide a new benchmark for bug-related research.

6.1 Design and Principle

One of the fundamental principles in designing GrowingBugs is that the new bug repository should reuse the APIs of Defects4J. We notice that Defects4J represents the state of the art in this field, and has been widely employed. Consequently, inheriting its commonly used APIs could significantly facilitate the users of Defects4J to take advantage of GrowingBugs.

The second principle is that the construction (of the new repository) should not involve intensive human intervention. We should automate as many steps (of the construction) as possible. At the same time, however, the data (especially the bug-fixing patches) should be highly accurate.

Finally, we should collect bugs from various real-world applications. Such applications should come from different domains and be developed by different developers/companies. Extracting bugs from a small number of applications may reduce the diversity of the resulting bug repository. Limited application domains and a limited number of involved developers/companies may also have a negative impact on the diversity of the bug repository.

6.2 Process

Fig. 5 specifies how we construct GrowingBugs. On the first step, we retrieve bug-fixing commits from version control system (GitHub) according to both commit messages and bug tracking systems (or subsystems) including Google-Code [73], Jira [74], GitHub [75], SourceForge [76] and Bugzilla [77]. Similar to existing work [37], [16], [17], we only retrieve these commits whose commit messages contain bug report IDs that could be retrieved from the associated bug tracking systems. Notably, we should list all projects to be exploited before we can start the automated construction. To this end, we add all Java projects from Apache Software Foundation to the list because most of the projects in Defects4J come from Apache community. Besides that, we also add to the list some popular Java applications that are collected automatically from Github. The final list contains 1,214 Java projects. For each of the selected projects, we manually specify the URL to its bug tracking system.

For each of the resulting bug-fixing commits, we automatically locate the configuration and download the third-party libraries requested by the commit. Notably, it is likely that the compilation results in errors because of various reasons, e.g., missing libraries and syntactic errors. In this case, we employ a sequence of heuristics to fix the errors. For example, if the specified version of a library is missing, we automatically replace it with the following new versions, expecting some of them can remove the compilation error. Another example is to replace the explicitly specified JDK version with the latest one. Only if the two successive versions (i.e., V_{buggy} and V_{fixed}) associated with a bug-fixing commit are compilable, the bug commit could be exploited to build the bug repository. Otherwise, it is discarded.

Because BugBuilder depends on triggering test cases to extract concise and complete bug-fixing patches, we

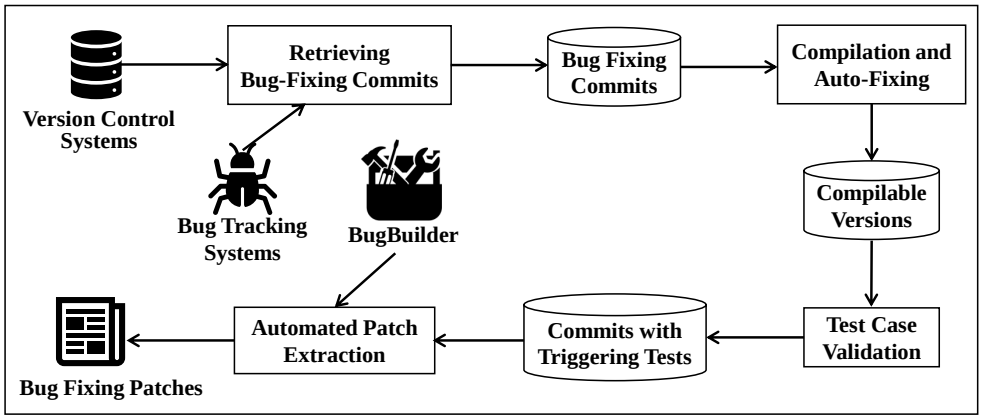


Fig. 5. Constructing `GrowingBugs` with `BugBuilder`

TABLE 4
Resulting Bug Repository

Metrics	Value
Number of Bugs	1,491
Number of Projects	169
Size of Source Code (LOC)	87,467,254
Size of Buggy Code (LOC)	3,934
Total Size of Patches (LOC)	14,881
Minimal Size of Patches (LOC)	1
Maximal Size of Patches (LOC)	196
Number of Triggering Test Cases	3,160

should exclude such commits that are not associated with triggering test cases. To this end, we run the test cases in V_{fixed}, identify all passed test cases, and validate whether any of them fail on the older version V_{buggy}. If not, the commit is discarded. Otherwise, the commit could be fed into BugBuilder to extract the concise bug-fixing patch. If BugBuilder succeeds in extracting the patch for the commit, we add the commit as well as the extracted concise patch to the repository (i.e., GrowingBugs).

