PRECFIX: Large-Scale Patch Recommendation by Mining Defect-Patch Pairs

Xindong Zhang
zxd139922@alibaba-inc.com
Alibaba Group

Chenguang Zhu*
cgzhu@utexas.edu
University of Texas at Austin

Yi Li
yi_li@ntu.edu.sg
Nanyang Technological University

Jianmei Guo
jianmei.gjm@alibaba-inc.com
Alibaba Group

Lihua Liu
lihua.llh@alibaba-inc.com
Alibaba Group

Haobo Gu
haobo.haobogu@alibaba-inc.com
Alibaba Group

ABSTRACT

Patch recommendation is the process of identifying errors in software systems and suggesting suitable fixes for them. Patch recommendation can significantly improve developer productivity by reducing both the debugging and repairing time. Existing techniques usually rely on complete test suites and detailed debugging reports, which are often absent in practical industrial settings. In this paper, we propose PRECFIX, a pragmatic approach targeting large-scale industrial codebase and making recommendations based on previously observed debugging activities. PRECFIX collects defect-patch pairs from development histories, performs clustering, and extracts generic reusable patching patterns as recommendations. We conducted experimental study on an industrial codebase with 10K projects involving diverse defect patterns. We managed to extract 3K templates of defect-patch pairs, which have been successfully applied to the entire codebase. Our approach is able to make recommendations within milliseconds and achieves a false positive rate of 22% confirmed by manual review. The majority (10/12) of the interviewed developers appreciated PRECFIX, which has been rolled out to Alibaba to support various critical businesses.

KEYWORDS

Defect detection, patch generation, patch recommendation.

ACM Reference Format:
Xindong Zhang, Chenguang Zhu, Yi Li, Jianmei Guo, Lihua Liu, and Haobo Gu. 2020. PRECFIX: Large-Scale Patch Recommendation by Mining Defect-Patch Pairs. In *Software Engineering in Practice (ICSE-SEIP '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3377813.3381356

1 INTRODUCTION

Patch recommendation is the process of identifying errors in software systems and suggesting suitable fixes. Recent studies show that on average 49.9% of software developers' time has been spent in debugging and about half of the development costs are associated with debugging and patching [5, 11, 49]. Automated patch recommendation can significantly reduce developers' debugging efforts and the overall development costs, improving software quality and system reliability.

Recommending patches automatically is a challenging task, especially for large-scale industrial codebase. Many state-of-the-art techniques from the literature make assumptions on the existence of auxiliary development artifacts such as complete test suites and detailed issue tracking and debugging reports, which may not be readily available in the day-to-day development environment. To better understand the specific challenges in applying existing techniques in the development environment of Alibaba, we investigated the development practices and the entire codebase at Alibaba, ended up extracting a sample benchmark set which includes over 10,000 software projects, spanning over 15 million commits and 30 million files. We have made the following observations through the study.

First, the project codebase often has insufficient documentation and manual labeling of defects and patches is hardly possible, which makes accurate patch mining and generation difficult. For example, recent studies proposed approaches based on *clone detection* [36], *patch mining* [17], *information retrieval* [59], and *machine learning* [6, 26] for fault localization and repair, which naturally require a large amount of labeled data. On the other hand, existing methods for automatic defect and patch identification suffer from inaccuracy. For instance, the SZZ algorithm [48] only achieves less than 25% accuracy on our benchmark set, which is mainly due to the inaccurate/missing bug reports and log messages.

Second, the patch recommendation process has to be highly responsive in suggesting patches, in order to be useful in the everyday development routine. However, many existing fault localization and patch generation techniques require dynamic execution of test suites. For example, the *spectrum-based* [1, 18] and *mutation-based* [35, 40] fault localization techniques both assume strong test cases and utilize test execution results to identify defects. The *generate-and-validate* approaches [7, 22, 32] for patch generation search for candidate patches and validate their correctness using test suites. The problem with these techniques is that the test suites in our benchmark set may not be complete and the test execution often takes significant amount of time, making responsive online patch recommendation impossible.

Finally, many fault localization and patch generation techniques focus on specific domains such as concurrency [30, 31], HTML content generation [46], and memory leaks [10]. Yet, our benchmark

*Corresponding author

ICSE-SEIP '20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7123-0/20/05…$15.00
https://doi.org/10.1145/3377813.3381356

set covers a variety of business scenarios from diverse application domains, including e-commerce, logistics, finance, cloud computing, etc. These domain-specific techniques may work well in their targeted cases, but they are often not generalizable to the diverse applications from the company codebase.

The above mentioned characteristics of the codebase and development environment make the adoption of existing techniques unsuitable. In this paper, we propose a pragmatic patch recommendation approach Precfix, with customized improvements in terms of both the *precision* and *efficiency* when applied on large-scale industrial codebase. First, Precfix does not rely on labeled defects or patches, which are difficult to obtain in practice. Instead, we automatically mine a large number of *defect-patch pairs* from historical changes. To improve the accuracy of the mining results, we introduce optimizations which take into account the characteristics of typical developer behaviors in committing bug fixing changes. We also allow developers in the loop to provide feedback on the quality of the recommended patches, and the feedback is used to improve the precision of future recommendations. Second, since the defects and patches are mined from the entire company codebase and we use generic features when processing and clustering them, Precfix is generally applicable to all company projects written in different languages, handling different business logic, and deployed on different platforms. Finally, Precfix consists of an *offline patch discovery* phase and an *online patch recommendation* phase. The online phase is designed to be extremely responsive and can finish recommending patches within milliseconds in practice. We found that being scalable and efficient is extremely important for patch recommendation tools to be integrated into day-to-day interactive and repetitive development tasks, such as code reviewing.

Precfix has been implemented and deployed as an internal web service in Alibaba. It is also integrated as a part of the code review process and provides patch recommendations whenever developers commit new changes to the codebase. We evaluated the effectiveness and efficiency of Precfix, and demonstrate its usefulness on a large-scale industrial benchmark with over 10,000 projects, spanning over more than 15 million commits and 30 million files.

Contributions. We make the following contributions in this paper.

- We propose Precfix— a semi-automated patch recommendation tool for large scale industrial codebase.
- Precfix implements customized optimizations in defect-patch pair mining and clustering, which help improve the accuracy of patch recommendation over existing techniques.
- Precfix managed to extract 3K defect templates from 10K projects. Our approach is able to make recommendations within milliseconds and achieves a false positive rate of 22% confirmed by manual review.
- We conducted a small-scale user study and the majority (10/12) of the interviewed developers appreciated Precfix, which has been rolled out to Alibaba to support various critical businesses.

2 RELATED WORK

The techniques presented in this paper intersect with different areas of research. In this section, we compare Precfix with fault localization, automated patch generation, and patch recommendation.

