Mimic: Bug Report driven Program Repair

Anil Koyuncu
University of Luxembourg
Luxembourg
anil.koyuncu@uni.lu

Kui Liu*
University of Luxembourg
Luxembourg
kui.liu@uni.lu

Tegawendé F. Bissyandé
University of Luxembourg
Luxembourg
tegawende.bissyande@uni.lu

Dongsun Kim
University of Luxembourg
Luxembourg
dongsun.kim@uni.lu

Martin Monperrus
KTH Royal Institute of Technology
Sweden
martin.monperrus@csc.kth.se

Jacques Klein
University of Luxembourg
Luxembourg
jacques.klein@uni.lu

Yves Le Traon
University of Luxembourg
Luxembourg
yves.letraon@uni.lu

ABSTRACT

Issue tracking systems are commonly used in modern software development for collecting feedback from users and developers. An ultimate automation target of software maintenance is then the systematization 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, `MIMIC`, 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 `MIMIC` 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). `MIMIC` generates genuine/plausible patches for 21/44 Defects4J faults with its IR-based fault localizer. `MIMIC` 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).

*Corresponding author, the same contribution as the first author.

ESEC/FSE 2019, 26–30 August, 2019, Tallinn, Estonia
© 2019 Association for Computing Machinery.
ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

KEYWORDS

Information retrieval, fault localization, automatic patch generation.

1 INTRODUCTION

Automated program repair (APR) has gained incredible momentum in the last decade. Since the seminal work by Weimer et al. [88] who relied on genetic programming to evolve program variants until one variant is found to satisfy the functional constraints of a test suite, the community has been interested in test-based techniques to repair *programs without specifications*. Thus, various approaches [13, 14, 21, 23, 28, 29, 35, 36, 39, 42, 50, 51, 53, 54, 56, 64, 67, 88, 89, 97, 98] 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 [18], the current challenges lie in fixing *semantic bugs*, i.e., cases where implementation of program behavior deviates from developer's intention [63].

Ten years ago, the work of Weimer et al. [88] was explicitly motivated by the fact that, despite significant advances in specification mining (e.g., [44]), 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, 30, 70]. Therefore, while current test-based APR approaches would be suitable in a test-driven development[1] setting, their adoption by practitioners faces a simple reality: developers majoritarily (1) write few tests [30], (2) write tests after the source code [8], and (3) write tests to validate that bugs are indeed fixed and will not reoccur [26].

Although APR bots [83] 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 daily [5]. Bugs are indeed reported in natural language, where users tentatively describe the execution scenario that was being carried out and the unexpected outcome (e.g., crash stack traces). Such bug reports constitute an essential artifact within a software development cycle and can become an overwhelming concern for

[1]Test-Driven Development: requirements are translated into specific test cases and source code is written or iteratively improved to pass the tests [7].

software maintainers. For example, as early as in 2005, a triager of the Mozilla project was reported in [5, page 363] to have commented that:

> "Everyday, almost 300 bugs appear that need triaging. This is far too much for only the Mozilla programmers to handle."

With respect to bug reports, the literature proposes various techniques to improve their triaging, mainly by detecting duplicates [80, 81, 86], classifying them accurately [4, 19], assigning them efficiently [5, 6, 22, 62], or attempting to localize the reported bugs [74, 90, 92, 104]. Recently, in the Android realm, given limited execution scenarios on Graphical User Interfaces, researchers have shown that it is possible to automatically translate (properly-written) bug reports into test cases [16]. Very few studies [10, 47] have however undertaken to automate patch generation based on bug reports. To the best of our knowledge, Liu et al. [47] proposed the most advanced study in this direction. Unfortunately, their R2Fix approach carries several caveats: as illustrated in Figure 1, it focuses on perfect bug reports (1) which explicitly include localization information [47, page 283], (2) where the symptom (e.g., buffer overrun) is explicitly indicated by the reporter [47, page 283], and (3) which are about one of the following three simple bug types: Buffer overflow, Null Pointer dereference or memory leak [47, page 283].

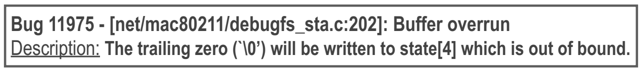

Bug 11975 - [net/mac80211/debugfs_sta.c:202]: Buffer overrun
Description: **The trailing zero (`\0') will be written to state[4] which is out of bound.**

Figure 1: Example of Linux bug report addressed by R2Fix.

R2Fix runs a straightforward supervised learning classification to identify the bug category and uses a match and transform engine (e.g., Coccinelle [68]) to generate patches. As the authors admitted, their target space represents less than 1% of bug reports in their collected dataset. Furthermore, it should be noted that, given the limited scope of the changes implemented in its fix patterns, R2Fix does not need to run tests for verifying that the generated patches do not break any functionality.

This paper. We propose to investigate the feasibility of a program repair system driven by bug reports. To that end, we make no assumption on the simplicity or semantics of the bugs that are reported, just like in state-of-the-art APR, neither on the quality of the bug reports, nor on the profile of the reporter. Eventually, we propose MIMIC, a new program repair workflow which considers a practical repair setup by imitating the fundamental steps of manual debugging. MIMIC works under the following constraint:

When a bug report is submitted to the issue tracking system, a relevant test case reproducing the bug may not be readily available.

Therefore, MIMIC is leveraged in this study to assess *to what extent an automatic program repair pipeline is feasible under the practical constraint of limited test suites.* MIMIC uses common bug reports written in natural language as the main input. Eventually, we make the following contributions:

- We present the architecture of a program repair system adapted to the constraints of software practitioners dealing with user-reported bugs. In particular, MIMIC replaces classical spectrum-based localization with Information Retrieval (IR)-based fault localization.
- We propose a strategy to prioritize patches for recommendation to developers. Indeed, given that we assume only the presence of regression test cases to validate patch candidates, many of these patches may fail on the future test cases that are relevant to the reported bugs. We order patches in the hope of presenting genuine patches first.
- We assess and discuss the performance of MIMIC against the Defects4J benchmark to compare against the state-of-the-art APR performance results in the literature. To that end, we provide a refined Defects4J benchmark for APR targeting bug reports. Bugs are carefully linked with the corresponding bug reports, and for each bug we are able to dissociate *future test cases* that were introduced after the relevant fixes.