Notably, many of the selected projects were discarded without activating BugBuilder because of the following reasons. First, many of the projects could not be compiled or executed successfully, and thus they could not be handled by Defects4J's framework. Second, many of them do not employ bug tracking systems, and thus Defects4J and BugBuilder cannot identify bug-fixing commits from them. Third, many bug-fixing commits do not contain any potential triggering test cases, and thus they were filtered out by Defects4J's framework. If we failed to retrieve (with the help of Defects4J's framework) any bug-fixing commits from a project, the project was discarded. For the preceding three reasons, 952 out of the 1,214 selected projects have been discarded, and only 262 project have been fed into BugBuilder to extract concise patches.

6.3 Results

After months of construction, we built a bug repository of 1,491 real-world bugs and their concise patches. An overview of the repository is presented in Table.4. *Number of bugs* is the total number of bugs in the repository. *Number of projects* is the number of Java projects from which the bugs were extracted. *Size of source code* is the size of the the buggy projects within the bug repository. *Size of buggy code* is the number of buggy code lines that are removed or updated by the bug-fixing patches in the repository. *Total size of patches* is the total number of source code lines that are removed, updated, or inserted by the patches in the repository. *Number of triggering test cases* is the total number of triggering test cases in the repository.

GrowingBugs covers 169 open-source applications and most of such applications are well-known and widely used in the industry. We also notice that such applications cover various domains, including (but not limited to) mathematical computing, image processing, document processing, graphical user interface, compilation, network, middleware, web, data mining, mobile computing, and resource management. Covering a large number of widely used applications from various domains may improve the diversity of the bugs in the repository.

We also notice that the number of bugs from a single application varies significantly from application to application. While Closure Compiler contains 174 bugs, up to 56 applications contain only a single bug. Notably, the number of bugs from a single application could be influenced by various factors, e.g., the size of the application and the evolutionary history of the application. Besides, it is also significantly influenced by the employed process of bug tracking and resolving, i.e., how often bug reports are recorded in bug tracking systems and how often bug fixing patches are accompanied by triggering test cases.

6.4 Challenges and Limitations

The most challenging part of the construction was to make involved applications compilable. It was very often that the specified version of the application resulted in compilation errors. However, it remained challenging to fix such errors automatically although we have proposed a sequence of heuristics to fix them (as specified in Section 6.2). As a result, a large percentage of bug-fixing commits were discarded.

Another challenge was that bug-fixing commits might

not contain triggering test cases that could be leveraged to verify the corresponding patches, or the test cases did not cover all bug-fixing changes. As a result, BugBuilder cannot extract concise patches from such commits.

One of the limitations of the proposed approach is that it relies on bug tracking systems whereas many open-source applications do not employ bug tracking systems at all or rarely used them. According to our experience in the construction of GrowingBugs, less than ten percentages of the applications in GitHub frequently employed professional bug tracking systems (e.g., Bugzilla and Jira) to track software bugs. Some applications employed GitHub to track issues but failed to explicitly distinguish bug reports from other issues, which also prevented BugBuilder from extracting bug-fixing commits for such applications.

We conclude based on the preceding analysis that BugBuilder could be employed to build bug repositories although some technical and non-technical challenges are preventing it from reaching its maximal potential. Future research is required to resolve such challenges.

7 Discussion

7.1 It is Critical to Detect and Reapply Refactorings

Bug-fixing commits may contain three categories of changes: bug-fixing changes, refactorings, and functionality-related bug-irrelevant changes (e.g., implementation of new features). If a bug-fixing commit contains bug-fixing changes only (called `pure bug-fixing commit`), BugBuilder has the potential to extract the patch (that is equivalent to the whole commit) by generating and validating potential patches. If a bug-fixing commit contains both bug-fixing changes and refactorings (but no functionality-related bug-irrelevant changes), BugBuilder leverages the Refactoring Detection and Reapplication (RDR for short) to turn the commit into a pure bug-fixing commit, and then extracts a patch from it. Notably, 24%=192/809 of the bug-fixing commits in Defects4J contain refactorings, which quantitatively suggests the importance of RDR (Section 4.2). RDR has the potential to rescue such commits that otherwise are deemed to be missed. However, the current implementation of BugBuilder supports only eight categories of refactorings, which prevents RDR from reaching its maximal potential: It improved recall by 10% only in the evaluation. Supporting all refactorings in the future may result in further improvement.

