ABSTRACT

Issue tracking systems are commonly used in modern software development for collecting feedback from users and developers. An ultimate automatic target of software maintenance is then the systematicatization of patch generation for user-reported bugs. Although this ambition is aligned with the momentum of automated program repair, the literature has, so far, mostly focused on generate-and-validate setups where fault localization and patch generation are driven by a well-defined test suite. On the one hand, however, the common (yet strong) assumption on the existence of relevant test cases does not hold in practice for most development settings: many bugs are reported without the available test suite being able to reveal them. On the other hand, for many projects, the number of bug reports generally outstrips the resources available to triage them. Towards increasing the adoption of patch generation tools by practitioners, we investigate a new repair pipeline, iFixR, driven by bug reports: (1) bug reports are fed to an IR-based fault localizer; (2) patches are generated from fix patterns and validated via regression testing; (3) a prioritized list of generated patches is proposed to developers. We evaluate iFixR on the Defects4J dataset, which we enriched (i.e., faults are linked to bug reports) and carefully-reorganized (i.e., the timeline of test-cases is naturally split). iFixR generates genuine/plausible patches for 21/44 Defects4J faults with its IR-based fault localizer. iFixR accurately places a genuine/plausible patch among its top-5 recommendation for 8/13 of these faults (without using future test cases in generation-and-validation).

CCS CONCEPTS

- Software and its engineering → Software verification and validation; Software testing and debugging.

KEYWORDS

Information retrieval, fault localization, automatic patch generation.

1 INTRODUCTION

Automated program repair (APR) has gained incredible momentum since the seminal work of GenProg [83], various approaches [13, 14, 19, 20, 26, 27, 33, 34, 39, 45, 48, 49, 52, 53, 55, 62, 65, 83, 84, 92, 93] have been proposed in the literature aiming at reducing manual debugging efforts through automatically generating patches. Beyond fixing syntactic errors, i.e., cases where the code violates some programming language specifications [17], the current challenges lie in fixing semantic bugs, i.e., cases where implementation of program behavior deviates from developer’s intention [23, 61].

Ten years ago, the work of Weimer et al. [83] was explicitly motivated by the fact that, despite significant advances in specification mining (e.g., [40]), formal specifications are rarely available. Thus, test suites represented an affordable approximation to program specifications. Unfortunately, the assumption that test cases are readily available still does not hold in practice [8, 28, 68]. Therefore, while current test-based APR approaches would be suitable in a test-driven development setting [7], their adoption by practitioners faces a simple reality: developers majoritarily (1) write few tests [28], (2) write tests after the source code [8], and (3) write tests to validate that bugs are indeed fixed and will not reoccur [24].

Although APR bots [79] can come in handy in a continuous integration environment, the reality is that bug reports remain the main source of the stream of bugs that developers struggle to handle.
indeed shows (IR)-based fault localization. Eventually, we propose their R2Fix approach carries several caveats: as illustrated in Fig-
tests suites tent an APR pipeline is feasible under the practical constraint of limited
program repair workflow which considers a practical repair setup
cal spectrum-based fault localization with Information Retrieval
gram repair system driven by bug reports, thus we replace classi-
its fix patterns, R2Fix does not need to run tests for verifying that
noted that, given the limited scope of the changes implemented in
engine to generate patches. As the authors admitted, their target
cation to identify the bug category and uses a match and transform
test cases to validate patch candidates, many of these
regression test cases to validate patch candidates, many of these
test-based systems. We propose to carefully examine the actual bug
Defects4J offers a user-friendly way of checking out buggy versions
and the test case is being overlooked by the dataset users. Indeed,
Defects4J is a manual curated dataset widely used in the APR litera-
localization errors. Furthermore, iFixR's prioritization strategy tends to place
more correct/plausible patches on top of the recommendation list.

2 MOTIVATION

We motivate our work by revisiting two essential steps in APR:

(1) During fault localization, relevant program entities are identified
as suspicious locations that must be changed. Commonly, state-
of-the-art APR tools leverage spectrum-based fault localization
(SBFL) [13, 33, 50, 52, 92, 93], which uses execution coverage
information of passing and failing test cases to predict buggy
statements. We dissect the Defects4J dataset to highlight the
practical challenges of fault localization for user-reported bugs.

(2) Once a patch candidate is generated, the patch validation step
ensures that it is actually relevant for repairing the program.
Currently, widespread test-based APR techniques use test suites
as the repair oracle. This however is challenged by the incom-
pleteness of test suites, and may further not be inline with
developer requirements/expectations in the repair process.

2.1 Fault Localization Challenges

Defects4J is a manual curated dataset widely used in the APR litera-
ture [13, 19, 73, 84, 90, 91]. Since Defects4J was not initially built for
APR, the real order of precedence between the bug report, the patch
and the test case is being overlooked by the dataset users. Indeed,
Defects4J offers a user-friendly way of checking out buggy versions
of programs with all relevant test cases for readily benchmarking
test-based systems. We propose to carefully examine the actual bug
fix commits associated with Defects4J bugs and study how the test
suite is evolved. Table 1 provides detailed information.

<table>
<thead>
<tr>
<th>Test case related commits</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Commit does not alter test cases</td>
<td>14</td>
</tr>
<tr>
<td>Commit is inserting new test case(s) and updating previous test case(s)</td>
<td>62</td>
</tr>
<tr>
<td>Commit is updating previous test case(s) (without inserting new test cases)</td>
<td>76</td>
</tr>
<tr>
<td>Commit is inserting new test case(s) (without updating previous test cases)</td>
<td>243</td>
</tr>
</tbody>
</table>

Overall, for 96% (381/395) bugs, the relevant test cases are ac-
tually future data with respect to the bug discovery process. This
finding suggests that, in practice, even the fault localization may
be challenged in the case of user-reported bugs, given the lack of
relevant test cases. The statistics listed in Table 2 indeed shows that
if future test cases are dropped, no test case is failing when
executing buggy program versions for 365 (i.e., 92%) bugs.

Table 2: Failing test cases after removing future test cases.

<table>
<thead>
<tr>
<th>Failing test cases</th>
<th># bugs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failing test cases exist (and no future test cases are committed)</td>
<td>14</td>
</tr>
<tr>
<td>Failing test cases exist (but future test cases update the test scenarios)</td>
<td>9</td>
</tr>
<tr>
<td>Failing test cases exist (but they are fewer when considering future test cases)</td>
<td>4</td>
</tr>
<tr>
<td>Failing test cases exist (but they differ from future test cases which trigger the bug)</td>
<td>3</td>
</tr>
<tr>
<td>No failing test case exists (i.e., only future test cases trigger the bug)</td>
<td>365</td>
</tr>
</tbody>
</table>
In the APR literature, fault localization is generally performed using the GZoltar [12] testing framework and a SBFL formula [88] (e.g., Ochiai [3]). To support our discussions, we attempt to perform fault localization without the future test cases to evaluate the performance gap. Experimental results (see details forward in Table 6 of Section 5) expectedly reveal that the majority of the Defects4J bugs (i.e., 375/395) cannot be localized by SBFL at the time the bug is reported by users.