Overall, experimental results show that there are promising research directions to further investigate towards the integration of automatic patch generation in actual software development cycles. In particular, our findings suggest that IR-fault localization errors lead less to overfitting patches than spectrum-based fault localization errors. Furthermore, MIMIC offers provides comparable results to most state-of-the-art APR tools, although it is run under the constraint that post-fix knowledge (i.e., future test cases) is not available. Finally, MIMIC's prioritization strategy tends to place more genuine/plausible patches on top of the recommendation list.

2 MOTIVATION

We motivate our work by revisiting two essential steps in automated program repair:

1. During *fault localization*, relevant program entities are identified as suspicious locations that must be changed. Commonly, state-of-the-art APR approaches leverage spectrum-based fault localization [13, 14, 28, 29, 35, 36, 40, 43, 53, 54, 56, 64, 67, 88, 97, 98], which uses execution coverage information of passing and failing test cases to predict buggy statements. We dissect the construction of 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 incompleteness 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 widely used dataset (and benchmark) in the APR literature [13, 21, 75, 89, 95, 96]. This dataset is the result of a manual curation effort: each of its 395 bugs was reproduced and included with at least one failing test case. Given that 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: when checking out a bug, the failing test cases are immediately available. 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 of the findings.

Table 1: Test case changes in fix commits of Defects4J bugs.

Test case related commits	# bugs
Commit does not alter test cases	14
Commit is inserting new test case(s) and updating previous test case(s)	62
Commit is updating previous test case(s) (without inserting new test cases)	76
Commit is inserting new test case(s) (without updating previous test cases)	243

Overall, for 96% (i.e., 381 out the 395) bugs, the relevant test cases are actually *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.

Failing test cases	# bugs
Failing test cases exist (and no future test cases are committed)	14
Failing test cases exist (but future test cases update the test scenarios)	9
Failing test cases exist (but they are fewer when considering future test cases)	4
Failing test cases exist (but they differ from future test cases which trigger the bug)	3
No failing test case exists (i.e., only future test cases trigger the bug)	365

In the APR literature, fault localization is generally performed using the GZoltar [12] testing framework, and a spectrum-based fault localization formula [93], such as Ochiai [1]. 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 spectrum-based fault localization 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* [29, 65] and *correctness* [78, 96] of the patches generated by APR tools. Notably, various studies [11, 37, 71, 78, 99] have raised concerns about overfitting patches: a typical APR technique 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 [94, 102] 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 [103] in automatic test generation (i.e., some of the automatically-generated tests can encode wrong behavior).

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 [32, 68]. In proprietary settings, Facebook has recently reported on their *Getafix* [77] tool, which automatically suggests fixes to their developers. Similarly, Ubisoft developed *Clever* [66] 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 MIMIC APPROACH

Figure 2 overviews the workflow of the proposed MIMIC 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 that the user of the program has submitted. We rely on the information tokens in this report to localize the buggy code locations.
(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 node representing the code entity in the identified suspicious code location.
(3) **Which patches are valid?** We make no assumptions on the availability of *positive test cases* [88] 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 be acceptable to the developer. We rely on heuristics to re-prioritize the validated patches in order to increase the probability of placing a genuine 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 implement, 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 developers[2] 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, MIMIC 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

[2]We rely on email addresses of committers and issue reporters to intersect users and developers

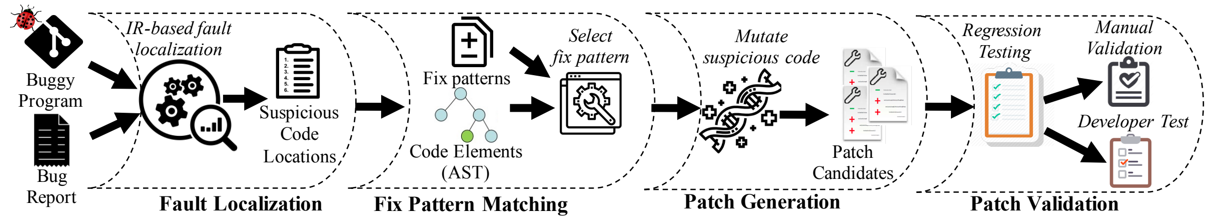


Figure 2: The MIMIC Program Repair Workflow.

Table 3: Example bug report (Defects4J Lang-7).

Issue No.	LANG-822
Summary	NumberUtils#createNumber - bad behaviour for leading "–"
Description	NumberUtils#createNumber checks for a leading "–" in the string, and returns null if found. This is documented as a work round for a bug in BigDecimal. Returning nulll is contrary to the Javadoc and the behaviour for other methods which would throw NumberFormatException. It's not clear whether the BigDecimal problem still exists with recent versions of Java. However, if it does exist, then the check needs to be done for all invocations of BigDecimal, i.e. needs to be moved to createBigDecimal.

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) [69, 84]. 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 [58, 74, 85, 90, 92, 101, 104] 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 IRFL techniques have so far focused on file-level localization, which is too coarse-grained (in comparison to spectrum-based fault localization output). Recently, Locus [90] and BLIA [101] 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 IRFL 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 IRFL tool, thus yielding a fine-grained IR-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 IRFL tool. Given that expensive extractions of tokens from a large corpus of bug reports is often necessary to tune IRFL tools [45], we selected a tool for which the authors provide datasets and pre-processed data. We use the D&C [33] as the specific implementation of file-level IRFL available online[3], which is a machine learning-based IRFL 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 MIMIC, 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 [49] 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. [49] reported that the following specific AST statement nodes were significantly more prone to be faulty than others: `IfStatements`, `ExpressionStatements`, `FieldDeclarations`, `ReturnStatements` and `VariableDeclarationStatements`. Lines 7–17 in Algorithm 1 detail the process to produce a ranked list of suspicious statements.