Another significant benefit of RDR is that it improves the efficiency of the proposed approach. Excluding refactorings significantly reduces the size of commits, and thus reduces the number of potential patches. Evaluation results in Section 5.4.5 suggest that disabling RDR increased BugBuilder's run time by 24% on refactoring-contained commits.

7.2 Extremely High Precision VS Fairish Recall

We prefer high precision to high recall in extracting bug-fixing patches although both of them are desirable. Only if the proposed approach achieves an extremely high precision, we can guarantee the quality of the bug repositories built with the proposed approach. Our evaluation results in Section 5 confirm that the proposed approach achieved a high precision of 99%, and the automatically generated patches were comparable to (and sometimes better than) patches manually constructed by human experts. That is the reason why we leveraged it to build GrowingBugs.

Notably, a fairish recall (43%) of the proposed approach is acceptable because it could be remedied by applying the proposed approach to more bug-fixing commits. There are massive bug-fixing commits available online, e.g., GitHub. However, improving the recall of the proposed approach would increase the size (and thus the diversity) of the bug repository built by the proposed approach. Consequently, any improvement on the recall is highly desirable if it manages to maintain the precision of the proposed approach.

```
1  diff --git a/src/com/google/javascript/jscomp/
       InlineVariables.java b/src/com/google/javascript/
       jscomp/InlineVariables.java
2  index 857845b..47b8e8a 100644
3  --- a/src/com/google/javascript/jscomp/InlineVariables.
       java
4  +++ b/src/com/google/javascript/jscomp/InlineVariables.
       java
5  @@ -558,13 +558,17 @@ class InlineVariables implements
       CompilerPass {
6        return false;
7    }
8
9    if (value.isFunction()) {
10     Node callNode = reference.getParent();
11     if (reference.getParent().isCall()) {
12   + CodingConvention convention = compiler.
       getCodingConvention();
13     SubclassRelationship relationship =
14   - compiler.getCodingConvention().
       getClassesDefinedByCall(callNode);
15   + convention.getClassesDefinedByCall(callNode);
16     if (relationship != null) {
17       return false;
18     }
19
20   + if(convention.getSingletonGetterClassName(callNode)
       !=null){
21   +       return false;
22   +   }
23     }
24   }
```

Listing 10. Excluding Refactorings May Influence Program Repair and Fault Localization

7.3 Controversy in Isolation of Refactorings

The empirical study conducted recently by Herbold et al. [31] suggests that sometimes different participants cannot reach consensus on which changes are bug-irrelevant (and thus should be isolated from the bug-fixing patches). It may suggest that controversy is almost inevitable in labelling patches. Whether refactorings should be excluded from bug-fixing patches is a typical controversy in labeling of bug-fixing commits. On one side, the the isolation of bug-irrelevant changes from bug-fixing commits (as the authors of Defects4J did) may influence the performance of automated program repair tools [78]. We take the patch in Listing 10 as an example to illustrate the impact. Developers have extracted the expression "compiler.getCodingConvention()" as a new variable "convention" (Line 12) and replaced the original expression (Line 14) with the variable (Line 15). It is a typical *extract variable* refactoring, and thus such changes are isolated from the final bug-fixing patch by BugBuilder. The other changes (i.e.,

changes on Lines 20-22) fix the bug, and thus they constitute the bug-fixing patch. To fix the bug, APR tools should only synthesize the *if* statement (Line 20-22). However, if we do not isolate the *extract variable* refactoring from the patch, APR tools should either synthesize all changes on Lines 12, 14-15, and 20-22, or to synthesize the *if* statement (Line 20-22) where direct variable access "convention" on Line 20 should be replaced with more complex expression "compiler.getCodingConvention()". In any case, however, the patch to be synthesized becomes more complex, which may reduce the chance for APR tools to succeed.

On the other side, however, failing to isolate refactorings from patches may have significant impact on the performance of fault localization tools. To assess the performance of fault localization tools, we usually identify faulty code according to bug-fixing patches: Source code removed or changed by the patches is faulty. Such faulty code serves as the ground truth in assessing the performance of fault localization tools. For the example in Listing 10, if we do not remove the *extract variable* refactoring (Lines 12, 14, and 15) from the patch, Line 14 would be tagged as *faulty code* and thus fault localization tools that locate this line would be counted as "successful". However, this line (Line 14) is in fact bug-free and thus it is *incorrect* for fault localization tools to locate this line.