2.1 Fault Localization

Fault localization [41, 51, 54, 60] is the activity of identifying the locations of faults in a program. Many different types of fault localization techniques have been proposed. Spectrum-based fault localization [1, 18] utilizes test coverage information to pinpoint faulty program or statistical techniques. For example, Tarantula [18] uses a homonym ranking metric to calculate the suspiciousness values of statements, which are calculated according to the frequency of the statements in passing and failing test cases. Mutation-based fault localization [35, 40] mutates a program and runs its test cases, using the test results to locate faults. For example, Metallaxis [40] generates a set of mutants for each statement, assigns each mutant a suspiciousness score, and aggregates the scores to yield the suspiciousness of the statement. Other faults localization techniques identify locations of faults in some alternative ways, including dynamic program dependency analysis [2, 45], stack trace analysis [53, 55], conditional expressions mutation analysis [58], information retrieval [59], and version history analysis [23, 44].

2.2 Automated Patch Generation

Automated patch generation [11, 34, 50, 57] aims to automatically repair software systems by producing a fix that can be validated before it is fully accepted into the system. Automated patch generation techniques can be divided into two main categories: generate-and-validate approaches and semantics-driven approaches. Generate-and-validate approaches [7, 22, 32, 43, 52] iteratively execute two activities: patch generation, which produces candidate patch of the bug by making atomic changes or applying bug fix templates; patch validation, which checks the correctness of the generated solutions by running test cases. For example, GenProg [52] uses genetic programming to guide the generation and validation activities. At every iteration, it randomly determines the location to apply an atomic change, according to a probability distribution that matches the suspiciousness of the statements computed with fault localization algorithms. Every candidate solution is validated running the available test suite. It defines a fitness function that measures the fitness of each program variant based on the number of passing and failing test cases. Semantics-driven approaches [21, 30, 38] formally encode the problem, either as a formula whose solutions correspond to the possible fixes, or as an analytical procedure whose outcome is a fix. A solution found by such approaches is correct by construction, thus no validation is needed. For example, SemFix [38] uses fault localization to identify the statement that should be changed, then tries to synthesize a fix by modifying a branch predicate or changing the right hand side of an assignment.

2.3 Patch Recommendation

Patch recommendation [3, 16, 20, 24, 36] suggests a few candidate changes which may repair a given fault. In some cases, the recommended patches are perfect fixes, while in other cases some efforts are required from the developers to produce the final fix. Although these techniques do not guarantee a working repair, their results are still useful in assisting developers in deriving the patch. A number of patch recommendation techniques have been proposed so far. For example, Getafix [3] from Facebook learns recurring fix patterns for static analysis warnings and suggests fixes for future occurrences of the same bug category. It firstly splits a given set of example fixes into AST-level edits, then it learns recurring fix patterns from these edits based on a clustering technique which produces a hierarchy of fix patterns. Finally, given a bug under fix, it finds suitable fix patterns, ranks candidate fixes, and suggests the top-most fixes to developers. As another example, CLEVER [36] from Ubisoft aims to intercept risky commits before they reach the central repository. It first builds a metric-based model to assess the risky level of incoming commits, then it uses clone detection to compare code blocks in risky commits with some known historical fault-introducing commits.

All these aforementioned techniques depend on existing patches or already-known bug patterns. In contrast, we do not assume enough debugging reports, and we extract templates of defect-patch pairs through data mining. Open-source dataset such as Defect4J [19] contains labeled defects and the corresponding patches, which have been examined and analyzed by many researchers. Yet, recent studies [56] indicate that many state-of-the-art patch generation techniques has the problem of over-fitting to specific benchmark set. Therefore, many of the existing techniques cannot be directly applied on the industrial codebase, which is quite different from the open-source dataset in many ways.

3 PRELIMINARY STUDY

To better understand the codebase at Alibaba and the challenges in applying existing techniques in the industrial development environment, we conducted a preliminary study of the usage scenarios of patch recommendation techniques within the company and empirically analyzed the characteristics of the company codebase.

3.1 Challenges for Existing Techniques

Through manual inspection, interviews with developers, and empirical studies, we identified three key challenges for existing fault localization and automated patch generation techniques to be successfully applied on our benchmark.

Insufficient Labeled Data. A lot of fault localization and automated patch generation techniques require labeled defect and patch samples to be able to extract patterns of typical bug fixes. Yet, this is a substantial obstacle in our case, since there exists very few labeled defects or patches in the company codebase. Moreover, due to the widespread legacy code in the codebase, a large number of software projects only have partial debugging reports and very limited test cases. The commit messages may be succinct and do not follow any standard template either. Therefore, it is challenging to label defects and the associated fixes manually, given the size and complexity of the codebase. The business logic and bug fix patterns of the internal company codebase are quite different from that of open source projects [42, 47]. Thus, we decide not to directly use the labeled data from open-source projects.

High Responsive Standard. The application scenario of patch recommendation in Alibaba is highly interactive. Patch recommendation needs to be run whenever new commits are submitted by developers for code review. The recommended patches are then checked by developers, who may decide to incorporate the suggestions into the commits. On average, a developer submits three to four commits per week, and both the submitter and reviewer expect

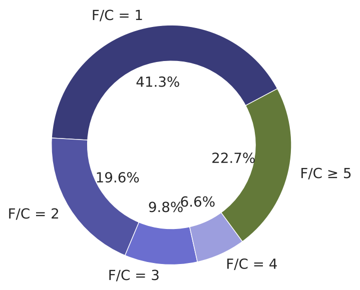


Figure 1: Distribution of the number of changed files per keyword-matched commit (F/C).

prompt patch recommendations to avoid delays during the review process. Therefore, the responding time for patch recommendation is supposed to be reasonably low in order to be integrated into the development routine. This renders some automated patch generation approaches inappropriate, since they need to repeatedly compile and execute tests for each identified defect.

Generalizability Requirement. Our benchmark set consists of more than 10K projects supporting more than 100 software applications. These applications cover a variety of domains including e-commerce, finance, cloud computing, and artificial intelligence, many of which are used by millions of users on a daily basis. The patch recommendation techniques should be generalizable to cover all different projects and defect types.

3.2 Challenges in Defect-Patch Identification

There exists techniques [27–29, 48] which automatically identify changes of certain kinds, e.g., bugs and fixes, from commit histories. These techniques, such as the SZZ algorithm [48], can be useful for labeling *defect-patch pairs* in the absence of high-quality labeled data. The high-level idea is to first locate *bug-fixing commits* based on keywords in commit messages and debugging reports. For example, a commit is considered as bug-fixing commit if it appears in a bug report or its commit message contains keywords such as "bug" and "fix". For each identified bug-fixing commit, one can trace each line of changed code back in history to locate the *bug-inducing commits* using version control information such as `git-blame` [12].

Various optimizations have been introduced in the SZZ algorithm to reduce the false positives. For example, the timestamp of a candidate bug-inducing commit is compared with the time when the bug is reported to rule out unreasonable results. Yet, we found that even with these optimizations, the SZZ algorithm still does not perform well on our benchmark (25% true positive rate). To figure out the reason why the SZZ algorithm does not perform well on our codebase, we quantitatively analyzed the dataset and identified two stages in the algorithm where imprecision can be introduced: (1) the identification of bug-fixing commits, and (2) the location of bug-inducing commits via back-tracing in history.