Algorithm 1: Statement-level IR-based Fault Localization.

```
   Input    : $br$ : a bug report
   Input    : $irTool$ : IRFL tool
   Output   : $S_{score}$ : Suspicious Statements with weight scores
1  Function main ($br$,$irTool$)
2      $F \leftarrow$ fileLocalizations ($irTool$,$br$)
3      $F \leftarrow$ selectTop ($F$,$k$)
4      $c_b \leftarrow$ bagOfTokens ($br$)            /* $c_b$: Bag of Tokens of bug report */
5      $c'_b \leftarrow$ preprocess ($c_b$)    /* tokenization,stopword removal, stemming */
6      $v_b \leftarrow$ tfIdfVectorizer($c'_b$)          /* $v_b$: Bug report Feature Vector */
7      for $f$ in $F$ do
8          $S \leftarrow$ parse($f$)                             /* $S$: List of statements */
9          for $s$ in $S$ do
10             $c_s \leftarrow$ bagOfTokens ($s$)   /* $c_s$: Bag of Tokens of statements */
11             $c'_s \leftarrow$ preprocess ($c_s$)
12             $v_s \leftarrow$ tfIdfVectorizer($c'_s$)  /* $v_s$: Statements Feature Vector */
13             /* Cosine similarity between bug report and statement   */
14             $sim_{cos} \leftarrow$ similarity$_{cosine}$ ($v_b$,$v_s$)
15             $w_{score} \leftarrow sim_{cos} \times f$.score;    /* score: Suspicious Value */
16             $W_{score}$.add($s$,$w_{score}$)
17     $S_{score} \leftarrow W_{score}$.sort()
18     return $S_{score}$
```

Algorithm 1 describes the process of our fault localization approach used in MIMIC. Top k files are selected among the returned

[3]https://github.com/d-and-c/d-and-c

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 punctuations, 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[4] is then applied before performing stemming (using the PorterStemmer [27]) 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 [29] (also referred to as fix templates [52] or program transformation schemas [21]). Several APR systems [15, 21, 29, 34, 50–52, 61, 75] 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. [29]. Concretely, we build on *kPAR* [50], 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 MIMIC.

Pattern description	used by*	Pattern description	used by*
Insert Cast Checker	Genesis	Mutate Literal Expression	SimFix
Insert Null Pointer Checker	NPEFix	Mutate Method Invocation	ELIXIR
Insert Range Checker	SOFix	Mutate Operator	jMutRepair
Insert Missed Statement	HDRepair	Mutate Return Statement	SketchFix
Mutate Conditional Expression	ssFix	Mutate Variable	CapGen
Mutate Data Type	AVATAR	Move Statement(s)	PAR
Remove Statement(s)	FixMiner		

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

```
+  if (exp instanceof T) {
       ...(T) exp...; ......
+  }
```

Figure 3: Illustration of "Insert Cast Checker" fix pattern.

For a given reported bug, once our fault localizer yields its list of suspicious statements, MIMIC iteratively attempts to select fix patterns for each statement. The selection of fix patterns is conducted in a naïve way based on the context information of each suspicious statement (i.e., all nodes in its abstract syntax tree, AST). Specifically, MIMIC 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 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, MIMIC 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. MIMIC parses the buggy statement (i.e., statement at line 302 in the file *Frequency.java*) into an AST as illustrated by Figure 5. First, MIMIC 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))`) 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, MIMIC assigns a fix pattern, calling another method with the same parameters (cf. PAR [29, page 804]), to mutate the method name, and then selects fix patterns to mutate the parameter. Furthermore, MIMIC will match fix patterns for the type and variable of the cast expression respectively and successively.

```
File: src/main/java/org/apache/commons/math/stat/Frequency.java
Line-301    public double getPct(Object v) {
Line-302        return getCumPct((Comparable<?>) v);
Line-303    }
```

Figure 4: Buggy code of Defects4J bug Math-75.

3.4 Patch Validation with Regression Testing

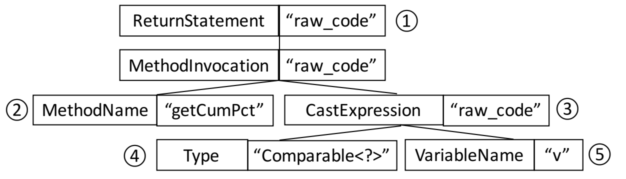
For every reported bug, fault localization followed by pattern matching and code mutation will yield a set of patch candidates. In a typical test-based APR system, these patch candidates must let the program pass all test cases (including some *positive test cases* [88], which encode the actual functional 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 MIMIC 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* [100], 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, MIMIC 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:

[4]Stop words are from the NTLK framework :https://www.nltk.org/



*"raw_code" denotes the corresponding source code at the related node position.

Figure 5: AST of bug Math-75 source code statement.

(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 [50], and is noised by the pre-set breadth-first strategy for matching fix patterns. We thus design an ordering process with a function[5], $f_{rcmd} : 2^{\mathbb{P}} \rightarrow \mathbb{P}^k$, as follows:

$$f_{rcmd}(patches) = (pri_{type} \circ pri_{susp} \circ pri_{change})(patches) \quad (1)$$

