**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:**


**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...
We also allow developers in the loop to provide feedback on the patches and their usefulness. In this section, we compare and evaluate Precfix with existing techniques.

Contributions. We make the following contributions in this paper.

- **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 fault 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 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 t-b1ame [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
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 **Prefix**. **Prefix** 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-KDTreeReducers. 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, 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, PrefixX 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.

---

**Algorithm 1:** Similarity computation of defect-patch pairs.

**Input:** \((d_1, p_1), (d_2, p_2)\) – 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;

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{if} \(\exists r \in R \text{ s.t. } (d_1, p_1) \in r \text{ and } (d_2, p_2) \in r\) \textbf{then}
\hspace{1em} \State \textbf{return} 0;
\State \(seq_1 \leftarrow \text{ExtractAPISeq}(d_1, p_1)\);
\State \(seq_2 \leftarrow \text{ExtractAPISeq}(d_2, p_2)\);
\State \textbf{if} \(seq_1 \cap seq_2 = \emptyset\) \textbf{then}
\hspace{1em} \State \textbf{return} 0;
\State \(\text{Sim}_d \leftarrow \text{ComputeSimilarityScore}(d_1, d_2, w)\);
\State \(\text{Sim}_p \leftarrow \text{ComputeSimilarityScore}(p_1, p_2, w)\);
\State \(\text{Sim} \leftarrow (\text{Sim}_d + \text{Sim}_p) / 2\);
\State \textbf{return} \(\text{Sim}\);
\end{algorithmic}
\end{algorithm}

---
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 the 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, Precifix 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, Precifix matches each of developers’ newly committed code snippet with the templates database. If a code snippet matches with a template, Precifix replaces the generic parameter placeholders in the template with the actual parameter names used in the specific context. If multiple templates are matched, they are ranked based on the frequency of the corresponding defect-patch pairs before clustering. By default, Precifix 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 Precifix. 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, Precifix also accepts patch templates created by developers and is able to integrate them into Precifix’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 Precifix 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 ".../Recc***der.java" and ".../Trip***Impl.java", respectively. The right side is a form allowing developers to devise their own patches and submit to the database.

---

**Figure 4:** The process of extracting templates.

**Figure 5:** Precifix user interface for patch recommendation.
5 IMPLEMENTATION AND EVALUATION

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

\textit{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. \textbf{RQ1}: How effective is \textit{Precfix} in locating defects and discovering patches? \textbf{RQ2}: How efficient is \textit{Precfix} in recommending patches? \textbf{RQ3}: How much do our improvements of DBSCAN increase the efficiency of clustering? \textbf{RQ4}: What kind of patches does \textit{Precfix} recommend? \textbf{RQ5}: What are the users’ opinions on the usability of \textit{Precfix}?

We run the offline patch discovery component of \textit{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, \textit{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 \textit{Precfix}: False Positive Rate (FPR) and Extractability Score (ES).

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

\[
FPR = \frac{N_{\text{total}} - N_{\text{correct}}}{N_{\text{total}}} \tag{1}
\]

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

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

\[
ES = \frac{N_{\text{templates}}}{N_{\text{clusters}}} \tag{2}
\]

where \(N_{\text{clusters}}\) denotes the number of clusters obtained in the clustering step of patch discovery, and \(N_{\text{templates}}\) 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, \textit{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, \textit{Precfix} sometimes could not achieve 100% ES score.

We randomly sampled 170 templates extracted by \textit{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

5.2 Efficiency

As invoking patch recommendation on any code snippet only take several milliseconds, we mainly evaluate the efficiency of \textit{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.
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.

### 5.3 Patch Categories

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.

### 5.4 Other Fixes

#### 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.

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

### Answer to RQ 4

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

---

**Figure 7:** Execution time of different clustering approaches.

**Figure 8:** Distribution of patches in each category.

**Figure 9:** An example of API modification.

**Figure 10:** An example of validation check.

**Figure 11:** An example of API wrapping.

**Figure 12:** An example of other fixes.
5.4 User Study

With the template database, we ran patch recommendation of Precfix on the whole internal codebase, 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 necessary 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 order of changes. 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.

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

REFERENCES