It is necessary to investigate alternate fault localization approaches that build on bug report information since relevant test cases are often unavailable when users report bugs.

### 2.2 Patch Validation in Practice

The repair community has started to reflect on the acceptability [27, 63] and correctness [76, 91] of the patches generated by APR tools. Notably, various studies [11, 35, 69, 76, 94] have raised concerns about overfitting patches: a typical APR tool that uses a test suite as the correctness criterion can produce a patched program that actually overfits the test-suite (i.e., the patch makes the program pass all test cases but does not actually repair it). Recently, new research directions [89, 97] are being explored in the automation of test case generation for APR to overcome the overfitting issue. Nevertheless, so far they have had minimal positive impact due to the oracle problem [98] in automatic test generation.

At the same time, the software industry takes a more systematic approach for patch validation by developers. For instance, in the open-source community, the Linux development project has integrated a patch generation engine to automate collateral evolutions that are validated by maintainers [50, 66]. In proprietary settings, Facebook has recently reported on their Getafix [75] tool, which automatically suggests fixes to their developers. Similarly, Ubisoft developed Clever [64] to detect risky commits at commit-time using patterns of programming mistakes from the code history.

*Patch recommendation for validation by developers is acceptable in the software development communities. It may thus be worthwhile to focus on tractable techniques for recommending patches in the road to fully automated program repair.*

### 3 THE IFIXR APPROACH

Figure 2 overviews the workflow of the proposed IFIXR approach. Given a defective program, we consider the following issues:

1. **Where is the bug?** We take as input the bug report in natural language submitted by the program user. We rely on the information in this report to localize the bug positions.

2. **How should we change the code?** We apply fix patterns that are recurrently found in real-world bug fixes. Fix patterns are selected following the structure of the abstract syntax tree representing the code entity of the identified suspicious code.

3. **Which patches are valid?** We make no assumptions on the availability of **positive test cases** [83] that encode functionality requirements at the time the bug is discovered. Nevertheless, we leverage existing test cases to ensure, at least, that the patch does not regress the program.

4. **Which patches do we recommend first?** In the absence of a complete test suite, we cannot guarantee that all patches that pass regression tests will fix the bug. We rely on heuristics to re-prioritize the validated patches in order to increase the probability of placing a correct patch on top of the list.

#### 3.1 Input: Bug reports

Issue tracking systems (e.g., Jira) are widely used by software development communities in the open source and commercial realms. Although they can be used by developers to keep track of the bugs that they encounter and the features to be implemented, issue tracking systems allow for user participation as a communication channel for collecting feedback on software executions in production.

Table 3 illustrates a typical bug report when a user of the LANG library code has encountered an issue while using the `NumberUtils` API. A description of erroneous behavior is provided. Occasionally, the user may include in the bug description some information on how to reproduce the bug. Oftentimes, users simply insert code snippets or dump the execution stack traces.

In this study, among our dataset of 162 bug reports, we note that only 27 (i.e., ~17%) are reported by users who are also developer-contributing to the projects. 15 (i.e., ~9%) bugs are reported and again fixed by the same project contributors. These percentages suggest that, for the majority of cases, the bug reports are indeed genuinely submitted by users of the software who require project developers’ attention.

*Given the buggy program version and a bug report, IFIXR must unfold the workflow for precisely identifying (at the statement level) the buggy code locations. We remind the reader that, in this step, future test cases cannot be relied upon. We consider that if such test cases could have triggered the bug, a continuous integration system would have helped developers deal with the bug before the software is shipped towards users.*

#### 3.2 Fault Localization w/o Test Cases

To identify buggy code locations within the source code of a program, we resort to Information Retrieval (IR)-based fault localization (IRFL) [67, 80]. The general objective is to leverage potential similarity between the terms used in a bug report and the source code to identify relevant buggy code locations. The literature includes a large body of work on IRFL [57, 72, 81, 85, 87, 96, 99] where researchers systematically extract tokens from a given bug report to formulate a *query* to be matched in a search space of *documents* formed by the collections of source code files and indexed through tokens extracted from source code. IRFL approaches then rank the documents based on a probability of relevance (often measured as a similarity score). Highly ranked files are predicted to be the ones that are likely to contain the buggy code.

Despite recurring interest in the literature, with numerous approaches continuously claiming new performance improvements
over the state-of-the-art, we are not aware of any adoption in program repair research or practice. We postulate that one of the reasons is that IRLF techniques have so far focused on file-level localization, which is too coarse-grained (in comparison to spectrum-based fault localization output). Recently, Locus [85] and BLIA [96] are state-of-the-art techniques which narrow down localization, respectively to the code change or the method level. Nevertheless, to the best of our knowledge, no IRLF technique has been proposed in the literature for statement-level localization.

In this work, we develop an algorithm to rank suspicious statements based on the output (i.e., files) of a state-of-the-art IRLF tool, thus yielding a fine-grained IRLF-based fault localizer which will then be readily integrated into a concrete patch generation pipeline.

### 3.2.1 Ranking Suspicious Files

We leverage an existing IRLF tool. Given that expensive extractions of tokens from a large corpus of bug reports is often necessary to tune IRLF tools [41], we selected a tool for which the authors provide datasets and pre-processed data. We use the D&C [31] as the specific implementation of file-level IRLF available online [1], which is a machine learning-based IRLF tool using a similarity matrix of 70-dimension feature vectors (7 features from bug reports and 10 features from source code files): D&C uses multiple classifier models that are trained each for specific groups of bug reports. Given a bug report, the different predictions of the different classifiers are merged to yield a single list of suspicious code files. Our execution of D&C (Line 2 in Algorithm 1) is tractable given that we only need to preprocess those bug reports that we must localize. Trained classifiers are already available. We ensure that no data leakage is induced (i.e., the classifiers are not trained with bug reports that we want to localize in this work).

### 3.2.2 Ranking Suspicious Statements

Patch generation requires fine-grained information on code entities that must be changed. For iFixR, we propose to produce a standard output, as for spectrum-based fault localization, to facilitate integration and reuse of state-of-the-art patch generation techniques. To start, we build on the conclusions on a recent large-scale study [47] of bug fixes to limit the search space of suspicious locations to the statements that are more error-prone. After investigating in detail the abstract syntax tree (AST)-based code differences of over 16,000 real-world patches from Java projects, Liu et al. [47] reported that the following specific AST statement nodes were significantly more prone to be faulty than others: IFStatements, ExpressionStatements, FieldDeclarations, ReturnStatements and VariableDeclarations. Lines 7–17 in Algorithm 1 detail the process to produce a ranked list of suspicious statements.