Imprecision in Locating Bug-Fixing Commits. The first reason causing imprecision is that the keyword matching approach is not always reliable. For example, through manual inspection, we discovered that a commit with the commit message containing the keyword "fix", does not change any program logic and only modifies the label of an Android UI button. Even if a commit does fix

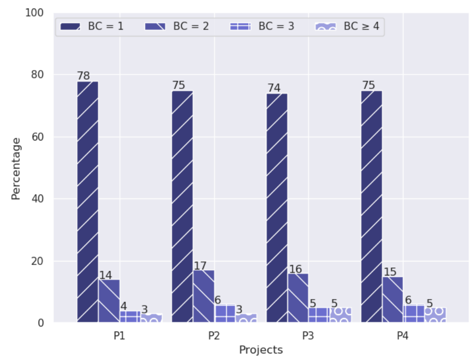


Figure 2: Distributions of the number of bug-inducing commits (BC).

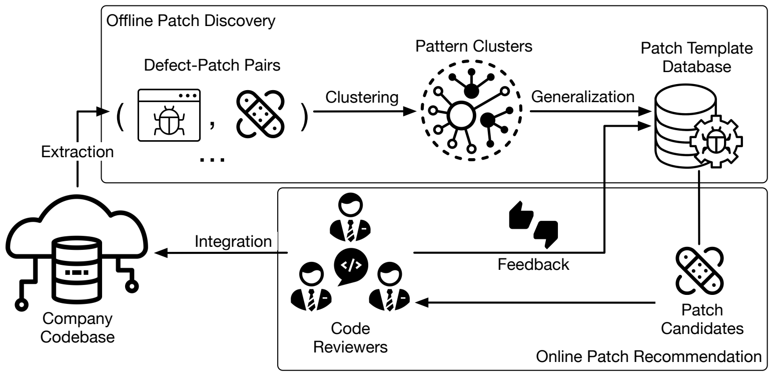


Figure 3: Overview of the Precfix workflow.

a bug, it can also make irrelevant changes to other parts of the code. Fig. 1 shows the distribution of the number of changed files in each commit matched by at least one keyword. The keywords we used include "fix", "bug", "repair", "wrong", "fail", "problem", and their corresponding Chinese translations. About 60% of the commits change more than one file, among which 40% touch over two files. Many such commits are multi-purpose, causing imprecision in locating bug-fixing commits.

Imprecision in Locating Bug-Inducing Commits. Additional imprecision can be introduced when there are multiple bug-inducing commits in the history. When back-tracing in history, it is possible to end up with more than one commits, each partially contributing to the defect. We randomly select four projects from the dataset. Fig. 2 plots the distribution of the number of bug-inducing commits identified by the SZZ algorithm for each bug-fixing commit. For example, for project $P1$, among all the defects, 78% are induced by one commit, 14% are induced by two commits, 4% are induced by three commits, and 3% are induced by more than three commits. We found that the commit-level back-tracing often introduces too much irrelevant changes for the similar reason discussed before.

4 OUR APPROACH

Fig. 3 overviews the workflow of Precfix. Precfix consists of an *offline patch discovery* component and an *online patch recommendation* component. The patch discovery component first extracts potential defect-patch pairs from commits in the version controlled history, clusters defect-patch pairs based on their similarity, and finally extracts generic patch templates to be stored in a database. The patch recommendation component recommends patch candidates to developers and collects their feedback. It removes patches rejected by developers, and includes manually crafted patch templates submitted by developers to improve the patch database.

4.1 Patch Discovery

The offline patch discovery component performs three steps to generate patch templates: extracting defect-patch pairs, clustering defect-patch pairs, and collecting generic patch templates.

4.1.1 *Extracting Defect-Patch Pairs.*

The first step is to extract a large number of defect-patch pairs from the codebase. We make two improvements to the SZZ algorithm, to adapt to the needs of the industrial codebase.

Constraining the Number of Changed Files. The first adjustment is to set a threshold on the number of files modified in a bug-fixing commit, filtering out any commit that exceed the threshold. In this way, we could reduce the false positives that are caused by multi-purpose commits (C.f. Sect. 3). We heuristically chose five as the threshold, which help reduce false positives without discarding too many candidate commits (22.7% as indicated in Fig. 1).

Identifying Bug-Inducing Code Snippets. The second improvement aims to improve the precision of identifying bug-inducing changes. As studied in Sect. 3, commit-level back-tracing in history tends to be imprecise, especially when there are many multi-purpose commits contributing to the defect.

Therefore, instead of tracing commits back in time and identifying bug-inducing commits, we directly identify *bug-inducing code snippets*. In fact, the differences before and after the introduction of a bug-fixing commit already contain information about both the defects and the patches. Now we describe the improved method-level defect-patch pair extraction in more details.

Method-Level Defect-Patch Pair Extraction. To further reduce false positives, we constrain the defects and patches as the removed and inserted lines in a bug-fixing commit, respectively. We confine the scope of defects and patches to be within a single method. Of course, these constraints may rule out some genuine defect-patch pairs, e.g., patches only removing old lines or inserting new lines. Our current bug-fix model balances between the completeness and accuracy, attempting to be simple but applicable to the most common cases. We randomly sampled 90 bug fixes from the benchmark set and confirmed that 68 (76%) of them contain both removed and inserted lines, which fall into our simplified bug model. We found in practice that the compromises made in the recall (24%) help significantly improve the precision.

Given a bug-fixing commit and the source code of the project, the workflow of extracting defect-patch pairs is the following.

(1) Check out the last-updated snapshot before the bug-fixing commit and record inserted and removed lines in the commit for later parsing.
(2) Extract inserted lines in the commit, which are patch snippets.

(3) Extract removed lines in the commit, which are defect snippets.
(4) Associate each defect snippet with the corresponding patch snippet, forming defect-patch pairs.
(5) Filter out the lines of defect snippets that are not in the same method scope as any patch lines, making defect-patch pairs more cohesive.
(6) Filter out assertion lines, comments, and logging lines from the defect-patch pairs as they are generally irrelevant to bug fixes.

At this point, we have obtained a set of defect-patch pairs, on which we perform the remaining steps of the patch discovery.

4.1.2 *Clustering Defect-Patch Pairs.*

To obtain common defects patterns in the codebase, we group all the extracted defect-patch pairs into a set of clusters.

We use density-based spatial clustering of applications with noise (DBSCAN) [9] as our clustering algorithm. DBSCAN is a well-established clustering method without the need to determine the number of clusters (K) in advance. Given a set of points in some vector space, DBSCAN groups points that are closely packed together (nearby neighbors), marking as outliers those points that lie in low-density regions. We choose DBSCAN instead of other clustering algorithms, such as KNN [8] and K-means, because in our case, the number K could not be known in advance. A well known limitation of the vanilla DBSCAN algorithm is that it is computationally expensive when coping with large-scale dataset [37], since it considers all possible comparisons between every data point. In our application scenario, most of the comparisons are unnecessary, as many code snippets are very loosely related. Therefore, we make three customized improvements to DBSCAN to mitigate this issue.