where pri_* are three heuristics-based prioritization functions used in MIMIC. f_{rcmd} takes a set of patches validated via regression testing (cf. Section 3.4) and produces an ordered sequence of patches ($f_{rcmd}(patches) = seq_{rcmd} \in \mathbb{P}^k$). 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 $pri_{change} : 2^{\mathbb{P}} \rightarrow \mathbb{P}^n$ where $n = |patches|$, $pri_{change}(patches) = [p_i, p_{i+1}, p_{i+2}, \cdots]$ and holds $\forall p \in patches, C_{change}(p_i) \leq C_{change}(p_{i+1})$. Here, $C_{change}(p)$ is a function that counts the number of deleted and inserted AST nodes by the change actions of p.
(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 $C_{change}(p_i) == C_{change}(p_{i+1})$, pri_{susp} re-orders the two patch candidates. We define $pri_{susp} : \mathbb{P}^n \rightarrow \mathbb{P}^n$ such that $pri_{susp}(seq_{change}) = [\cdots, p_i, p_{i+1}, \cdots]$ holds $S_{susp}(p_i) \geq S_{susp}(p_{i+1})$, where seq_{change} is the result of pri_{change} and S_{susp} returns a suspicious score of the statement that a given patch p_i 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, MIMIC systematically under-prioritizes their generated patches against any other patches, although among themselves the ranking obtained so far (through pri_{change} and pri_{susp}) 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 $pri_{type} : \mathbb{P}^n \rightarrow \mathbb{P}^k$, 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 $j_4 > j_1, j_4 > j_2, j_4 > j_3$. This prioritization function takes the result of pri_{susp} and returns another sequence $[p_i, p_{i+1}, p_{i+2}, \cdots]$ that holds $\forall p_i, D_{type}(p_i) \geq D_{type}(p_{i+1})$. Here, D_{type} is defined as $D_{type} : 2^{\mathbb{P}} \rightarrow \{j_1, j_2, j_3, j_4\}$ and determines how a patch p_i has been generated as defined above. From the ordered sequence, the function returns the leftmost (i.e., top) k patches as a result.

[5]The domain of the function is a power set $2^{\mathbb{P}}$, and the co-domain ($\mathbb{P}^k$) is a k-dimensional vector space [31] where k is the maximum number of recommended patches, and $\mathbb{P}$ denotes the set of all generated patches.

4 EXPERIMENTAL SETUP

We now provide details on the experiments that we carry out to assess the MIMIC patch generation pipeline for user-reported bugs. Notably, we discuss the dataset and benchmark, some implementation details before enumerating the research questions.

4.1 Dataset & Benchmark

To evaluate MIMIC we propose to rely on the Defects4J [25] dataset which is widely used as a benchmark in the Java APR literature. Nevertheless, given that Defects4J does not provide direct links to the bug reports that are associated with the benchmark bugs, we must undertake a *fairly accurate* bug linking task [82]. Furthermore, to realistically evaluate MIMIC, we must reorganize the dataset test suites to accurately simulate the context at the time the bug report is submitted by users.

4.1.1 Bug linking. To identify the bug report describing a given bug in the Defects4J dataset we focus on recovering the links between the bug fix commits and bug reports from the issue tracking system. Unfortunately, projects Joda-Time, JFreeChart and Closure have migrated their source code repositories and issue tracking systems into GitHub without a proper reassignment of bug report identifiers. 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. [17] and Thomas et al. [82] 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.

4.2 Implementation Choices

During implementation, we have made the following parameter choices in the MIMIC 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 [60], ACS [97], ssFix [95] and CapGen [89] among others).

4.3 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 MIMIC 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 fine-grained IRFL implementation against the classical spectrum-based localization.
- RQ2 [Overfitting] : *To what extent does IR-based fault localization point to locations that are less subject to overfitting?* In particular, we study the impact on the *overfitting* problem that incomplete test suites generally carry.
- RQ3 [Patch ordering] : *What is the effectiveness of* MIMIC*'s patch ordering strategy?* In particular, we investigate the overall workflow of MIMIC, by re-simulating the real-world cases of software maintenance cycle when a bug is reported: future test cases are not available for patch validation.

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 MIMIC. As recently thoroughly studied by Liu et al. [50], an APR tool should not be expected to fix a bug that current fault localization systems fail to localize. Nevertheless, with MIMIC, we must demonstrate that our fine-grained IRFL offers comparable performance with spectrum-based fault localization (SFL) tools used in the APR literature.

Table 5 provides performance measurements on the localization of bugs. Spectrum-based fault localization 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 [50, 57], we consider that a bug is localized if any of the buggy statements is localized.

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

(171 bugs)		Top-1	Top-10	Top-50	Top-100	Top-200	All
IRFL		25	72	102	117	121	139
SFL	**GZ**$_{v1}$	26	75	106	110	114	120
	GZ$_{v2}$	23	79	119	135	150	156

† GZ$_{v1}$ and GZ$_{v2}$ refer to GZoltar 0.1.1 and 1.6.0 respectively, which are widely used in APR systems for Java programs.

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 SFL 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.

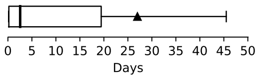


Figure 6: Distribution of elapsed time (in days) between bug report submission and test case attachment.

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.

Table 6: Fault localization performance.

GZoltar + Ochiai (395 bugs)	Top-1	Top-10	Top-50	Top-100	Top-200	All
without future tests	5	10	17	17	19	20
with future tests	45	140	198	214	239	263

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 SFL 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.

Fine-grained IR-based fault localization in MIMIC *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 [99].

We perform patch generation experiments to investigate the impact of localization bias. We compare our IRFL implementation against commonly-used SFL 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 in the suspicious statements yielded the fault localization system (either IRFL or SFL), 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* [71] (i.e., that passes all test cases) and *genuine* (a.k.a. correctness [71], i.e., that is eventually manually validated as semantically equivalent to the developer fix) patches that can be found by considering top-k statements returned by the fault localizer.

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

	Lang	Math	Total
IRFL Top-1	1/4	3/4	4/8
SFL Top-1	1/4	**6/8**	7/12
IRFL Top-5	**3**/6	7/14	10/20
SFL Top-5	2/**7**	**11/17**	13/24
IRFL Top-10	**4/9**	9/17	13/26
SFL Top-10	4/11	**16/27**	20/38
IRFL Top-20	**7/12**	9/18	16/30
SFL Top-20	4/11	**18/30**	22/41
IRFL Top-50	**7/15**	10/22	17/37
SFL Top-50	4/13	**19/34**	23/47
IRFL Top-100	**8/18**	10/23	18/41
SFL Top-100	5/14	**19/36**	24/50
IRFL All	**11/19**	10/25	21/44
SFL All	5/14	**19/36**	24/50

* We indicate x/y numbers of patches: x is the number of bugs for which a *genuine* patch is generated; y is the number of bugs for which a *plausible* patch is generated.