Algorithm 1 describes the process of our fault localization approach used in iFixR. Top \( k \) files are selected among the returned list of suspicious files of the IRFL along with their computed suspiciousness scores. Then each file is parsed to retain only the relevant error-prone statements from which textual tokens are extracted. The summary and descriptions of the bug report are also analyzed (lexically) to collect all its tokens. Due to the specific nature of stack traces and other code elements which may appear in the bug report, we use regular expressions to detect stack traces and code elements to improve the tokenization process, which is based on punctuation, camel case splitting (e.g., findNumber splits into find, number) as well as snake case splitting (e.g., find_number splits into find, number). Stop word removal [2] is then applied before performing stemming (using the PorterStemmer [25]) on all tokens to create homogeneity with the term’s root (i.e., by conflating variants of the same term). Each bag of tokens (for the bug report, and for each statement) is then eventually used to build a feature vector. We use cosine similarity among the vectors to rank the file statements that are relevant to the bug report.

Given that we considered \( k \) files, the statements of each having their own similarity score with respect to the bug report, we weight these scores with the suspiciousness score of the associated file. Eventually, we sort the statements using the weighted scores and produce a ranked list of code locations (i.e., statements in files) to be recommended as candidate fault locations.

### 3.3 Fix Pattern-based Patch Generation

A common, and reliable, strategy in automatic program repair is to generate concrete patches based on fix patterns [27] (also referred to as fix templates [51] or program transformation schemas [19]). Several APR systems [15, 19, 27, 32, 46, 48, 49, 51, 60, 73] in the literature implement this strategy by using diverse sets of fix patterns obtained either via manual generation or automatic mining of bug fix datasets. In this work, we consider the pioneer PAR system by Kim et al. [27]. Concretely, we build on \( k \)PAR [48], an open-source
Java implementation of PAR in which we included a diverse set of fix patterns collected from the literature. Table 4 provides an enumeration of fix patterns used in this work. For more implementation details, we refer the reader to our replication package. All tools and data are released as open source to the community to foster further research into these directions. As illustrated in Figure 3, a fix pattern encodes the recipe of change actions that should be applied to mutate a code element.

### Table 4: Fix patterns implemented in iFixR.

<table>
<thead>
<tr>
<th>Pattern description used by</th>
<th>Pattern description used by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Cast Checker Genesis</td>
<td>Mutate Literal Expression SimFix</td>
</tr>
<tr>
<td>Insert Null Pointer Checker NPIFix</td>
<td>Mutate Method Invocation ELDR</td>
</tr>
<tr>
<td>Insert Range Checker SORix</td>
<td>Mutate Operator jMutRepair</td>
</tr>
<tr>
<td>Insert Missing Statement HDRepair</td>
<td>Mutate Return Statement SketchFix</td>
</tr>
<tr>
<td>Mutate Conditional Expression sMut</td>
<td>Mutate Variable CapGen</td>
</tr>
<tr>
<td>Mutate Data Type AVATAR</td>
<td>Move Statement(s) PAR</td>
</tr>
<tr>
<td>Remove Statement(s) FixMiner</td>
<td></td>
</tr>
</tbody>
</table>

We mention only one example tool even when several tools implement it.

```java
if (exp instanceof T) {
    return getCumPct((Comparable<v>) v);
}
```

**Figure 3:** Illustration of “Insert Cast Checker” fix pattern.

For a given reported bug, once our fault localizer yields its list of suspicious statements, iFixR iteratively attempts to select fix patterns for each statement. The selection of fix patterns is conducted in a naive way based on the context information of each suspicious statement (i.e., all nodes in its abstract syntax tree, AST). Specifically, iFixR parses the code and traverses each node of the suspicious statement AST from its first child node to its last leaf node in a breadth-first strategy (i.e., left-to-right and top-to-bottom). If a node matches the context of a fix pattern (i.e., same AST node types), the fix pattern will be applied to generate patch candidates by mutating the matched code entity following the recipe in the fix pattern. Whether the node matches a fix pattern or not, iFixR keeps traversing its children nodes and searches fix patterns for them to generate patch candidates successively. This process is iteratively performed until leaf nodes are encountered.

Consider the example of bug Math-75 illustrated in Figure 4. If iFixR parses the buggy statement (i.e., statement at line 302 in the file Frequency.java) into an AST as illustrated by Figure 5. First, if iFixR matches a fix pattern that can mutate the expression in the return statement with other expression(s) returning data of type **double**. It further selects fix patterns for the direct child node (i.e., method invocation: `getCumPct((Comparable<v>) v)`)) of the return statement. This method invocation can be matched against fix patterns with two contexts: method name and parameter(s). With the breadth-first strategy, iFixR assigns a fix pattern, calling another method with the same parameters (cf. PAR [27, page 804]), to mutate the method name, and then selects fix patterns to mutate the parameter. Furthermore, iFixR will match fix patterns for the type and variable of the cast expression respectively and successively.

#### 3.4 Patch Validation with Regression Testing

For every reported bug, fault localization followed by pattern matching and code mutation will yield a set of patch candidates. In a typical test-based APR system, these patch candidates must let the program pass all test cases (including some positive test cases [83], which encode the actual functional requirements relevant to the bug). Thus, the patch candidates set is actively pruned to remove all patches that do not meet these requirements. In our work, in accordance with our investigation findings that such test cases may not be available at the time the bug is reported (cf. Section 2), we assume that iFixR cannot reason about future test cases to select patch candidates.

Instead, we rely only on past test cases, which were available in the code base, when the bug is reported. Such test cases are leveraged to perform regression testing [95], which will ensure that, at least, the selected patches do not obstruct the behavior of the existing, unchanged part of the software, which is already explicitly encoded by developers in their current test suite.

#### 3.5 Output: Patch Recommendation List

Eventually, iFixR produces a ranked recommendation list of patch suggestions for developers. Until now, the order of patches is influenced mainly by two steps in the workflow:

1. (localization): our statement-level IRFL yields a ranked list of statements to modify in priority.
2. (pattern matching): the AST node of the buggy code entity is broken down into its children and iteratively navigated in a breadth-first manner to successively produce candidate patches.