Utilizing SimHash-KDTree Reducers. The first customization is to use SimHash-KDTree reducers to avoid the comparison of some irrelevant data points. SimHash [33] is an algorithm that generates a low-dimensional vector signature for any high-dimensional vector. The generated low-dimensional vector has the same expressiveness as the original high-dimensional vector, but is shorter in length, thus enables faster comparison. During preprocessing, we use the SimHash algorithm to map the contents of code snippets to 16-bit hashcode sequences. Then we run the KDTree algorithm [4] to group all the sequences that have a Hamming distance less than 4 into the same reducer. During the clustering, we only compute similarity of code snippets in the same reducer.

Exploiting API Sequence Information. The second improvement is to exploit the information from API call sequences to avoid irrelevant comparisons. The intuition behind this is that if two code snippets contain two completely different API call sequences, then they are obviously not belonging to the same patch pattern, thus should not be compared during the clustering. Therefore, we parse the ASTs of defect-patch pairs and extract API call sequences for each code snippet in advance. During clustering, even if two code snippets are in the same reducer, as long as their API call sequences do not match, we skip the comparison of them. We name this filtering process as APISEQ.

Normalizing Code Snippets. We also improve the clustering accuracy of DBSCAN by normalizing code snippets. Some common coding patterns have multiple equivalent ways of expression. These ways, although semantically equivalent, are syntactically different,

Algorithm 1: Similarity computation of defect-patch pairs.

```
input  : $\langle d_1, p_1 \rangle$, $\langle d_2, p_2 \rangle$ – two defect-patch pairs; $R$ – the set of
         reducers obtained from SimHash-KDTree; $w$ – the
         weight of Jaccard similarity coefficient;
output : $Sim$ – the similarity score of the two input defect-patch
         pairs;
1  if $\nexists r \in R$ s.t. $\langle d_1, p_1 \rangle \in r$ and $\langle d_2, p_2 \rangle \in r$ then
2      return 0;
3  $seq_1 \leftarrow$ ExtractAPISeq($\langle d_1, p_1 \rangle$);
4  $seq_2 \leftarrow$ ExtractAPISeq($\langle d_2, p_2 \rangle$);
5  if $seq_1 \cap seq_2 = \emptyset$ then
6      return 0;
7  $Sim_d \leftarrow$ ComputeSimilarityScore($d_1, d_2, w$);
8  $Sim_p \leftarrow$ ComputeSimilarityScore($p_1, p_2, w$);
9  $Sim \leftarrow (Sim_d + Sim_p) \div 2$;
10 return $Sim$;
11
12 Function ComputeSimilarityScore($s_1, s_2, w$):
13     return
         $w \times$ ComputeJac($s_1, s_2$) $+ (1 - w) \times$ ComputeLev($s_1, s_2$);
```

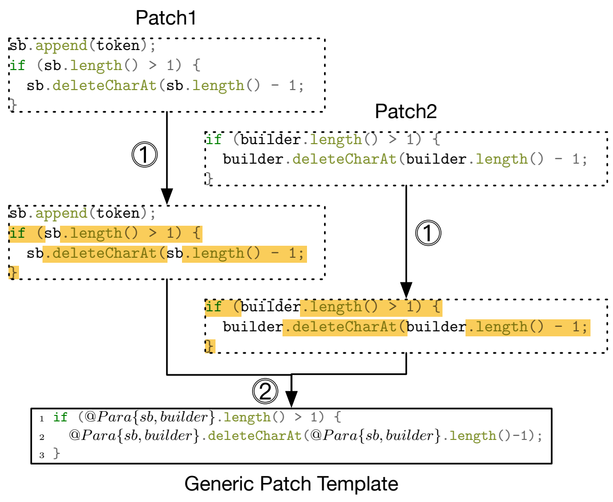
which decreases the accuracy of clustering. For instance, a string value could be expressed as either a single string literal or the concatenation of several string literals. When we compare code snippets, we want to consider these equivalent expressions as the same. Therefore, we normalize the code snippets to convert these semantically equivalent snippets into syntactically same format. Our normalization rules are as follows: (1) merge consecutive calls of the same API into one API call sequence, concatenated by the dot ("`.`") operator; (2) change "`.append(string)`" to the concatenation of strings; and (3) merge the concatenation of strings into a single string.

DBSCAN computes the distance between two defect-patch pairs based on their similarity. The computation of similarity is described in Algorithm 1. Given two defect-patch pairs, the algorithm first checks whether they are in the same reducer. The reducers are obtained using the customized hashing algorithm (SimHash + KDTree). If they are not in the same reducer, then the similarity computation is skipped (Line 2). Otherwise, the algorithm invokes APISEQ, additional filtering based on the API sequence used, on each defect-patch pair to extract API sequences (Lines 3-4). If there is no intersection between the two API sequences, the algorithm also skips the similarity computation (Line 6).

The remaining defect-patch pairs after the filtering proceed to the similarity computation. The similarity function we used in Algorithm 1 is a weighted sum of two similarity metrics: the Jaccard similarity coefficient [39] and the Levenshtein distance [25] (Lines 7-9). We choose these two similarity metrics because the Jaccard similarity coefficient is able to capture the token overlapping rate while the Levenshtein distance has the ability to capture the ordering information of the token list.

4.1.3 *Collecting Generic Patch Templates.*

After the clustering step finishes, for each pattern cluster, PRECFIX tries to extract a generic patch template, which summarizes the common pattern of defect-patch pairs in that cluster. Each template encodes a pattern of defect and its corresponding patch.



```
sb.append(token);
if (sb.length() > 1) {
  sb.deleteCharAt(sb.length() - 1);
}
```

```
if (builder.length() > 1) {
  builder.deleteCharAt(builder.length() - 1);
}
```

```
if (@Para{sb, builder}.length() > 1) {
  @Para{sb, builder}.deleteCharAt(@Para{sb, builder}.length()-1);
}
```

Figure 4: The process of extracting templates.

The goal of template generalization is to make the patch generally applicable in different contexts and understandable to developers. Although all the defect-patch pairs from the same cluster are similar, they differ from each other in some context-specific way, e.g., variable names. We abstract away all context-specific contents and only preserve the generic templates.

The first step of template generalization is to convert code lines into a list of tokens, each token is either a *symbol* or a *parameter*. Symbols are context-independent, they together form the common patterns among all the defect-patch pairs in a cluster; parameters are context-specific that represent the abstraction of variable names. When a template is applied at a specific context, its parameters are supposed to be replaced by the actual variable names.

The second step is to extract templates from the token lists. We use the recursive longest common substring (RLCS) algorithm [15] to perform matching between token lists. RLCS recursively calls longest common substring matching until all the tokens are matched. The highlighted parts in Fig. 4 are the matched tokens after the RLCS process (Step 1). We adjust the vanilla RLCS so that different parameter names are not matched while the other symbols are. Finally, we collect all the parameter names that are matched in the same cluster to a list and store them in our self-defined parameter format. Step 2 in Fig. 4 shows that different parameter names are recognized and collected, and stored in the standard format: @*Para*{parameter name list}.