Overall, we find that IRFL and SFL localization information lead to similar repair performance in terms of the number of fixed bugs (either plausibly or genuinely). Actually IRFL-supported APR outperforms SFL-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 SFL output (although assuming the availability of the bug triggering test cases to run the SFL 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 genuine.

	Localization Error	Pattern Prioritization	Lack of Fix ingredients
w/ IRFL	6	1	16
w/ SFL	15	1	10

*A plausible patch passes all test cases, but may not be semantically equivalent to developer patch (i.e., genuine). 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 genuine, 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 [50]. Nevertheless, we note that patches generated based on SFL identified code locations suffer more of fault localization bias: 15 of the 26 (= 50−24) plausible patches are concerned by this issue.
(2) Pattern prioritization failures may also lead to plausible patches: while a genuine 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 found to be matching the statement 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 [48]).

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 `MIMIC` 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 genuine/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 `MIMIC`: we present the number of bugs for which a genuine/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 genuine (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 *genuineness* (also referred to as *correctness* [71]) 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 MIMIC 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 genuine.

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

Recommendation rank	Top-1	Top-5	Top-10	Top-20	All
without patch re-prioritization	3/3	4/5	6/10	6/10	13/27
with patch re-prioritization	3/4	8/13	9/14	10/15	13/27

* x/y: x is the number of bugs for which a *genuine* 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 MIMIC (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 MIMIC can propose a genuine/plausible patch. We consider three scenarios with MIMIC:

(1) [MIMIC$_{top5}$] - developers will be provided with only top 5 recommended patches which have been validated only with regression tests: in this case, MIMIC outperforms about half of the state-of-the-art in terms of numbers bugs fixed with both plausible or genuine patches.
(2) [MIMIC$_{all}$] - 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 MIMIC.
(3) [MIMIC$_{opt}$] - 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, MIMIC outperforms all state-of-the-art, except SimFix [23] 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. MIMIC$_{opt}$ 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.

MIMIC *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.*

5.3.3 Properties of MIMIC*'s patches.* In Table 11, we characterize the genuine and plausible patches recommended by MIMIC$_{top5}$. Overall, update and insert changes have been successful; most patches affect a single statement, and impact precisely an expression entity within a statement.

5.3.4 Diversity of MIMIC*'s fixed bugs.* Finally, in Table 12 we dissect the nature of the bugs for which MIMIC$_{top5}$ is able to recommend a genuine 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 10: MIMIC vs state-of-the-art APR tools.

APR tool	**Lang***	**Math***	**Total***
jGenProg [60]	0/0	5/18	5/18
jKali [60]	0/0	1/14	1/14
jMutRepair [60]	0/1	2/11	2/12
HDRepair [39]	2/6	4/7	6/13
Nopol [98]	3/7	1/21	4/28
ACS [97]	3/4	12/16	15/20
ELIXIR [75]	8/12	12/19	20/31
JAID [13]	1/8	1/8	2/16
ssFix [95]	5/12	10/26	15/38
CapGen [89]	5/5	12/16	17/21
SketchFix [21]	3/4	7/8	10/12
FixMiner [34]	2/3	12/14	14/17
LSRepair [48]	8/14	7/14	15/28
SimFix [23]	9/13	**14/26**	**23**/39
kPAR [50]	1/8	7/18	8/26
AVATAR [51]	5/11	6/13	11/24
MIMIC$_{opt}$	**11/19**	10/25	21/**44**
MIMIC$_{all}$	6/11	7/16	13/27
MIMIC$_{top5}$	3/7	5/6	8/13

* x/y: x is the number of bugs for which a *genuine* patch is generated; y is the number of bugs for which a *plausible* patch is generated.
MIMIC$_{opt}$: the version of MIMIC where available test cases are relevant to the bugs.
MIMIC$_{all}$: all recommended patches are considered.
MIMIC$_{top5}$: only top 5 recommended patches are considered.

Table 11: Change properties of MIMIC's genuine patches.

Change action	#bugs*	Impacted statement(s)	#bugs*	Granularity	#bugs*
Update	5/7	Single-statement	8/12	Statement	1/2
Insert	3/5	Multiple-statement	0/1	Expression	7/11
Delete	0/1				

* x/y $\longrightarrow$ for x bugs the patches are genuine, while for y bugs they are plausible.

Table 12: Dissection of bugs successfully fixed by MIMIC.

Patch Type	Defect4J Bug ID	Issue ID	Root Cause	Priority
G	L-6	LANG-857	String index out of bounds exception	Minor
G	L-24	LANG-664	Wrong behavior due missing condition	Major
G	L-57	LANG-304	Null pointer exception	Major
G	M-15	MATH-904	Double precision floating point format error	Major
G	M-34	MATH-779	Missing "read only access" to internal list	Major
G	M-35	MATH-776	Range check	Major
G	M-57	MATH-546	Wrong variable type truncates double value	Minor
G	M-75	MATH-329	Method signature mismatch	Minor
P	L-13	LANG-788	Serialization error in primitive types	Major
P	L-21	LANG-677	Wrong Date Format in comparison	Major
P	L-45	LANG-419	Range check	Minor
P	L-58	LANG-300	Number formatting error	Major
P	M-2	MATH-1021	Integer overflow	Major

"G" denotes genuine 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 [47] 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* [88]. Our experiments on the widely-used Defects4J 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 MIMIC, several insights can be drawn:
Test cases can be buggy: During manual analysis of results, we have noted that MIMIC actually fails to generate genuine patches for three bugs (namely, Math-5, Math-59 and Math-65) because even the regression test cases were buggy. Figure 7 illustrates the

example of bug Math-5 where the bug fix 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.

```
// Patched Source Code:
--- a/src/main/java/org/apache/commons/math3/complex/Complex.java
+++ b/src/main/java/org/apache/commons/math3/complex/Complex.java
@@ -304,3 +304,3 @@ public class Complex implements FieldElement<
       Complex>, Serializable  {
          if (real == 0.0 && imaginary == 0.0) {
-             return NaN;
+             return INF;
          }

// Patched Test Case:
--- a/src/test/java/org/apache/commons/math3/complex/ComplexTest.java
+++ b/src/test/java/org/apache/commons/math3/complex/ComplexTest.java
@@ -332,4 +332,4 @@ public class ComplexTest {
     @Test
     public void testReciprocalZero() {
-        Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.NaN);
+        Assert.assertEquals(Complex.ZERO.reciprocal(), Complex.INF);
     }
```

Figure 7: Patched source code and test case of fixing Math-5.