Eventually, the produced list of patches has an order, which carries the biases of fault localization [48], and is biased by the pre-set breadth-first strategy for matching fix patterns. We thus design an ordering process with a function $f_{\text{rcmd}}: \mathbb{P}^n \rightarrow \mathbb{P}^k$, as follows:

$$f_{\text{rcmd}}(p) = (p_{\text{pris}} \circ p_{\text{prisup}} \circ p_{\text{prichange}})(p)$$

where $p_{\text{pris}}$, $p_{\text{prisup}}$, and $p_{\text{prichange}}$ are three heuristics-based prioritization functions used in iFixR. $f_{\text{rcmd}}$ takes a set of patches validated via regression testing (cf. Section 3.4) and produces an ordered sequence of patches ($f_{\text{rcmd}}(p) \in \mathbb{P}^k$). We propose the following heuristics to re-prioritize the patch candidates:

1. **Minimal changes**: we favor patches that minimize the differences between the patched program and the buggy program. To that end, patches are ordered following their AST edit script sizes. Formally, we define $p_{\text{prichange}}: \mathbb{Z}^n \rightarrow \mathbb{P}$ where $n = \{p_{\text{pris}}, p_{\text{prisup}}, p_{\text{prichange}}\}$ and holds $\forall p \in \mathbb{P}$, $C_{\text{change}}(p_i) \leq C_{\text{change}}(p_{i+1})$ where $C_{\text{change}}(p)$ is a function that counts the number of deleted and inserted AST nodes by the change actions of $p$. The domain of the function is a power set $\mathbb{Z}^n$, and the co-domain (Ek) is a $k$-dimensional vector space [29] where $k$ is the maximum number of recommended patches, and Ek denotes the set of all generated patches.
Therefore, for these projects, bug IDs referred to in the commit logs are ambiguous (for some bugs this may match with the GitHub issue tracking numbering, while in others, it refers to the original issue tracker). To avoid introducing noise in our validation data, we simply drop these projects. For the remaining projects (Lang and Math), we leverage the bug linking strategies implemented in the Jira issue tracking software. We use a similar approach to Fischer et al. [16] and Thomas et al. [78] to link to commits to corresponding bug reports. Concretely, we crawled the bug reports related to each project and assessed the links with a two-step search strategy: (i) we check commit logs to identify bug report IDs and associate the corresponding changes as bug fix changes; then (ii) we check for bug reports that are indeed considered as such (i.e., tagged as "BUG") and are further marked as resolved (i.e., with tags "RESOLVED" or "FIXED"), and completed (i.e., with status "CLOSED").

Eventually, our evaluation dataset includes 156 faults (i.e., Defects4J bugs). Actually, for the considered projects, Defects4J enumerates 171 bugs associated with 162 bug reports: 15 bugs are indeed left out because either (1) the corresponding bug reports are not in the desired status in the bug tracking system, which may lead to noisy data, or (2) there is ambiguity in the buggy program version (e.g., some fixed files appear to be missing in the repository at the time of bug reporting).

4.1.2 Test suite reorganization. We ensure that the benchmark separates past test cases (i.e., regression test cases) from future test cases (i.e., test cases that encode functional requirements specified after the bug is reported). This timeline split is necessary to simulate the snapshot of the repository at the time the bug is reported. As highlighted in Section 2, for over 90% cases of bugs in the Defects4J benchmark, the test cases relevant to the defective behavior was actually provided along the bug fixing patches. We have thus manually split the commits to identify test cases that should be considered as future test cases for each bug report.

4.4 Implementation Choices

During implementation, we have made the following parameter choices in the ifixR workflow:

- IR fault localization considers the top 50 (i.e., k = 50 in Algorithm 1) suspicious files for each bug report, in order to search for buggy code locations.
- For patch recommendation experiments, we limit the search space to the top 20 suspected buggy statements yielded by the fine-grained IR-based fault localization.
- For comparison experiments, we implement spectrum-based fault localization using the GZoltar testing framework with the Ochiai ranking strategy. Unless otherwise indicated, GZoltar version 0.1.1 is used (as it is widely adopted in the literature, by Astor [59], ACS [92], ssFix [90] and CapGen [84] among others).

4.5 Research Questions

The assessment objective is to assess the feasibility of automating the generation of patches for user-reported bugs, while investigating the foreseen bottlenecks as well as the research directions that the community must embrace to realize this long-standing endeavor. To that end, we focus on the following research questions associated with the different steps in the ifixR workflow.

- RQ1 [Fault localization]: To what extent does IR-based fault localization provide reliable results for an APR scenario? In particular, we investigate the performance differences when comparing our

(2) [Fault localization suspiciousness]: when two patch candidates have equal edit script sizes, the tie is broken by using the suspiciousness scores (of the associated statements) yielded during IR-based fault localization. Thus, when $C_{change}(p_i) = C_{change}(p_{i+1})$, $pr_{issus}$ re-orders the two patch candidates. We define $pr_{issus} : \mathbb{P} \rightarrow \mathbb{P}$ such that $pr_{issus}(seq_{change}) = [\cdots, p_i, p_{i+1}, \cdots]$ holds $S_{issus}(p_i) \geq S_{issus}(p_{i+1})$, where $seq_{change}$ is the result of $pr_{change}$ and $S_{issus}$ returns a suspicious score of the statement that a given patch $p_j$ changes.

(3) [Affected code elements]: after a manual analysis of fix patterns and the performance of associated APR in the literature, we empirically found that some change actions are irrelevant to bug fixing. Thus, for the corresponding pre-defined patterns, ifixR systematically under-prioritizes their generated patches against any other patches, although among themselves the ranking obtained so far (through $pr_{change}$ and $pr_{issus}$) is preserved for those under-prioritized patches. These are patches generated by (i) mutating a literal expression, (ii) mutating a variable into a method invocation or a final static variable, or (iii) inserting a method invocation without parameter. This prioritization, is defined by $pr_{type} : \mathbb{P} \rightarrow \mathbb{P}$, which returns a sequence of top $k$ ordered patches ($k \leq n = |patches|$). To define this prioritization function, we assign natural numbers $j_1, j_2, j_3, j_4 \in \mathbb{N}$ to each patch generation types (i.e., $j_1 \leftarrow (i)$, $j_2 \leftarrow (ii)$, and $j_3 \leftarrow (iii)$, respectively) and $(j_4 \leftarrow)$ everything else, which strictly hold $j_1 > j_2 > j_3 > j_4$. This prioritization function takes the result of $pr_{issus}$ and returns another sequence $[p_{i_1}, p_{i_2}, p_{i_3}, \cdots]$ that holds $\forall p_i, D_{type}(p_i) \geq D_{type}(p_{i+1})$. $D_{type}$ is defined as $D_{type} : \mathbb{P} \rightarrow \{j_1, j_2, j_3, j_4\}$ and determines how a patch $p_j$ has been generated as defined above. From the ordered sequence, the function returns the leftmost (i.e., top) $k$ patches as a result.
5 ASSESSMENT RESULTS

In this section, we present the results of the investigations for the previously enumerated research questions.