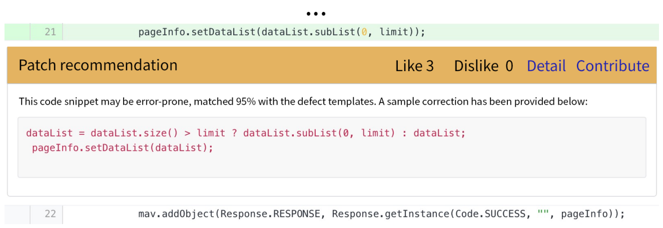
Finally, for each defect-patch pair, we change the parameter names in the patch snippet to match the parameter names in the corresponding defect snippet. After this step, PRECFIX constructs a patch template database, which contains all the extracted templates.

4.2 Patch Recommendation

The online patch recommendation component is triggered whenever developers commit new code changes. During code review, PRECFIX matches each of developers' newly committed code snippet with the templates database. If a code snippet matches with a template, PRECFIX replaces the generic parameter placeholders in the template with the actual parameter names used in the specific contexts. If multiple templates are matched, they are ranked based on the frequency of the corresponding defect-patch pairs before clustering. By default, PRECFIX recommends the most popular patch candidate. The patch candidate is then sent to the code submitters and reviewers for feedback. We collect feedback from developers to validate the patch candidates and improve the template database of PRECFIX. More specifically, the defects and the corresponding recommended patches are presented to the developers. As developers inspect each recommended patch, we also collect their reactions on the patch, i.e., *accept* or *reject*. If patches from a certain template are frequently rejected, we will manually inspect the template and decide whether to remove it.

During the patch validation process, in addition to feedback on the recommended patches, PRECFIX also accepts patch templates created by developers and is able to integrate them into PRECFIX's template database. We encourage developers to make custom adjustments to the existing patches or build their own patch templates based on their experience. We believe this contribution mechanism can enrich the template database in the long run.

Fig. 5 shows the PRECFIX user interface for patch recommendation. Developers interact with it during code reviews. Fig. 5(a) shows the inline view of an identified defect and the corresponding patch recommended. Fig. 5(b) shows the detailed view of the defect-patch pairs (left) and the developer template suggestion form (right). The left side shows the list of defect-patch pairs from which the recommended template was extracted. The details of each defect-patch pair can be expanded, viewed, and voted for or against. Specifically, the two records shown in Fig. 5(b) are from the files "`.../Rec***dler.java`" and "`.../Trip***Impl.java`", respectively. The right side is a form allowing developers to devise their own patches and submit to the database.



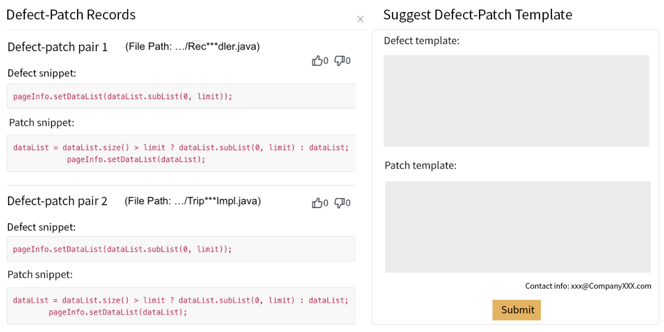
```
21        pageInfo.setDataList(dataList.subList(0, limit));
```

This code snippet may be error-prone, matched 95% with the defect templates. A sample correction has been provided below:

```
dataList = dataList.size() > limit ? dataList.subList(0, limit) : dataList;
 pageInfo.setDataList(dataList);
```

```
22        mav.addObject(Response.RESPONSE, Response.getInstance(Code.SUCCESS, "", pageInfo));
```

(a) Inline view of recommended patch.



Defect-patch pair 1 (File Path: .../Rec***dler.java)

Defect snippet:

```
pageInfo.setDataList(dataList.subList(0, limit));
```

Patch snippet:

```
dataList = dataList.size() > limit ? dataList.subList(0, limit) : dataList;
        pageInfo.setDataList(dataList);
```

Defect-patch pair 2 (File Path: .../Trip***Impl.java)

Defect snippet:

```
pageInfo.setDataList(dataList.subList(0, limit));
```

Patch snippet:

```
dataList = dataList.size() > limit ? dataList.subList(0, limit) : dataList;
      pageInfo.setDataList(dataList);
```

Defect template:

Patch template:

Contact info: xxx@CompanyXXX.com

(b) Detailed view of defect-patch pairs and the template suggestion form.

Figure 5: PRECFIX user interface for patch recommendation.

5 IMPLEMENTATION AND EVALUATION

In this section, we describe the implementation details of Precfix and evaluate its effectiveness and efficiency on our benchmark set.

Precfix is implemented on top of the cloud-based data processing platform, MaxCompute [14], developed by Alibaba. The commit history data is preprocessed and stored in data tables on the Alibaba Cloud [13] storage first. The defect-patch pair extraction is implemented as a set of SQL scripts and user-defined functions (about 900 LOC). The clustering of defect-patch pairs is highly parallelized (taking about 1 KLOC Java code) and handled by the MapReduce engine of MaxCompute. The online patch recommendation component is integrated with the internal code review process and is triggered whenever a new commit is submitted.

The goal of our empirical evaluation is to answer the following research questions. **RQ1**: How effective is Precfix in locating defects and discovering patches? **RQ2**: How efficient is Precfix in recommending patches? **RQ3**: How much do our improvements of DBSCAN increase the efficiency of clustering? **RQ4**: What kind of patches does Precfix recommend? **RQ5**: What are the users' opinions on the usability of Precfix?

We run the offline patch discovery component of Precfix on the randomly extracted sample dataset described in Sec 3. The sample dataset includes 10K projects, 15M commits, and 30M files. On this dataset, Precfix extracted 3K bug fix templates, forming the template database. During the clustering stage, 4,098 MB memory was allocated for each reducer.

5.1 Effectiveness

We use two metrics for measuring the effectiveness of Precfix: False Positive Rate (FPR) and Extractability Score (ES).

FPR measures the quality of patches, which is calculated as:

$$FPR = (N_{total} - N_{correct})/N_{total} \quad (1)$$

where N_{total} denotes the total number of bug fix templates extracted by Precfix, $N_{correct}$ denotes the number of correct templates extracted by Precfix. A low FPR indicates high quality of patches.

ES score measures the effectiveness of Precfix in extracting templates from clusters, is calculated as:

$$ES = N_{templates}/N_{clusters} \quad (2)$$

where $N_{clusters}$ denotes the number of clusters obtained in the clustering step of patch discovery, and $N_{templates}$ denotes the number of bug fix templates extracted from the clusters. A high ES score indicates high effectiveness in extracting templates. In the ideal case, Precfix is supposed to extract one template from every cluster. However, in practice, there could be some clusters of which the defect-patch pairs are too complex to summarize. Thus, Precfix sometimes could not achieve 100% ES score.

We randomly sampled 170 templates extracted by Precfix in our experiment and manually inspected them. To reduce potential bias in our inspection, we applied the following rule of thumb in determining the correctness of a template: (1) if a template is too generic, e.g., modification to a "for-loop" bound, it is often the result of unintended clustering of similarly structured but semantically unrelated code snippets; and (2) if a template is too specific to a certain context, such as replacing an "if" condition using some project-specific APIs, it cannot be easily migrated to other application contexts. We consider the templates falling into either category incorrect. According to these criteria, the FPR obtained from the inspection is 22%. Note that in our application scenario, this number is supposed to be gradually reduced as developers provide feedback on the discovered patches and contribute their own patches.