The conflicting impact of the isolation as explained in the preceding paragraphs makes it difficult to build a *perfect* bug repository. As a compromise, if the patches extracted by BugBuilder exploit methods or variables created by *extract method* refactorings or *extract variable* refactorings that have been excluded from the patches, we create alternative patches for them and let users to decide which patches should be used. For patches exploiting extracted methods, we create their alternative patches by adding the whole *extract method* refactorings to the patches. For patches exploiting extracted variables, we create their alternative patches by replacing the accesses to the extracted variables with the equivalent expressions. For the example in Listing 10, the alternative patch is presented in Listing 11. The access to extracted variable "convention" on Line 20 in Listing 10 has been replaced with the original expression "compiler.getCodingConvention()" on Line 9 in Listing 11. Notably, Lines 12, 14, and 15 in Listing 10 are part of the *extract variable* refactoring, but they are not included by the alternative patch because they are not directly connected to the patch.

```
1  if (value.isFunction()) {
2    Node callNode = reference.getParent();
3    if (reference.getParent().isCall()) {
4   CodingConvention convention = compiler.
         getCodingConvention();
5    SubclassRelationship relationship = convention.
          getClassesDefinedByCall(callNode);
6    if (relationship != null) {
7         return false;
8    }
9  + if(compiler.getCodingConvention().
       getSingletonGetterClassName(callNode) !=null){
10 +        return false;
11 +   }
12   }
13 }
```

Listing 11. Alternative Patch

7.4 Limitations

During the evaluation introduced in Section 5, BugBuilder failed on around 57% of the bug-fixing commits for various reasons. BugBuilder works on a bug-fixing commit only if

- The commit is composed of bug-fixing changes only, or
- The commit is composed of only bug-fixing changes and refactorings.

However, BugBuilder may fail when the commit contains functionality-related bug-irrelevant changes, e.g., implementation of new features, which has a significant negative impact on the usability of the proposed approach. Take the bug-fixing commit in Listing 7 as an example to explain why it is so challenging to distinguish bug-fixing changes from functionality-related bug-irrelevant changes. Adding the functionality to print headers of *CSVFormat* is taken as a bug-fixing action there. However, it could be taken as an implementation of a new feature (printing herders of *CSVFormat*) as well if this functionality has not been specified in the original requirements. Consequently, it is challenging (even for human experts) to distinguish bug-fixing changes from other functionality-related changes without the help of requirements and bug reports. However, automatic and accurate comprehension of requirements and bug reports in plain texts remains challenging, let alone requirements that are often unavailable. Most of the bug-fixing commits where BugBuilder failed to contain functionality-related bug-irrelevant changes, and this is the major reason for the low recall of BugBuilder.

BugBuilder may also fail even if a given commit only contains bug-fixing changes and refactorings. Notably, if the refactorings within the commit are only applicable after the bug-fixing changes, existing refactoring mining tools like *RefactoringMiner* [32] cannot identify such refactorings by comparing the fixed version (v_n) and the original buggy version (v_{n-1}). For example, if developers insert a fragment of source code to fix a bug, and then apply *extract method* refactoring to extract the inserted source code as a new method, *RefactoringMiner* cannot identify the extract method refactoring because the extracted source code is not available in the original buggy version. As a result, BugBuilder would fail to split the commit accurately into a refactoring patch and a following bug-fixing patch. If the refactorings are required by the bug fix (and thus applied before the fix) or independent of the fix (and thus could be applied before the fix), BugBuilder has the potential to split the bug-fixing commit into a refactoring patch and its following bug-fixing patch.

Although BugBuilder succeeded on only 43% of the bug-fixing commits, it enables automatic construction of large bug-patch repositories for the following reasons. First, BugBuilder is automated with extremely high precision. Second, BugBuilder is not biased by the types of bugs, but affected by only whether the fixes are mixed with other functionality-related changes. Finally, there are many open-source projects to be exploited. Applying BugBuilder to such projects automatically could significantly increase the size of bug repositories.