5.1 RQ1: [Fault Localization]

Fault localization being the first step in program repair, we evaluate the performance of the IR-based fault localization developed within iFixR. As recently thoroughly studied by Liu et al. [48], an APR tool should not be expected to fix a bug that current fault localization systems fail to localize. Nevertheless, with iFixR, we must demonstrate that our fine-grained IRFL offers comparable performance with SBFL tools used in the APR literature.

Table 5 provides performance measurements on the localization of bugs. SBFL is performed based on two different versions of the GZoltar testing framework, but always based on the Ochiai ranking metric. Finally, because fault localization tools output a ranked list of suspicious statements, results are provided in terms of whether the correct location is placed under the top-k suspected statements. In this work, following the practice in the literature [48, 56], we consider that a bug is localized if any buggy statement is localized.

Table 5: Fault localization results: IRFL (IR-based) vs. SBFL (Spectrum-based) on Defects4J (Math and Lang) bugs.

<table>
<thead>
<tr>
<th></th>
<th>IRFL</th>
<th>SBFL GZv1</th>
<th>SBFL GZv2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Top-1</td>
<td>8/17</td>
<td>7/13</td>
<td>7/12</td>
</tr>
<tr>
<td>Top-10</td>
<td>15/22</td>
<td>11/12</td>
<td>10/12</td>
</tr>
<tr>
<td>Top-50</td>
<td>17/19</td>
<td>11/12</td>
<td>10/12</td>
</tr>
<tr>
<td>Top-100</td>
<td>17/19</td>
<td>11/12</td>
<td>10/12</td>
</tr>
<tr>
<td>All</td>
<td>17/19</td>
<td>11/12</td>
<td>10/12</td>
</tr>
</tbody>
</table>

Overall, the results show that our IRFL implementation is strictly comparable to the common implementation of spectrum-based fault localization when applied on the Defects4J bug dataset. Note that the comparison is conducted for 171 bugs of Math and Lang, given that these are the projects for which the bug linking can be reliably performed for applying the IRFL. Although performance results are similar, we remind the reader that SBFL is applied by considering future test cases. To highlight a practical interest of IRFL, we compute for each bug localizable in the top-10, the elapsed time between the bug report date and the date the relevant test case is submitted for this bug. Based on the distribution shown in Figure 6, on mean average, IRFL could reduce this time by 26 days.

Figure 6: Distribution of elapsed time (in days) between bug report submission and test case attachment.

Finally, to stress the importance of future test cases for spectrum-based fault localization, we consider all Defects4J bugs and compute localization performance with and without future test cases.

Results listed in Table 6 confirms that in most bug cases, the localization is impossible: Only 10 bugs (out of 395) can be localized among the top-10 suspicious statements of SBFL at the time the bug is reported. In comparison, our IRFL locates 72 bugs under the same conditions of having no relevant test cases to trigger the bugs.

Table 6: Fault localization performance.

<table>
<thead>
<tr>
<th></th>
<th>GZoltar + Ochiai (395 bugs)</th>
<th>Top-1</th>
<th>Top-10</th>
<th>Top-50</th>
<th>Top-100</th>
<th>Top-200</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>without future tests</td>
<td>5</td>
<td>10</td>
<td>17</td>
<td>17</td>
<td>19</td>
<td>20</td>
</tr>
<tr>
<td></td>
<td>with future tests</td>
<td>45</td>
<td>140</td>
<td>198</td>
<td>214</td>
<td>239</td>
<td>263</td>
</tr>
</tbody>
</table>

Fine-grained IR-based fault localization in iFixR is as accurate as Spectrum-based fault localization in localizing Defects4J bugs. Additionally, it does not have the constraint of requiring test cases that may not be available when the bug is reported.

5.2 RQ2: [Overfitting]

Patch generation attempts to mutate suspected buggy code with suitable fix patterns. Aside from having adequate patterns or not (which is out of the scope of our study), a common challenge of APR lies in the effective selection of buggy statements. In typical test-based APR, test cases drive the selection of these statements. The incompleteness of test suites is however currently suspected to often lead to overfitting of generated patches [94].

We perform patch generation experiments to investigate the impact of localization bias. We compare our IRFL implementation against commonly-used SBFL implementations in the literature of test-based APR. We recall that the patch validation step in these experiments makes no assumptions about future test cases (i.e., all test cases are leveraged as in classical APR pipeline). For each bug, depending on the rank of the buggy statements yielded the fault localization system (either IRFL or SBFL), the patch generation can produce more or less relevant patches. Table 7 details the repair performance in relation to the position of buggy statements in the output of fault localization. Results are provided in terms of numbers of plausible and correct [69] patches that can be found by considering top-k statements returned by the fault localizer.

Table 7: IRFL vs. SBFL impacts on the number of generated correct/plausible patches for Defects4J bugs.

<table>
<thead>
<tr>
<th></th>
<th>Lang</th>
<th>Math</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>IRFL Top-1</td>
<td>4/4</td>
<td>3/3</td>
<td>4/0</td>
</tr>
<tr>
<td>SBFL Top-1</td>
<td>1/4</td>
<td>0/1</td>
<td>1/0</td>
</tr>
<tr>
<td>IRFL Top-5</td>
<td>3/6</td>
<td>7/14</td>
<td>10/0</td>
</tr>
<tr>
<td>SBFL Top-5</td>
<td>2/7</td>
<td>11/17</td>
<td>13/24</td>
</tr>
<tr>
<td>IRFL Top-10</td>
<td>4/9</td>
<td>9/14</td>
<td>13/24</td>
</tr>
<tr>
<td>SBFL Top-10</td>
<td>4/11</td>
<td>16/27</td>
<td>20/38</td>
</tr>
<tr>
<td>IRFL Top-20</td>
<td>7/12</td>
<td>9/18</td>
<td>16/30</td>
</tr>
<tr>
<td>SBFL Top-20</td>
<td>4/11</td>
<td>18/30</td>
<td>22/41</td>
</tr>
<tr>
<td>IRFL Top-50</td>
<td>7/45</td>
<td>20/22</td>
<td>27/37</td>
</tr>
<tr>
<td>SBFL Top-50</td>
<td>4/11</td>
<td>19/34</td>
<td>23/47</td>
</tr>
<tr>
<td>IRFL Top-100</td>
<td>8/48</td>
<td>20/35</td>
<td>28/43</td>
</tr>
<tr>
<td>SBFL Top-100</td>
<td>5/14</td>
<td>19/36</td>
<td>24/50</td>
</tr>
<tr>
<td>IRFL All</td>
<td>15/19</td>
<td>10/25</td>
<td>24/44</td>
</tr>
<tr>
<td>SBFL All</td>
<td>5/14</td>
<td>19/36</td>
<td>24/50</td>
</tr>
</tbody>
</table>

We indicate x/y numbers of patches: x is the number of bugs for which a correct patch is generated; y is the number of bugs for which a plausible patch is generated.
Overall, we find that IRFL and SBFL localization information lead to similar repair performance in terms of the number of fixed bugs plausibly/correctly. Actually IRFL-supported APR outperforms SBFL-supported APR on the Lang project bugs and vice-versa for Math project bugs: overall, 6 bugs that are fixed using IRFL output, cannot be fixed using SBFL output (although assuming the availability of the bug triggering test cases to run the SBFL tool).