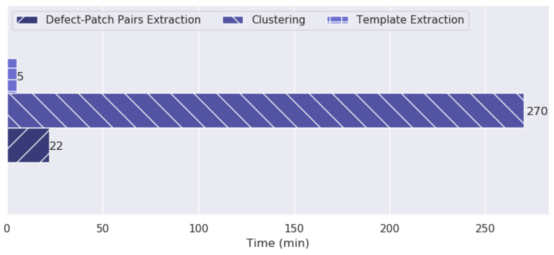


Figure 6: Execution time of each phase of patch discovery.

To have a direct comparison with the SZZ algorithm, we ran our implementation of SZZ on the same sample dataset (10K projects) and compared with Precfix. Since SZZ and Precfix are of different objectives, and end-to-end comparison is not possible. Instead, we compared the phase of locating defects which is common for both techniques. As introduced in Sec. 3.2, SZZ locates defects in two steps: finding bug-fix commit with keyword matching and identifying the bug-inducing commits with back-tracing. We also manually examined 170 bug-fix commits identified by SZZ, and only reported incorrect ones which are easy to confirm. Examples of such false positives include comments and log file modifications, import statement insertions and deletions, style changes, error message modifications, etc. The FPR of SZZ is 63% (107 out of 170) according to the inspection. In contrast, Precfix locates defects with the extracted templates. Most of the aforementioned incorrect bug-fixes are caused by keyword matching, which could be effectively reduced in the clustering stage. As a result, Precfix achieves better precision in identifying defects (FPR 22%).

We also randomly sample and inspect 100 clusters obtained by Precfix and the templates extracted from these clusters. The inspection confirms that Precfix successfully extract templates from 93 clusters. In other words, the ES score obtained by Precfix on the sample dataset is 93%.

Answer to RQ 1

Overall, Precfix achieves a false positive rate of 22% in patch discovery and an extractability score of 93%.

5.2 Efficiency

As invoking patch recommendation on any code snippet only take several milliseconds, we mainly evaluate the efficiency of Precfix for patch discovery. We measure the time taken by each step on the sample dataset: extracting defect-patch pairs, clustering defect-patch pairs, and extracting templates.

Fig. 6 shows the time consumed in each step. In summary, the time consumed in the defect-patch pairs extraction step takes 22 min; the clustering is the most time-consuming step, taking 270 min; the template extraction step takes 5 min.

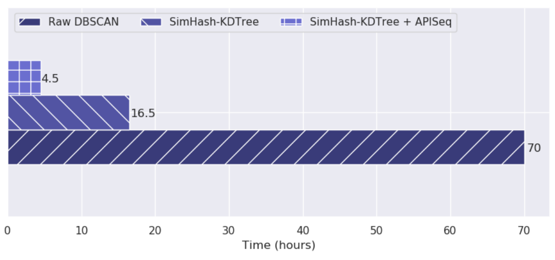


Figure 7: Execution time of different clustering approaches.

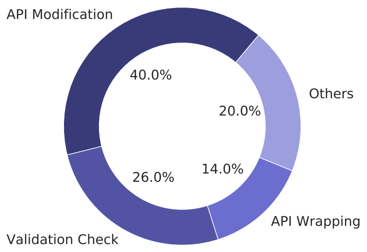


Figure 8: Distribution of patches in each category.

Answer to RQ 2

Of the steps of patch discovery, extracting defect-patch pairs, clustering defect-patch pairs, and extracting templates consumes 22 min, 270 min, and 5 min, respectively.

To answer RQ3, we investigated how much our improvements of the vanilla DBSCAN improve the efficiency. We compared the execution time of our default SimHash-KDTree + APISeq DBSCAN clustering, and the raw DBSCAN clustering, SimHash-KDTree + DBSCAN clustering. The result of the comparison is shown in Fig. 7.

As is shown, the execution time of the default SimHash-KDTree + APISeq DBSCAN is 4.5 hours, which saves 95% of the time compared with the raw DBSCAN. The execution time of SimHash-KDTree DBSCAN is 16.5 hours, which saves 75% compared with the raw DBSCAN. This result confirms that both SimHash-KDTree and APISeq improves the efficiency of clustering.

Answer to RQ 3

Our two improvements of the DBSCAN, SimHash-KDTree and APISeq, improves the efficiency of clustering by 75% and 20%, respectively.

5.3 Patch Categories

We randomly sampled 50 templates from the template database and manually inspected each one. Based on the inspection results, we categorized these templates into four categories: API modification, validation check, API wrapping, and others. Fig. 8 shows the distribution of templates of each category. There are 20 templates assigned to the API modification category, accounting for 40% of the total number; 13 templates (26%) are assigned to the validation check category; 7 templates (14%) are assigned to the API wrapping category; 10 templates (20%) are assigned to the "others" category.

API Modification. Templates in this category fix the defect by modifying the call of an API, including: (1) update the caller of an API, (2) add, update, or delete the parameters of an API, and (3) update the return type of an API. Fig. 9 is a template of API modification. It deletes a parameter of `buildReqParams()` method, and adds two new parameters.

```
  multipleSource.setParams(
    MultiSourceConvertUtil.buildReqParams(
-       itemSku.getItemId().getValue(),
+              itemSku.getConfigId(),
               itemSku.getSkuId(),
               itemSku.getSellerId(),
+              itemSku.getGpuId()},
               multipleSource.getPageSize(),
               multipleSource.getPageIndex()));
```

Figure 9: An example of API modification.

Validation Check. Templates in this category add or modify validation checks. Such checks include: (1) null pointers, i.e., adding "if(xxx==null) return;" above defect snippets, (2) boundary conditions, i.e., adding or modifying conditions in condition expression above such as "if" or "while" expressions, and (3) access permissions, i.e., replacing the variable assignment with a ternary operation such as "a = b == null ? 0 : 1". Fig. 10 is a template of validation check. It adds a condition to check whether `accountStatus` is null and its length is greater than zero.

```
+ if (accountStatus != null && accountStatus.length > 0) {
    query.addCondition(new In(STR, accountStatus));
+ }
```

Figure 10: An example of validation check.

API Wrapping. Templates in this category fix the defect by wrapping logic-independent code snippets to form an API (mostly a utility API), which improves code quality and facilitates software maintenance, preventing from introducing defects due to over complex code.

Fig. 11 is a template of API wrapping. It packages the original code snippets into `getUrl()` and `requestAndCheckThenFillResult()`, forming two new APIs.

```
- String ip = host.getIp();
- String url ="http://".concat(ip).concat(flowHost);
- HttpResponse<String> response = httpRequest.asString();
- ErrorUtil.checkError(httpRequest, response, TREND, start);
- bodys.add(response.getBody());
+ String url = UrlUtil.getUrl(headHost, flowHost);
+ UrlUtil.requestAndCheckThenFillResult(httpRequest, bodys, TREND, sta
```

Figure 11: An example of API wrapping.