7.5 Further Improvement on Recall

In theory, BugBuilder should be able to generate complete and concise patches for all pure bug-fixing commits (without any bug-irrelevant changes). However, BugBuilder succeeded on only 281 out of the 400 pure bug-fixing commits (called *pure commits* for short) in Defects4J. The major reason for the failure is the setting of the maximal time slots: 71%=84/119 of the failed pure commits ran out of the maximal time slots. Increasing the time slots may improve the recall in the future.

```
1  /*StringEscapeUtils.java from Lang(BugID:52)*/
2  public class StringEscapeUtils {
3      ...
4   switch (ch) {
5        case '\'':
6              if (escapeSingleQuote) {
7                    out.write('\\');
8              }
9              out.write('\'');
10             break;
11       case '"':
12             out.write('\\');
13             out.write('"');
14             break;
15       case '\\':
16             out.write('\\');
17             out.write('\\');
18             break;
19 +     case '/' :
20 +           out.write('\\');
21 +           out.write('/');
22 +           break;
23       default :
24             out.write(ch);
25             break;
26 }
```

Listing 12. Potential Redundancy in Patches

Another reason for the failure is the redundancy of some patches. The patch in Listing 12 is a good example where potential redundancy prevents the proposed approach from extracting the correct patch. This example comes from Lang and the manual patch provided by Defects4J is publicly available at https://github.com/rjust/Defects4J/blob/master/framework/projects/Lang/patches/52.src.patch The patch inserts four lines of source code (i.e., Lines 19-22) to handle a special case where *ch*='/ '. We agree with Defect4J that all of the changes on the four lines are bug-fixing changes. However, rejecting some changes (i.e., changes on Lines 21-22) of the patch would not change the semantics of the program: Removing Line 22 means that the program will enter the following `default` branch where 'out.write(ch)' (Line 24) would be executed. We also notice that *ch*='/' (Line 19), and thus 'out.write(ch)' (Line 24) is equivalent to the removed statement 'out.write('/')' (Line 21). Consequently, removing Lines 21-22 (resulting in a new candidate patch) happens to keep the semantics of the program. However, BugBuilder was confused by such a case where multiple patches are semantically equivalent.

Improving the implementation of the proposed approach to support additional categories of refactorings may also significantly improve recall in future. Notably, 57%=109/192 of the refactoring-containing commits in Defects4J contain some refactorings unsupported by the current implementation. Consequently, supporting all such refactoring in future has the potential to double the effect of RDR that currently improves recall by 10%.

8 Conclusions and Future Work

Bug-related research extensively depends on large-scale and high-quality repositories of real-world bugs. However, existing approaches to building such repositories either fail to exclude bug-irrelevant changes from patches or require extensive human intervention. To this end, in this paper, we propose a novel approach, called *BugBuilder*, to extracting complete and concise patches from bug-fixing commits automatically. BugBuilder excludes refactorings by automatic detection and reapplication of refactorings. On the resulting refactoring-included version, BugBuilder generates all potential patches and validates them with test cases. If only a single potential patch for a bug-fixing commit passes the validation, BugBuilder presents it as a patch for the associated bug report. If more than one potential patch pass the validation, BugBuilder employs a sequence of heuristic rules to select the most likely one from them. BugBuilder has been evaluated on 809 bug-fixing commits in Defects4J. Our evaluation results suggest that it successfully generated complete and concise patches for more than forty percent of the bug-fixing commits, and its precision was even higher than human experts. With the help of BugBuilder, we also built GrowingBugs, a large-scale and high-quality repository of real-world bugs. The resulting bug repository could serve as a publicly available benchmark for bug-related research.

We have released both BugBuilder and GrowingBugs on GitHub to facilitate both replications of the evaluation and potential third-party reuse. Although GrowingBugs is a generic Java bug repository, it is practical and meaningful in future to build repositories of bugs in specific domains or planform, e.g., real-world bugs in mobile applications and real-world bugs in machine learning applications. We also plan to host bug-related competitions, e.g., fault localization and program repair, based on our bug repository. Finally, it could be potentially fruitful to support additional refactoring types and to exploit additional strategies to fix compilation errors in subject applications.

Acknowledgments

The authors would like to thank the anonymous reviewers from ICSE'2021 for their insightful comments and constructive suggestions.

This work was partially supported by the National Natural Science Foundation of China (62172037) and CCF-Huawei Formal Verification Innovation Research Plan.

References

[1] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[2] J. Sohn and S. Yoo, "FLUCCS: Using code and change metrics to improve fault localization," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 273–283.

[3] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 169–180.

[4] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller, "Evaluating and improving fault localization," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 609–620.

[5] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. Le Traon, "Bench4BL: reproducibility study on the performance of ir-based bug localization," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 61–72.

[6] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang, "Hydra: Massively compositional model for cross-project defect prediction," *IEEE Transactions on software Engineering*, vol. 42, no. 10, pp. 977–998, 2016.