We investigate the cases of plausible patches in both localization scenarios to characterize the reasons why these patches appear to only be overfitting the test suites. Table 8 details the overfitting reasons for the two scenarios.

Table 8: Dissection of reasons why patches are plausible* but not correct.

<table>
<thead>
<tr>
<th>w/ IRFL</th>
<th>w/ SBFL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Localization Error</td>
<td>15</td>
</tr>
<tr>
<td>Pattern Prioritization</td>
<td>1</td>
</tr>
<tr>
<td>Lack of Fix ingredients</td>
<td>6</td>
</tr>
</tbody>
</table>

*(A plausible patch passes all test cases, but may not be semantically equivalent to developer patch (i.e., correct). We consider a plausible patch to be overfitted to the test suite

(1) Among the 23 (= 44 − 21) plausible patches that are generated based on IRFL identified code locations and that are not found to be correct, 6 are found to be caused by fault localization errors: these bugs are plausibly fixed by mutating irrelevantly-suspicious statements that are placed before the actual buggy statements in the fault localization output list. This phenomenon has been recently investigated in the literature as the problem of fault localization bias [48]. Nevertheless, we note that patches generated based on SBFL identified code locations suffer more of fault localization bias: 15 of the 26 (≈ 50−20) plausible patches are concerned by this issue.

(2) Pattern prioritization failures may lead to plausible patches: while a correct patch could have been generated using a specific pattern at a lower node in the AST, another pattern (leading to an only plausible patch) was first matched the node during the iterative search of matching nodes (cf. Section 3.3).

(3) Finally, we note that both configurations yield plausible patches due to the lack of suitable patterns or due to a failed search for the adequate donor code (i.e., fix ingredient [44]).

Experiments with the Defects4J dataset suggest that code locations provided by IR-based fault localization lead less to overfitted patches than the code locations suggested by Spectrum-based fault localization: cf. “Localization error” column in Table 8.

5.3 RQ3: [Patch Ordering]

While the previous experiment focused on patch generation, our final experiment assesses the complete pipeline of 1F1xR as it was imagined for meeting the constraints that developers can face in practice: future test cases, i.e., those which encode the functionality requirements that are not met by the buggy programs, may not be available at the time the bug is reported. We thus discard the future test cases of the Defects4J dataset and generate patches that must be recommended to developers. The evaluation protocol thus consists in assessing to what extent correct/plausible patches are placed in the top of the recommendation list.

5.3.1 Overall performance. Table 9 details the performance of the patch recommendation by 1F1xR: we present the number of bugs for which a correct/plausible patch is generated and presented among the top-k of the list of recommended patches. In the absence of future test cases to drive the patch validation process, we use heuristics (cf. Section 4.2) to re-prioritize the patch candidates towards ensuring that patches which are recommended first will eventually be correct (or at least plausible when relevant test cases are implemented). We present results both for the case where we do not re-prioritize and the case where we re-prioritize.

Recall that, given that the re-organized benchmark separately includes the future test cases, we can leverage them to systematize the assessment of patch plausibility. The correctness (also referred to as correctness [69]) of patches, however, is still decided manually by comparing against the actual bug fix provided by developers and available in the benchmark. Overall, we note that 1F1xR performance is promising as it manages, for 13 bugs, to present a plausible patch among its top-5 recommended patches per bug. Among those plausible patches, 8 are eventually found to be correct.

Table 9: Overall performance of 1F1xR for patch recommendation on the Defects4J benchmark.

<table>
<thead>
<tr>
<th>Recommendation rank</th>
<th>Top-1</th>
<th>Top-5</th>
<th>Top-10</th>
<th>Top-20</th>
<th>All</th>
</tr>
</thead>
<tbody>
<tr>
<td>w/ IRFL w/ SBFL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>without patch re-prioritization</td>
<td>3/1</td>
<td>4/9</td>
<td>6/10</td>
<td>6/10</td>
<td>13/27</td>
</tr>
<tr>
<td>with patch re-prioritization</td>
<td>3/4</td>
<td>8/13</td>
<td>9/14</td>
<td>10/15</td>
<td>17/27</td>
</tr>
</tbody>
</table>

* x/y: x is the number of bugs for which a correct patch is generated, y is the number of bugs for which a plausible patch is generated.

5.3.2 Comparison with the state-of-the-art test-based APR systems. To objectively position the performance of 1F1xR (which does not require future test cases to localize bugs, generate patches and present a sorted recommendation list of patches), we count the number of bugs for which 1F1xR can propose a correct/plausible patch. We consider three scenarios with 1F1xR:

(1) [1F1xR_{top5}] - developers will be provided with only top 5 recommended patches which have been validated only with regression tests: in this case, 1F1xR outperforms about half of the state-of-the-art in terms of numbers bugs fixed with both plausible or correct patches.

(2) [1F1xR_{all}] - developers are presented with all (i.e., not only top-5) generated patches validated with regression tests: in this case, only four (out of sixteen) state-of-the-art APR techniques outperform 1F1xR.

(3) [1F1xR_{opt}] - developers are presented with all generated patches which have been validated with augmented test suites (i.e., optimistically with future test cases): with this configuration, 1F1xR outperforms all state-of-the-art, except SimFix [20] which uses sophisticated techniques to improve the fault localization accuracy and search for fix ingredients. It should be noted that in this case, our prioritization strategy is not applied to the generated patches. 1F1xR_{opt} represents the reference performance for our experiment which assesses the prioritization.

Table 10 provides the comparison matrix. Information on state-of-the-art results are excerpted from their respective publications.

1F1xR offers a reasonable performance in patch recommendation when we consider the number of Defects4J bugs that are successfully patched among the top-5 (in a scenario where we assume not having relevant test cases to validate the patch candidates). Performance results are even comparable to many state-of-the-art test-based APR tools in the literature.
### Table 10: iFixR vs state-of-the-art APR tools.