Other. Templates in this category fix the defect in some special ways, which usually depend on the context. For example, Fig. 12 shows a template fixing a defect by changing the name of a class.

```
- String cur = CurrencyConvertUtil.getAvailableCurrency(currencyCode);
+ String cur = MoneyConvertUtil.getAvailableCurrency(currencyCode);
```

Figure 12: An example of other fixes.

Answer to RQ 4

89.8% of the patches discovered by Precfix are in one of the three categories: API modification, validation check, and API wrapping.

5.4 User Study

With the template database, we ran patch recommendation of Precfix on the whole internal codebase[1], from which Precfix identified 30K defects from 7K projects in total and provided corresponding patches. Precfix has been deployed in Alibaba for about one year so far. Every week, it recommends about 400 patches to developers on average, and receives about two to three false positive reports.

To answer RQ5, we randomly sampled 50 defects identified by Precfix, and sent the recommended patches to the submitters of the defective code snippets for their opinions. In the end, 12 developers responded to our requests, all of whom are maintainers of the corresponding repositories. We interviewed these developers, presented them the defects and patches with two questions:

Q1: "*Is the patch suggested by Precfix valid? If so, would you like to apply the patch?*"
Q2: "*Would you like to use Precfix during code review?*"

For Q1, 10 of 12 developers answered "yes". They confirmed that the defect code found by Precfix are true positive, and agreed to fix them using the suggested patch. For the other two developers who rejected the patches, one developer reported that although the recommended patch provided by Precfix works for local testing configuration but is not valid for global configuration, as it induces problems when running together with other components. She commented that the patch was not appropriate for all possible contexts. Another negative case is that the developer has already used a null pointer check inside the called API, so she commented it is not necessary to have another null pointer checking outside the call as Precfix suggested. Overall, these comments from developers confirm the value of Precfix. In special cases, developer could judge the validity of the recommended patch and decide whether to adopt it.

For Q2, all of the 12 developers answered "yes". They agreed on that Precfix would improve the effectiveness of code review, and would like to see Precfix adopted on their projects. As an example, one interviewed developer said "I often struggle finding defects when reviewing code. It would be really helpful when having a patch recommendation technique to assist me."

Answer to RQ 5

The majority (10/12) of the interviewed developers acknowledged the value of the patches provided by Precfix, and all of them would like to see Precfix adopted in practice.

5.5 Discussion

In the experiment, we have tried multiple combinations of the weight of Jaccard similarity coefficient and Levenshtein distance. We noticed that although Levenshtein distance captures token ordering information, it is highly sensitive to the change of ordering. As a result, we set 0.9 to the weight of Jaccard similarity coefficient, while 0.1 to the weight of Levenshtein distance, as heuristics.

5.6 Threats to Validity

Our experiments are subject to common threats to validity.

[1]Due to company policy, we could not report the total number of projects.

External. Subjects used in the sample dataset may not be representative for the entire internal codebase. We manually checked the selected subjects and confirmed that they cover more than 100 diverse application domains, which could help mitigate this threat. We only consider Java programs in our codebase as the current implementation of Precfix only handle Java, but the idea of Precfix is essentially independent of languages.

Internal. We rely on manual inspection to determine the correctness of bug-fix templates and the recommended patches, which may be subjective. To mitigate this threat, we established common criteria before the inspection, as described in Sec. 5.1, and then two authors inspected each case independently. A patch is determined to be incorrect only if they reached an agreement on it.

6 CONCLUSION

In this paper, we present Precfix, a patch recommendation technique designed for large-scale industrial codebase. For patch discovery, Precfix extracts defect-patch pairs from version controlled histories, clusters these pairs with common patterns, and extracts bug fix templates from clusters. For patch recommendation, Precfix relies on the feedback from developers during code review. Moreover, Precfix integrates custom patches created by developers to improve its template database. We run Precfix's patch generation on a subset (10K projects) of the internal codebase at Alibaba and extract 3K patches. Our manual inspection confirms that Precfix has low false positive rate (22%) while achieves a high extractability score (93%). In addition, our user study on the usability of Precfix's patch recommendation functionality shows that the majority (10/12) of the interviewed developers appreciated Precfix and would like to see it widely adopted.

The current implementation of Precfix still has some incompleteness in terms of the type of defects and patches handled. In the future, we would like to lift these restrictions while maintaining the high accuracy of patch recommendation. We would also like to apply machine learning-based approaches to automate the patch template improvement as much as possible.

ACKNOWLEDGMENTS

This article is partially supported by the Singapore Ministry of Education Academic Research Fund Tier 1 (No. 2018-T1-002-069), National Natural Science Foundation of China (No. 61772200), and Alibaba Group.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2007. On the Accuracy of Spectrum-Based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques-MUTATION*. 89–98.
[2] Hiralal Agrawal, Joseph R Horgan, Saul London, and W Eric Wong. 1995. Fault Localization Using Execution Slices and Dataflow Tests. In *International Symposium on Software Reliability Engineering*. 143–151.
[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to Fix Bugs Automatically. arXiv:1902.06111 [cs.SE]
[4] Jon Louis Bentley. 1975. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM* 18, 9 (1975), 509–517.
[5] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. 2013. *Reversible Debugging Software*. Technical Report. Judge Business School, University of Cambridge.
[6] Zimin Chen, Steve Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2018. SequenceR: Sequence-to-Sequence Learning for End-to-End Program Repair. arXiv:1901.01808 [cs.SE]

[7] Valentin Dallmeier, Andreas Zeller, and Bertrand Meyer. 2009. Generating Fixes From Object Behavior Anomalies. In *International Conference on Automated Software Engineering*. 550–554.

[8] Belur V Dasarathy. 1991. Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques. *IEEE Computer Society Tutorial* (1991).

[9] Martin Ester, Hans-Peter Kriegel, Jörg Sander, Xiaowei Xu, et al. 1996. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *International Conference on Knowledge Discovery and Data Mining*. 226–231.

[10] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe Memory-Leak Fixing For C Programs. In *International Conference on Software Engineering*. 459–470.

[11] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2019. Automatic Software Repair: A Survey. *Transactions on Software Engineering* 45, 1 (2019), 34–67.

[12] Git 2019. git-blame: Show What Revision and Author Last Modified Each Line of a File. https://git-scm.com/docs/git-blame.

[13] Alibaba Group. 2019. Alibaba Cloud. https://www.alibabacloud.com.

[14] Alibaba Group. 2019. MaxCompute: Conduct Large-Scale Data Warehousing with MaxCompute. https://www.alibabacloud.com/product/maxcompute.

[15] Daniel S. Hirschberg. 1977. Algorithms for the Longest Common Subsequence Problem. *Journal of the ACM* 24, 4 (1977), 664–675.

[16] Dennis Jeffrey, Min Feng, Neelam Gupta, and Rajiv Gupta. 2009. BugFix: A Learning-Based Tool to Assist Developers in Fixing Bugs. In *International Conference on Program Comprehension*. 70–79.