[7] A. Majd, M. Vahidi-Asl, A. Khalilian, P. Poorsarvi-Tehrani, and H. Haghighi, "SLDeep: Statement-level software defect prediction using deep-learning model on static code features," *Expert Systems with Applications*, vol. 147, p. 113156, 2020.

[8] Z. He, F. Peters, T. Menzies, and Y. Yang, "Learning from open-source projects: An empirical study on defect prediction," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. IEEE, 2013, pp. 45–54.

[9] J. Nam, S. Wang, Y. Xi, and L. Tan, "A bug finder refined by a large set of open-source projects," *Information and Software Technology*, vol. 112, pp. 164–175, 2019.

[10] D. Jeffrey, M. Feng, N. Gupta, and R. Gupta, "BugFix: A learning-based tool to assist developers in fixing bugs," in *2009 IEEE 17th International Conference on Program Comprehension*. IEEE, 2009, pp. 70–79.

[11] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus, "Automatic repair of real bugs in java: A large-scale experiment on the defects4j dataset," *Empirical Software Engineering*, vol. 22, no. 4, pp. 1936–1964, 2017.

[12] S. H. Tan and A. Roychoudhury, "relifix: Automated repair of software regressions," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, 2015, pp. 471–482.

[13] B. Daniel, V. Jagannath, D. Dig, and D. Marinov, "ReAssert: Suggesting repairs for broken unit tests," in *2009 IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 2009, pp. 433–444.

[14] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair," in *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, Eds. ACM, 2018, pp. 789–799. [Online]. Available: https://doi.org/10.1145/3180155.3180182

[15] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, S. Uchitel, A. Orso, and M. P. Robillard, Eds. IEEE / ACM, 2017, pp. 416–426. [Online]. Available: https://doi.org/10.1109/ICSE.2017.45

[16] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, 2014, pp. 437–440.

[17] V. Dallmeier and T. Zimmermann, "Extraction of bug localization benchmarks from history," in *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 433–436.

[18] H. Do, S. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering*, vol. 10, no. 4, pp. 405–435, 2005.

[19] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 01, pp. 34–55, jan 2017.

[20] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 802–811.

[21] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," *IEEE Transactions on Software Engineering*, vol. 47, pp. 1943–1959, 2019.

[22] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *International Conference on Learning Representations (ICLR)*, 2020.

[23] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: context-based code transformation learning for automated program repair," in *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, G. Rothermel and D. Bae, Eds. ACM, 2020, pp. 602–614. [Online]. Available: https://doi.org/10.1145/3377811.3380345

[24] H. Zhong, X. Wang, and H. Mei, "Inferring bug signatures to detect real bugs," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[25] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: Benchmarks for evaluating bug detection tools," in *Workshop on the Evaluation of Software Defect Detection Tools*, vol. 5, 2005.

[26] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer, "The ManyBugs and IntroClass benchmarks for automated repair of C programs," *IEEE Transactions on Software Engineering*, vol. 41, no. 12, pp. 1236–1256, 2015.

[27] S. H. Tan, J. Yi, S. Mechtaev, A. Roychoudhury *et al.*, "Codeflaws: a programming competition benchmark for evaluating automated program repair tools," in *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 2017, pp. 180–182.

[28] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, "QuixBugs: A multi-lingual program repair benchmark set based on the quixey challenge," in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2017, pp. 55–56.

[29] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in Android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 187–198.

[30] M. Böhme, E. O. Soremekun, S. Chattopadhyay, E. Ugherughe, and A. Zeller, "Where is the bug and how is it fixed? an experiment with practitioners," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 117–128.

[31] S. Herbold, A. Trautsch, B. Ledel, A. Aghamohammadi, T. A. Ghaleb, K. K. Chahal, T. Bossenmaier, B. Nagaria, P. Makedonski, M. N. Ahmadabadi *et al.*, "A fine-grained data set and analysis of tangling in bug fixing commits," *arXiv preprint arXiv:2011.06244*, 2020.

[32] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, "Accurate and efficient refactoring detection in commit history," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 483–494. [Online]. Available: https://doi.org/10.1145/3180155.3180206

[33] Y. Jiang, H. Liu, N. Niu, L. Zhang, and Y. Hu, "Extracting concise bug-fixing patches from human-written patches in version control systems," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 686–698.