<table>
<thead>
<tr>
<th>APR tool</th>
<th>Lang*</th>
<th>Math*</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>jComFix [39]</td>
<td>0/0</td>
<td>5/18</td>
<td>5/18</td>
</tr>
<tr>
<td>jKah [39]</td>
<td>0/0</td>
<td>5/18</td>
<td>5/18</td>
</tr>
<tr>
<td>jMutRepair [50]</td>
<td>0/0</td>
<td>1/14</td>
<td>1/14</td>
</tr>
<tr>
<td>HDRepair [37]</td>
<td>2/6</td>
<td>4/7</td>
<td>6/13</td>
</tr>
<tr>
<td>Nopid [31]</td>
<td>3/7</td>
<td>1/12</td>
<td>4/28</td>
</tr>
<tr>
<td>ACS [92]</td>
<td>3/4</td>
<td>12/16</td>
<td>15/20</td>
</tr>
<tr>
<td>ELXIR [75]</td>
<td>8/12</td>
<td>12/19</td>
<td>20/31</td>
</tr>
<tr>
<td>stFix [90]</td>
<td>5/12</td>
<td>10/26</td>
<td>15/38</td>
</tr>
<tr>
<td>CapCen [64]</td>
<td>5/5</td>
<td>12/16</td>
<td>17/21</td>
</tr>
<tr>
<td>SketchFix [10]</td>
<td>3/4</td>
<td>7/8</td>
<td>10/12</td>
</tr>
<tr>
<td>FixMiner [32]</td>
<td>2/3</td>
<td>12/14</td>
<td>14/17</td>
</tr>
<tr>
<td>LSRepair [44]</td>
<td>8/14</td>
<td>7/14</td>
<td>15/28</td>
</tr>
<tr>
<td>SimFix [20]</td>
<td>3/4</td>
<td>14/26</td>
<td>23/10</td>
</tr>
<tr>
<td>kPAR [40]</td>
<td>1/8</td>
<td>7/8</td>
<td>8/26</td>
</tr>
</tbody>
</table>

* X/y: x is the number of bugs for which a correct patch is generated; y is the number of bugs for which a plausible patch is generated.

5.3.3 Properties of iFixR’s patches. In Table 11, we characterize the correct and plausible patches recommended by iFixR. Overall, update and insert changes have been successful; most patches affect a single statement, and impact precisely an expression entity within a statement.

### Table 11: Change properties of iFixR’s correct patches.

<table>
<thead>
<tr>
<th>Change action</th>
<th>#bugs*</th>
<th>Impacted statement(s)</th>
<th>Granularity</th>
<th>#bugs*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Update</td>
<td>5/7</td>
<td>Single-statement</td>
<td>Statement</td>
<td>1/2</td>
</tr>
<tr>
<td>Insert</td>
<td>3/5</td>
<td>Multiple-statement</td>
<td>Expression</td>
<td>7/11</td>
</tr>
<tr>
<td>Delete</td>
<td>0/1</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* x/y — x for x bugs the patches are correct, while y bugs they are plausible.

5.3.4 Diversity of iFixR’s fixed bugs. Finally, in Table 12 we dissect the nature of the bugs for which iFixR recommends a correct or a plausible patch. Priority information about the bug report is collected from the issue tracking systems, while the root cause is inferred by analyzing the bug reports and fixes.

### Table 12: Dissection of bugs successfully fixed by iFixR.

<table>
<thead>
<tr>
<th>Patch Type</th>
<th>Defect4J Bug ID</th>
<th>Issue ID</th>
<th>Root Cause</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>G</td>
<td>L-6</td>
<td>LANG-857</td>
<td>String index out of bounds exception</td>
<td>Minor</td>
</tr>
<tr>
<td>G</td>
<td>L-24</td>
<td>LANG-664</td>
<td>Wrong behavior due missing condition</td>
<td>Major</td>
</tr>
<tr>
<td>G</td>
<td>L-57</td>
<td>LANG-304</td>
<td>Null pointer exception</td>
<td>Major</td>
</tr>
<tr>
<td>G</td>
<td>M-15</td>
<td>MATH-904</td>
<td>Double precision floating point format error</td>
<td>Major</td>
</tr>
<tr>
<td>G</td>
<td>M-34</td>
<td>MATH-779</td>
<td>Missing &quot;read only access&quot; to internal list</td>
<td>Major</td>
</tr>
<tr>
<td>G</td>
<td>M-35</td>
<td>MATH-776</td>
<td>Range check</td>
<td>Major</td>
</tr>
<tr>
<td>G</td>
<td>M-57</td>
<td>MATH-546</td>
<td>Wrong variable type truncates double value</td>
<td>Minor</td>
</tr>
<tr>
<td>G</td>
<td>M-75</td>
<td>MATH-329</td>
<td>Method signature mismatch</td>
<td>Minor</td>
</tr>
<tr>
<td>P</td>
<td>L-13</td>
<td>LANG-748</td>
<td>Serialization error in primitive types</td>
<td>Major</td>
</tr>
<tr>
<td>P</td>
<td>L-21</td>
<td>LANG-677</td>
<td>Wrong Date Format in comparison</td>
<td>Major</td>
</tr>
<tr>
<td>P</td>
<td>L-45</td>
<td>LANG-419</td>
<td>Range check</td>
<td>Minor</td>
</tr>
<tr>
<td>P</td>
<td>L-58</td>
<td>LANG-300</td>
<td>Number formatting error</td>
<td>Minor</td>
</tr>
<tr>
<td>P</td>
<td>M-2</td>
<td>MATH-1621</td>
<td>Integer overflow</td>
<td>Major</td>
</tr>
</tbody>
</table>

G’ denotes correct patch and P’ means plausible patch.

Overall, we note that 9 out of the 13 bugs have been marked as Major issues. 12 different bug types (i.e., root causes) are addressed. In contrast, R2Fix [43] only focused on 3 simple bug types.

### 6 DISCUSSION

This study presents the conclusions of our investigation into the feasibility of generating patches automatically from bug reports. We set strong constraints on the absence of test cases, which are used in test-based APR to approximate what the program is actually supposed to do and when the repair is completed [83]. Our experiments on the widely-used Defect4J bugs eventually show that patch generation without bug-triggering test cases is promising.

Manually looking at the details of failures and success in generating patches with iFixR, several insights can be drawn:

**Test cases can be buggy:** During manual analysis of results, we noted that if iFixR actually fails to generate correct patches for three bugs (namely, Math-5, Math-59 and Math-65) because even the test cases were buggy. Figure 7 illustrates the example of bug Math-5 where its patch also updated the relevant test case. This example supports our endeavor, given that users would find and report bugs for which the appropriate test cases were never properly written.

![Figure 7: Patched source code and test case of fixing Math-5.](image_url)

**Bug reports deserve more interest:** With iFixR, we have shown that bug reports could be handled automatically for a variety of bugs. This is an opportunity for issue trackers to add a recommendation layer to the bug triaging process by integrating patch generation techniques. There are, however, several directions to further investigation, among which: (1) help users write proper bug reports; and (2) re-investigate IRFL techniques at a finer-grained level that is suitable for APR.