[17] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *International Symposium on Software Testing and Analysis*. 298–309.

[18] James A Jones and Mary Jean Harrold. 2005. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *International Conference on Automated Software Engineering*. 273–282.

[19] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs. In *International Symposium on Software Testing and Analysis*. 437–440.

[20] Shalini Kaleeswaran, Varun Tulsian, Aditya Kanade, and Alessandro Orso. 2014. Minthint: Automated Synthesis of Repair Hints. In *International Conference on Software Engineering*. 266–276.

[21] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing Programs with Semantic Code Search. In *International Conference on Automated Software Engineering*. 295–306.

[22] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic Patch Generation Learned from Human-Written Patches. In *International Conference on Software Engineering*. 802–811.

[23] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. 2007. Predicting Faults from Cached History. In *International Conference on Software Engineering*. 489–498.

[24] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: Bug Report Driven Program Repair. In *Symposium on Foundations of Software Engineering*. 314–325.

[25] Vladimir I Levenshtein. 1966. Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. *Soviet physics doklady* 10, 8 (1966), 707–710.

[26] Jian Li, Pinjia He, Jieming Zhu, and Michael R. Lyu. 2017. Software Defect Prediction via Convolutional Neural Network. In *International Conference on Software Quality, Reliability and Security*. 318–328.

[27] Yi Li, Chenguang Zhu, Milos Gligoric, Julia Rubin, and Marsha Chechik. 2019. Precise Semantic History Slicing Through Dynamic Delta Refinement. *Automated Software Engineering* 26, 4 (Dec 2019), 757–793.

[28] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. FHistorian: Locating Features in Version Histories. In *Proceedings of the 21st International Systems and Software Product Line Conference - Volume A* (Sevilla, Spain). ACM, New York, NY, USA, 49–58.

[29] Yi Li, Chenguang Zhu, Julia Rubin, and Marsha Chechik. 2017. Semantic Slicing of Software Version Histories. *Transactions on Software Engineering* 44, 2 (2017), 182–201.

[30] Haopeng Liu, Yuxi Chen, and Shan Lu. 2016. Understanding and Generating High Quality Patches for Concurrency Bugs. In *Symposium on Foundations of Software Engineering*. 715–726.

[31] Peng Liu, Omer Tripp, and Charles Zhang. 2014. Grail: Context-Aware Fixing of Concurrency Bugs. In *Symposium on Foundations of Software Engineering*. 318–329.

[32] Fan Long and Martin Rinard. 2016. Automatic Patch Generation by Learning Correct Code. In *Symposium on Principles of Programming Languages*. 298–312.

[33] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting Near-Duplicates for Web Crawling. In *International Conference on World Wide Web*. 141–150.

[34] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *ACM Computing Surveys* 51, 1 (2018), 17:1–17:24.

[35] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. 2014. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *International Conference on Software Testing, Verification and Validation*. 153–162.

[36] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: Combining Code Metrics with Clone Detection for Just-In-Time Fault Prevention and Resolution in Large Industrial Projects. In *International Conference on Mining Software Repositories*. 153–164.

[37] Helmut Neukirchen. 2016. *Survey and Performance Evaluation of DBSCAN Spatial Clustering Implementations for Big Data and High-Performance Computing Paradigms*. Technical Report. Engineering Research Institute, University of Iceland.

[38] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. Semfix: Program Repair via Semantic Analysis. In *International Conference on Software Engineering*. 772–781.

[39] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. 2013. Using of Jaccard Coefficient for Keywords Similarity. In *International Multiconference of Engineers and Computer Scientists*. 380–384.

[40] Mike Papadakis and Yves Le Traon. 2015. Metallaxis-FL: Mutation-Based Fault Localization. *Software Testing, Verification and Reliability* 25, 5-7 (2015), 605–628.

[41] Spencer Pearson, Jose Campos, Rene Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *International Conference on Software Engineering*. 609–620.

[42] Vidyasagar Potdar and Elizabeth Chang. 2004. Open Source and Closed Source Software Development Methodologies. In *International Conference on Software Engineering*. 105–109.

[43] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An Analysis of Patch Plausibility and Correctness for Generate-And-Validate Patch Generation Systems. In *International Symposium on Software Testing and Analysis*. 24–36.

[44] Foyzur Rahman, Daryl Posnett, Abram Hindle, Earl Barr, and Premkumar Devanbu. 2011. Bugcache for Inspections: Hit or Miss?. In *Symposium on Foundations of Software Engineering*. 322–331.

[45] Manos Renieres and Steven P Reiss. 2003. Fault Localization with Nearest Neighbor Queries. In *International Conference on Automated Software Engineering*. 30–39.

[46] Hesam Samimi, Max Schafer, Shay Artzi, Todd Millstein, Frank Tip, and Laurie Hendren. 2012. Automated Repair of HTML Generation Errors in PHP Applications using String Constraint Solving. In *International Conference on Software Engineering*. 277–287.

[47] Guido Schryen. 2009. A Comprehensive and Comparative Analysis of the Patching Behavior of Open Source and Closed Source Software Vendors. In *International Conference on IT Security Incident Management and IT Forensics*. 153–168.

[48] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. 2005. When do Changes Induce Fixes?. In *International Conference on Mining Software Repositories*. 1–5.

[49] Undo Software. 2014. Increasing Software Development Productivity with Reversible Debugging.

[50] Yida Tao, Jindae Kim, Sunghun Kim, and Chang Xu. 2014. Automatically Generated Patches as Debugging Aids: A Human Study. In *Symposium on Foundations of Software Engineering*. 64–74.

[51] Shaowei Wang, David Lo, Lingxiao Jiang, Lucia, and Hoong Chuin Lau. 2011. Search-Based Fault Localization. In *International Conference on Automated Software Engineering*. 556–559.

[52] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically Finding Patches Using Genetic Programming. In *International Conference on Software Engineering*. 364–374.

[53] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *International Conference on Software Maintenance*. 181–190.

[54] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A Survey on Software Fault Localization. *Transactions on Software Engineering* 42, 8 (2016), 707–740.

[55] Rongxin Wu, Hongyu Zhang, Shing-Chi Cheung, and Sunghun Kim. 2014. CrashLocator: Locating Crashing Faults Based on Crash Stacks. In *International Symposium on Software Testing and Analysis*. 204–214.

[56] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2019. A Comprehensive Study of Automatic Program Repair on the Quixbugs Benchmark. In *2019 IEEE 1st International Workshop on Intelligent Bug Fixing (IBF)*. IEEE, 1–10.

[57] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A Correlation Study between Automated Program Repair and Test-Suite Metrics. In *International Conference on Software Engineering*. 24.

[58] Xiangyu Zhang, Neelam Gupta, and Rajiv Gupta. 2006. Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*. ACM, 272–281.

[59] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where Should the Bugs be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports. In *International Conference on Software Engineering*. 14–24.

[60] Daming Zou, Jingjing Liang, Yingfei Xiong, Michael D Ernst, and Lu Zhang. 2019. An Empirical Study of Fault Localization Families and Their Combinations. *Transactions on Software Engineering* (2019), To appear.