Bug reports deserve more interest: With MIMIC, 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 genuine. 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, MIMIC 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 MIMIC 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 [73].

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. [105] 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. [20] on measuring bug report significance).

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

Proj.	Unique Bug Reports	w/ Patch Attached	Average Comments	w/ Stack Traces	w/ Hints	w/ Code Blocks
Lang	62	11	4.53	4	62	31
Math	100	23	5.15	5	92	51

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

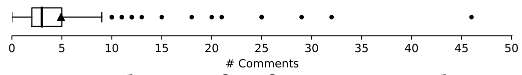


Figure 8: Distribution of # of comments per bug report.

Another threat to external validity relates to the diversity of the fix patterns used in this study. MIMIC 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. [49].

Additionally, another internal threat is in our patch generation steps: MIMIC 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: For our approach and experiments, 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. As stated in a recent study [50], fault localization is a critical task affecting the effectiveness of automated program repair. Several techniques have been proposed [69, 84, 93] and they use different information such as spectrum [3], text [90], slice [59], and statistics [46]. The first two types of techniques are widely studies in the community. Spectrum-based fault localization (SFL) techniques [2, 24] are widely adopted in APR pipelines since they identify bug positions at a fine-grained level (i.e., statements). However, they have limitations on localizing buggy locations since it highly relies on the test suite [50]. Information retrieval based fault localization (IRFL) [45] 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 further 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 [41, 55]. To improve repair performance, many APR systems have been explored to address the search space problem by using different information and approaches: stochastic mutation [42, 88], synthesis [54, 97, 98], pattern [15, 21, 23, 29, 35, 39, 51–53, 75], contract [13, 87], symbolic execution [67], learning [9, 18, 56, 72, 79, 91], and donor code searching [28, 64]. In this paper, patch generation is implemented with fix patterns presented in the literature since it may make the generated patches more robust [76].

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 [29, 39, 88]. However, these patches could be overfitting [37, 71] and even worse than the bug [78]. Since then, APR systems are evaluated with the precision of generating correct patches [23, 51, 89, 97]. Recently, researchers explore automated frameworks that can identify patch correctness for APR systems automatically [38, 96]. 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 `MIMIC`, 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* [88] 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 `MIMIC` can generate and recommend priority genuine (and more plausible) patches for a diverse set of user-reported bugs. The repair performance of `MIMIC` is even found to be comparable to that of the majority of test-based APR systems on the Defects4J dataset.

We open source `MIMIC`'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/fse19/mimic

REFERENCES

[1] Rui Abreu, Arjan JC Van Gemund, and Peter Zoeteweij. 2007. On the accuracy of spectrum-based fault localization. In *Proceedings of TAICPART-MUTATION.* IEEE, 89–98.

[2] Rui Abreu, Peter Zoeteweij, Rob Golsteijn, and Arjan JC Van Gemund. 2009. A practical evaluation of spectrum-based fault localization. *JSS* 82, 11 (2009), 1780–1792.

[3] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. 2009. Spectrum-based multiple fault localization. In *Proceedings of the 24th ASE.* IEEE, 88–99.

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *Proceedings of the 18th CASCON.* ACM, 23.

[5] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *Proceedings of the 28th ICSE.* ACM, 361–370.

[6] Olga Baysal, Michael W Godfrey, and Robin Cohen. 2009. A bug you like: A framework for automated assignment of bugs. In *Proceedings of the 17th ICPC.* IEEE, 297–298.

[7] Kent Beck. 2003. *Test-driven development: by example.* Addison-Wesley Professional.

[8] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. 2015. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 10th FSE.* ACM, 179–190.

[9] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-symbolic program corrector for introductory programming assignments. In *Proceedings of the 40th ICSE.* ACM, 60–70.

[10] Tegawendé F Bissyandé. 2015. Harvesting Fix Hints in the History of Bugs. *arXiv preprint arXiv:1507.05742* (2015).

[11] Marcel Böhme, Ezekiel O Soremekun, Sudipta Chattopadhyay, Emamurho Ugherughe, and Andreas Zeller. 2017. Where is the bug and how is it fixed? an experiment with practitioners. In *Proceedings of the 11th FSE.* ACM, 117–128.

[12] José Campos, André Riboira, Alexandre Perez, and Rui Abreu. 2012. Gzoltar: an eclipse plug-in for testing and debugging. In *Proceedings of the 27th ASE.* IEEE/ACM, 378–381.

[13] Liushan Chen, Yu Pei, and Carlo A Furia. 2017. Contract-based program repair without the contracts. In *Proceedings of the 32nd ASE.* IEEE, 637–647.

[14] Zack Coker and Munawar Hafiz. 2013. Program transformations to fix C integers. In *Proceedings of the 35th ICSE.* IEEE/ACM, 792–801.

[15] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *Proceedings of the 24th SANER.* IEEE, 349–358.

[16] Mattia Fazzini, Martin Prammer, Marcelo d'Amorim, and Alessandro Orso. 2018. Automatically translating bug reports into test cases for mobile apps. In *Proceedings of the 27th ISSTA.* ACM, 141–152.

[17] Michael Fischer, Martin Pinzger, and Harald Gall. 2003. Populating a release history database from version control and bug tracking systems. In *Proceeding of the 19th ICSM.* IEEE, 23–32.

[18] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the 31st AAAI.* AAAI Press, 1345–1351.

[19] Kim Herzig, Sascha Just, and Andreas Zeller. 2013. It's not a bug, it's a feature: how misclassification impacts bug prediction. In *Proceedings of the 35th ICSE.* IEEE Press, 392–401.