[34] H. Mei and L. Zhang, "Can big data bring a breakthrough for software automation?" *Sci. China Inf. Sci.*, vol. 61, no. 5, pp. 056 101:1–056 101:3, 2018. [Online]. Available: https://doi.org/10.1007/s11432-017-9355-3

[35] J. Spacco, J. Strecker, D. Hovemeyer, and W. Pugh, "Software repository mining with marmoset: An automated programming project snapshot and testing system," in *Proceedings of the 2005 International Workshop on Mining Software Repositories*, 2005, pp. 1–5.

[36] P. Gyimesi, B. Vancsics, A. Stocco, D. Mazinanian, A. Beszédes, R. Ferenc, and A. Mesbah, "BugsJS: A benchmark of JavaScript bugs," in *12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 90–101.

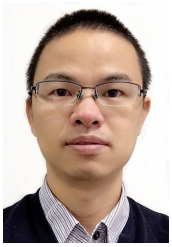
[37] R. K. Saha, Y. Lyu, W. Lam, H. Yoshida, and M. R. Prasad, "Bugs.jar: a large-scale, diverse dataset of real-world java bugs," in *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018, pp. 10–13.

[38] F. Madeiral, S. Urli, M. Maia, and M. Monperrus, "BEARS: An extensible java bug benchmark for automatic program repair studies," in *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2019, pp. 468–478.

[39] D. A. Tomassi, N. Dmeiri, Y. Wang, A. Bhowmick, Y.-C. Liu, P. T. Devanbu, B. Vasilescu, and C. Rubio-González, "BugsWarm: mining and continuously growing a dataset of reproducible failures and fixes," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 339–349.

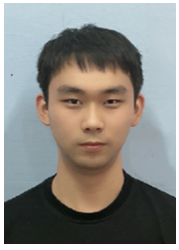
[40] D. Kawrykow and M. P. Robillard, "Non-essential changes in version histories," in *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 2011, pp. 351–360.
[41] B. Fluri, M. Wursch, M. PInzger, and H. Gall, "Change distilling:tree differencing for fine-grained source code change extraction," *IEEE Transactions on Software Engineering*, vol. 33, no. 11, pp. 725–743, 2007.
[42] F. Thung, D. Lo, and L. Jiang, "Automatic recovery of root causes from bug-fixing changes," in *20th Working Conference on Reverse Engineering (WCRE)*. IEEE, 2013, pp. 92–101.
[43] https://github.com/FasterXML/jackson-databind.
[44] https://github.com/FasterXML/jackson-databind/issues/1809.
[45] A. Almogahed and M. Omar, "Refactoring techniques for improving software quality: Practitioners' perspectives," *Journal of Information and Communication Technology*, vol. 20, no. 4, pp. 511–539, 2021.
[46] O. Hamdi, A. Ouni, E. A. AlOmar, M. O. Cinnéide, and M. W. Mkaouer, "An empirical study on the impact of refactoring on quality metrics in android applications," in *2021 IEEE/ACM 8th International Conference on Mobile Software Engineering and Systems (MobileSoft)*. IEEE, 2021, pp. 28–39.
[47] P. Weissgerber and S. Diehl, "Identifying refactorings from source-code changes," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 231–240.
[48] D. Silva and M. T. Valente, "RefDiff: detecting refactorings in version histories," in *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 2017, pp. 269–279.
[49] M. Kim, D. Cai, and S. Kim, "An empirical investigation into the role of api-level refactorings during software evolution," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 151–160.
[50] F. Palomba, A. Zaidman, R. Oliveto, and A. De Lucia, "An exploratory study on the relationship between changes and refactoring," in *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*. IEEE, 2017, pp. 176–185.
[51] D. Dig, K. Manzoor, R. E. Johnson, and T. N. Nguyen, "Effective software merging in the presence of object-oriented refactorings," *IEEE Transactions on Software Engineering*, vol. 34, no. 3, pp. 321–335, 2008.
[52] S. Kaur, L. K. Awasthi, and A. Sangal, "A brief review on multi-objective software refactoring and a new method for its recommendation," *Archives of Computational Methods in Engineering*, vol. 28, no. 4, pp. 3087–3111, 2021.
[53] A. Peruma, S. Simmons, E. A. AlOmar, C. D. Newman, M. W. Mkaouer, and A. Ouni, "How do i refactor this? an empirical study on refactoring trends and topics in stack overflow," *Empirical Software Engineering*, vol. 27, no. 1, pp. 1–43, 2022.
[54] H. He, Y. Xu, Y. Ma, Y. Xu, G. Liang, and M. Zhou, "A multi-metric ranking approach for library migration recommendations," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 72–83.
[55] N. Tsantalis, A. Ketkar, and D. Dig, "Refactoringminer 2.0," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020 (Online).
[56] https://www.eclipse.org/articles/Article-LTK/ltk.html.
[57] G. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Software*, vol. 23, no. 4, pp. 76–83, 2006.
[58] https://github.com/cli/cli.
[59] https://issues.apache.org/jira/browse/CLI-51.
[60] https://storage.googleapis.com/google-code-archive/v2/code.google.com/closure-compiler/issues/issue-1144.json.
[61] https://github.com/apache/commons-codec.
[62] https://issues.apache.org/jira/browse/CODEC-229.
[63] https://github.com/rjust/defects4j/blob/master/framework/projects/Codec/patches/17.src.patch.
[64] https://github.com/apache/commons-csv.
[65] https://issues.apache.org/jira/browse/CSV-120.
[66] https://github.com/rjust/defects4j/blob/master/framework/projects/Csv/patches/10.src.patch.
[67] https://github.com/google/gson.
[68] https://github.com/google/gson/pull/800.
[69] https://github.com/rjust/defects4j/blob/master/framework/projects/Gson/patches/6.src.patch.
[70] https://github.com/google/closure-compiler.
[71] https://storage.googleapis.com/google-code-archive/v2/code.google.com/closure-compiler/issues/issue-229.json.
[72] https://github.com/jiangyanjie/BugBuilder.
[73] https://code.google.com/.
[74] https://www.atlassian.com/software/jira.
[75] https://github.com/.
[76] https://sourceforge.net/.
[77] https://www.bugzilla.org/.
[78] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 302–313. [Online]. Available: https://doi.org/10.1145/3338906.3338911