**Prioritization techniques must be investigated:** In the absence of complete test suites for validating every single patch candidate, a recommendation system must ensure that patches presented first to the developers are the most likely to be plausible and even correct. There are thus two directions of research that are promising: (1) ensure that fix patterns are properly prioritized to generate good patches and be able to early-stop for not exploding the search space; and (2) ensure that candidate patches are effectively re-prioritized. These investigations must start with a thorough dissection of plausible patches for a deep understanding of plausibility factors.

**More sophisticated approaches to triaging and selecting fix ingredients are necessary:** In its current form, iFixR implements a naïve approach to patch generation, ensuring that the performance is tractable. However, the literature already includes novel APR techniques that implement strategies for selecting donor code and filters patterns. Integrating such techniques into iFixR may lead to performance improvement.

**More comprehensive benchmarks are needed:** Due to bug linking challenges, our experiments were only performed on half of the Defects4J benchmark. To drive strong research in patch generation for user-reported bugs, the community must build larger and reliable benchmarks, potentially even linking several artifacts of continuous integration (i.e., build logs, past execution traces, etc.). In the future, we plan to investigate the dataset of Bugs.jar [71].
Automatic test generation techniques could be used as a supplement: Our study tries to cope radically with the incompleteness of test suites. In the future, however, we could investigate the use of automatic test generation techniques to supplement the regression test cases during patch validation.

7 THREATS TO VALIDITY

Threats to external validity: The bug reports used in this study may be of low quality (i.e., wrong links for corresponding bugs). We reduced this threat by focusing only on bugs from the Lang and Math projects, which kept a single issue tracking system. We also manually verified the links between the bug reports and the Defects4J bugs. Table 13 characterizes the bug reports of our dataset following the criteria enumerated by Zimmermann et al. [100] in their study of “what makes a good bug report”. Notably, as illustrated by the distribution of comments in Figure 8, we note that the bug reports have been actively discussed before being resolved. This suggests that they are not trivial cases (cf. [18]).

Table 13: Dissection of bug reports related to Defects4J bugs.

<table>
<thead>
<tr>
<th>Proj</th>
<th>Unique Bug Reports</th>
<th>w/ Patch Attached</th>
<th>Average Comments</th>
<th>w/ Stack Traces</th>
<th>w/ Hints</th>
<th>w/ Code Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lang</td>
<td>62</td>
<td>11</td>
<td>4.53</td>
<td>4</td>
<td>62</td>
<td>31</td>
</tr>
<tr>
<td>Math</td>
<td>100</td>
<td>23</td>
<td>5.15</td>
<td>5</td>
<td>92</td>
<td>51</td>
</tr>
</tbody>
</table>

Code-related terms such as package/class names found in the summary and description, in addition to stack traces and code blocks, as separate features referred to as hints.

Figure 8: Distribution of # of comments per bug report.

Another threat to external validity relates to the diversity of the fix patterns used in this study. iFixR currently may not implement a reasonable number of relevant fix patterns. We minimize this threat by surveying the literature and considering patterns from several pattern-based APR.

Threats to internal validity: Our implementation of fine-grained IRFL carries some threats: during the search of buggy statements, we considered top-50 suspicious buggy files from the file-level IRFL tool, to limit the search space. Different threshold values may lead to different results. We also considered only 5 statement types as more bug-prone. This second threat is minimized by the empirical evidence provided by Liu et al. [47].

Additionally, another internal threat is in our patch generation steps: iFixR only searches for donor code from the local code files, which contain the buggy statement. The adequate fix ingredient may however be located elsewhere.

Threats to construct validity: In this study, we assumed that patch construction and test case creation are two separated tasks for developers. This may not be the case in practice. The threat is however mitigated given that, in any case, we have shown that the test cases are often unavailable when the bug is reported.

8 RELATED WORK

Fault Localization. A recent study [48] stated, fault localization is a critical task affecting the effectiveness of automated program repair. Several techniques have been proposed [67, 80, 88] and they use different information such as spectrum [5], text [85], slice [58], and statistics [42]. The first two types of techniques are widely studies in the community. SBFL techniques [4, 21] are widely adopted in APR pipelines since they identify bug positions at the statement level. However, they have limitations on localizing buggy locations since it highly relies on the test suite [48]. IRFL [41] leverages textual information in a bug report. It is mainly used to help developers narrow down suspected buggy files in the absence of relevant test cases. For the purpose of our study, we have proposed an algorithm for localizing the faulty code entities at the statement level.

Patch Generation. Patch generation is another key process of APR pipeline, which is, in other words, a task searching for another shape of a program (i.e., a patch) in the space of all possible programs [38, 54]. To improve repair performance, many APR systems have been explored to address the search space problem by using different information and approaches: stochastic mutation [39, 83], synthesis [53, 92, 93], pattern [15, 19, 20, 27, 33, 37, 49, 51, 52, 73], contract [13, 82], symbolic execution [65], learning [9, 17, 55, 70, 77, 86], and donor code searching [26, 62]. In this paper, patch generation is implemented with fix patterns presented in the literature since it may make the generated patches more robust [74].

Patch Validation. The ultimate goal of APR systems is to automatically generate a correct patch that can actually resolve the program defects rather than satisfying minimal functional constraints. At the beginning, patch correctness is evaluated by passing all test cases [27, 37, 83]. However, these patches could be overfitting [35, 69] and even worse than the bug [76]. Since then, APR systems are evaluated with the precision of generating correct patches [20, 49, 84, 92]. Recently, researchers explore automated frameworks that can identify patch correctness for APR systems automatically [36, 91]. In this paper, our approach validates generated patches with regression test suites since fail-inducing test cases are readily available for most of bugs as described in Section 2.

9 CONCLUSION

In this study, we have investigated the feasibility of automating patch generation from bug reports. To that end, we implemented iFixR, an APR pipeline variant adapted to the constraints of test cases unavailability when users report bugs. The proposed system revisits the fundamental steps, notably fault localization, patch generation and patch validation, which are all tightly-dependent to the positive test cases [83] in a test-based APR system.

Without making any assumptions on the availability of test cases, we demonstrate, after re-organizing the Defects4J benchmark, that iFixR can generate and recommend priority correct (and more plausible) patches for a diverse set of user-reported bugs. The repair performance of iFixR is even found to be comparable to that of the majority of test-based APR systems on the Defects4J dataset.

We open source iFixR’s code and release all data of this study to facilitate replication and encourage further research in this direction which is promising for practical adoption in the software development community: https://github.com/SerVal-DTF/iFixR

ACKNOWLEDGMENTS

This work is supported by the Fonds National de la Recherche (FNR), Luxembourg, through RECOMMEND 15/IS/10449467 and FIXPATTERN C15/IS/9964569.
REFERENCES