[20] Pieter Hooimeijer and Westley Weimer. 2007. Modeling bug report quality. In *Proceedings of the 22nd ASE.* ACM, 34–43.

[21] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th ICSE.* ACM, 12–23.

[22] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *Proceedings of the the 7th ESEC/FSE.* ACM, 111–120.

[23] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code. In *Proceedings of the 27th ISSTA.* ACM, 298–309.

[24] James A Jones and Mary Jean Harrold. 2005. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th ASE.* ACM, 273–282.

[25] René Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 23rd ISSTA.* ACM, 437–440.

[26] Natalia Juristo Juzgado, Ana María Moreno, and Wolfgang Strigel. 2006. Guest editors' introduction: Software testing practices in industry. *IEEE Software* 23, 4 (2006), 19–21.

[27] Wahiba Ben Abdessalem Karaa and Nidhal Gribâa. 2013. Information retrieval with porter stemmer: a new version for English. In *Advances in computational science, engineering and information technology.* Springer, 243–254.

[28] Yalin Ke, Kathryn T Stolee, Claire Le Goues, and Yuriy Brun. 2015. Repairing programs with semantic code search (t). In *Proceedings of the 30th ASE.* IEEE, 295–306.

[29] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *Proceedings of the 35th ICSE*. IEEE, 802–811.

[30] Pavneet Singh Kochhar, Tegawendé F Bissyandé, David Lo, and Lingxiao Jiang. 2013. An empirical study of adoption of software testing in open source projects. In *Proceedings of the 13th QRS*. IEEE, 103–112.

[31] A. N. Kolmogorov and S. V. Fomin. 1999. *Elements of the Theory of Functions and Functional Analysis* (dover books on mathematics edition ed.). Dover Publications, Mineola, NY.

[32] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2017. Impact of tool support in patch construction. In *Proceedings of the 26th ISSTA*. ACM, 237–248.

[33] Anil Koyuncu, Tegawendé F Bissyandé, Dongsun Kim, Kui Liu, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2019. D&C: A Divide-and-Conquer Approach to IR-based Bug Localization. *arXiv preprint arXiv:1902.02703* (2019).

[34] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2018. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *arXiv preprint arXiv:1810.01791* (2018).

[35] Xuan-Bach D Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 11th FSE*. ACM, 593–604.

[36] Xuan-Bach D Le, Quang Loc Le, David Lo, and Claire Le Goues. 2016. Enhancing automated program repair with deductive verification. In *Proceedings of the 32nd ICSME*. IEEE, 428–432.

[37] Xuan Bach D Le, Ferdian Thung, David Lo, and Claire Le Goues. 2018. Overfitting in semantics-based automated program repair. *EMSE Journal* (2018), 1–27.

[38] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, and Shanping Li. 2019. On Reliability of Patch Correctness Assessment. In *Proceedings of the 41st ICSE*.

[39] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd SANER*, Vol. 1. IEEE, 213–224.

[40] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *Proceedings of the 23rd SANER*, Vol. 1. IEEE, Suita, Osaka, Japan, 213–224.

[41] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *Proceedings of the 34th ICSE*. IEEE, 3–13.

[42] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *TSE* 38, 1 (2012), 54–72.

[43] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A Generic Method for Automatic Software Repair. *TSE* 38, 1 (2012), 54–72.

[44] Claire Le Goues and Westley Weimer. 2009. Specification mining with few false positives. In *Proceedings of the 15th TACAS*. Springer, 292–306.

[45] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. 2018. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ISSTA*. ACM, 61–72.

[46] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *Proceedings of the 26th PLDI*. ACM, 15–26.

[47] Chen Liu, Jinqiu Yang, Lin Tan, and Munawar Hafiz. 2013. R2Fix: Automatically generating bug fixes from bug reports. In *Proceedings of the 6th ICST*. IEEE, 282–291.

[48] Kui Liu, Koyuncu Anil, Kisub Kim, Dongsun Kim, and Tegawendé F. Bissyandé. 2018. LSRepair: Live Search of Fix Ingredients for Automated Program Repair. In *Proceedings of the 25th APSEC*. IEEE, 658–662.

[49] Kui Liu, Dongsun Kim, Anil Koyuncu, Li Li, Tegawendé F Bissyandé, and Yves Le Traon. 2018. A closer look at real-world patches. In *Proceedings of the 34th ICSME*. IEEE, 275–286.

[50] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *Proceedings of the 12th ICST*. IEEE.

[51] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR : Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *Proceedings of the 26th SANER*. IEEE.

[52] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *Proceedings of the 25th SANER*. IEEE, 118–129.

[53] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 11th FSE*. ACM, 727–739.

[54] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *Proceedings of the 10th FSE*. ACM, 166–178.

[55] Fan Long and Martin Rinard. 2016. An analysis of the search spaces for generate and validate patch generation systems. In *Proceedings of the 38th ICSE*. ACM, 702–713.

[56] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd POPL*. ACM, 298–312.

[57] Lucia LUCIA, Ferdian Thung, David Lo, and Lingxiao Jiang. 2012. Are Faults Localizable?. In *Proceedings of the 9th MSR*. 74–77.

[58] Stacy K Lukins, Nicholas A Kraft, and Letha H Etzkorn. 2010. Bug localization using latent Dirichlet allocation. *IST* 52, 9 (2010), 972–990.

[59] Xiaoguang Mao, Yan Lei, Ziying Dai, Yuhua Qi, and Chengsong Wang. 2014. Slice-based statistical fault localization. *JSS* 89 (2014), 51–62.

[60] Matias Martinez and Martin Monperrus. 2016. Astor: A program repair library for java. In *Proceedings of the 25th ISSTA*. ACM, 441–444.

[61] Matias Martinez and Martin Monperrus. 2018. Ultra-Large Repair Search Space with Automatically Mined Templates: The Cardumen Mode of Astor. In *Proceedings of the 10th SSBSE*. Springer, 65–86.

[62] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings 6th MSR*. IEEE, 131–140.