Yanjie Jiang received her BE degree from the College of Information Engineering, Northwest A&F University in 2017. She is currently working toward a Ph.D. degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. Her current research interests include software refactoring and software quality.



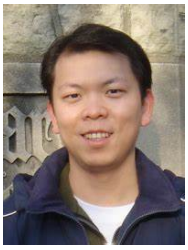
Hui Liu is a professor at the School of Computer Science and Technology, Beijing Institute of Technology, China. He received a BS degree in control science from Shandong University in 2001, an MS degree in computer science from Shanghai University in 2004, and a Ph.D. degree in computer science from Peking University in 2008. He was a visiting research fellow in CREST at University College London. He served on the program committees and organizing committees of prestigious conferences, such as ASE, ICSME, RE, and SANER. He is particularly interested in software refactoring, AI-based software engineering, and software quality. He is also interested in developing practical tools to assist software engineers.

Xiaoqing Luo received BS degree from Xidian University in 2020. She is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. She is particularly interested in software refactoring and software evolution.



Zhihao Zhu received BS degree from Xidian University in 2020. He is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. He is particularly interested in fault localization.

Xiaye Chi received his BE degree from Beijing Institute of Technology in 2021. He is currently working toward the master's degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. His current research interests include software refactoring and software quality.



Nan Niu is an associate professor in the University of Cincinnati's Department of Electrical Engineering and Computer Science, Cincinnati, Ohio, USA. His research interests include requirements engineering, scientific software development, and human-centric computing. He received his Ph.D. in computer science from the University of Toronto. He is an associate editor of Requirements Engineering and a Senior Member of IEEE.

Yuxia Zhang received the BS degree in software engineering from Northwest University, in 2015. She received the PhD degree in the School of Electronics Engineering and Computer Science, Peking University, in 2019. Her research interests include mining software repositories and open source software ecosystems, mainly focusing on the commercial participation in open source.

Yamin Hu received his BE degree from the College of Information Engineering, Northwest A&F University in 2016. He is currently working toward a Ph.D. degree at the School of Computer Science and Technology, Beijing Institute of Technology, under the supervision of Dr. Hui Liu. His current research interests include software refactoring and software quality.

Pan Bian is a Senior Engineer in Huawei Technologies Co., Ltd, Beijing, China. He received the B.S. degree in 2010, the M.S. degree in 2013 and the Ph.D. degree in 2020 from School of Information, Renmin University of China. His research interests focus on static bug detection and automatic bug repair.

Lu Zhang is a professor at the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He received both Ph.D. and BSc in Computer Science from Peking University in 2000 and 1995 respectively. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, UK. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of Journal of Software Maintenance and Evolution: Research and Practice and Software Testing, Verification, and Reliability. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.