[63] Sergey Mechtaev, Manh-Dung Nguyen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2018. Semantic Program Repair Using a Reference Implementation. In *Proceedings of the 40th ICSE*. ACM, 298–309.

[64] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *Proceedings of the 37th ICSE*. IEEE Press, 448–458.

[65] Martin Monperrus. 2014. A critical review of automatic patch generation learned from human-written patches: essay on the problem statement and the evaluation of automatic software repair. In *Proceedings of the 36th ICSE*. ACM, 234–242.

[66] Mathieu Nayrolles and Abdelwahab Hamou-Lhadj. 2018. CLEVER: combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects. In *Proceedings of the 15th MSR*. ACM, 153–164.

[67] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: program repair via semantic analysis. In *Proceedings of the 35th ICSE*. IEEE, 772–781.

[68] Yoann Padioleau, Julia Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of 3rd EuroSys*, Vol. 42. ACM, 247–260.

[69] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers?. In *Proceedings of the 20th ISSTA*. ACM, 199–209.

[70] Jean Petrić, Tracy Hall, and David Bowes. 2018. How Effectively Is Defective Code Actually Tested?: An Analysis of JUnit Tests in Seven Open Source Systems. In *Proceedings of the 14th PROMISE*. ACM, 42–51.

[71] Zichao Qi, Fan Long, Sara Achour, and Martin Rinard. 2015. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *Proceedings of the 24th ISSTA*. ACM, 24–36.

[72] Reudismam Rolim, Gustavo Soares, Loris D'Antoni, Oleksandr Polozov, Sumit Gulwani, Rohit Gheyi, Ryo Suzuki, and Björn Hartmann. 2017. Learning syntactic program transformations from examples. In *Proceedings of the 39th ICSE*. IEEE/ACM, 404–415.

[73] Ripon Saha, Yingjun Lyu, Wing Lam, Hiroaki Yoshida, and Mukul Prasad. 2018. Bugs.jar: a large-scale, diverse dataset of real-world java bugs. In *Proceedings of the 15th MSR*. IEEE, 10–13.

[74] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. 2013. Improving bug localization using structured information retrieval. In *Proceedings of the 28th ASE*. IEEE, 345–355.

[75] Ripon K Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R Prasad. 2017. ELIXIR: Effective object-oriented program repair. In *Proceedings of the 32nd ASE*. IEEE, 648–659.

[76] Eric Schulte, Zachary P Fry, Ethan Fast, Westley Weimer, and Stephanie Forrest. 2014. Software mutational robustness. *Genetic Programming and Evolvable Machines* 15, 3 (2014), 281–312.

[77] Andrew Scott, Johannes Bader, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *arXiv preprint arXiv:1902.06111* (2019).

[78] Edward K Smith, Earl T Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? Overfitting in automated program repair. In *Proceedings of the 10th FSE*. ACM, 532–543.

[79] Mauricio Soto and Claire Le Goues. 2018. Using a probabilistic model to predict bug fixes. In *Proceedings of the 25th SANER*. IEEE, 221–231.

[80] Chengnian Sun, David Lo, Siau-Cheng Khoo, and Jing Jiang. 2011. Towards more accurate retrieval of duplicate bug reports. In *Proceedings of the 26th ASE*. IEEE Computer Society, 253–262.

[81] Chengnian Sun, David Lo, Xiaoyin Wang, Jing Jiang, and Siau-Cheng Khoo. 2010. A discriminative model approach for accurate duplicate bug report retrieval. In *Proceedings of the 32nd ICSE*. ACM, 45–54.

[82] Stephen W Thomas, Meiyappan Nagappan, Dorothea Blostein, and Ahmed E Hassan. 2013. The impact of classifier configuration and classifier combination on bug localization. *TSE* 39, 10 (2013), 1427–1443.

[83] Simon Urli, Zhongxing Yu, Lionel Seinturier, and Martin Monperrus. 2018. How to design a program repair bot?: insights from the repairnator project. In *Proceedings of the 40th ICSE*. ACM, 95–104.

[84] Qianqian Wang, Chris Parnin, and Alessandro Orso. 2015. Evaluating the usefulness of IR-based fault localization techniques. In *Proceedings of the 24th ISSTA*. ACM, 1–11.

[85] Shaowei Wang and David Lo. 2014. Version History, Similar Report, and Structure: Putting Them Together for Improved Bug Localization. In *Proceedings of*

the 22nd ICPC. ACM, 53–63.

[86] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. 2008. An approach to detecting duplicate bug reports using natural language and execution information. In *Proceedings of the 30th ICSE*. ACM, 461–470.

[87] Yi Wei, Yu Pei, Carlo A Furia, Lucas S Silva, Stefan Buchholz, Bertrand Meyer, and Andreas Zeller. 2010. Automated fixing of programs with contracts. In *Proceedings of the 19th ISSTA*. ACM, 61–72.

[88] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *Proceedings of the 31st ICSE*. IEEE, 364–374.

[89] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th ICSE*. IEEE/ACM, 1–11.

[90] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. 2016. Locus: Locating bugs from software changes. In *Proceedings of the 31st ASE*. IEEE, 262–273.

[91] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and Transforming Program Repair Ingredients via Deep Learning Code Similarities. In *Proceedings of the 26th SANER*. IEEE.

[92] Chu-Pan Wong, Yingfei Xiong, Hongyu Zhang, Dan Hao, Lu Zhang, and Hong Mei. 2014. Boosting Bug-Report-Oriented Fault Localization with Segmentation and Stack-Trace Analysis. In *Proceedings of the 30th ICSME*. IEEE, 181–190.

[93] W Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* 42, 8 (2016), 707–740.

[94] Qi Xin and Steven P Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ISSTA*. ACM, 226–236.

[95] Qi Xin and Steven P Reiss. 2017. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32nd ASE*. IEEE/ACM, 660–670.

[96] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th ICSE*. ACM, 789–799.

[97] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *Proceedings of the 39th ICSE*. IEEE/ACM, 416–426.

[98] Jifeng Xuan, Matias Martinez, Favio DeMarco, Maxime Clement, Sebastian Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic repair of conditional statement bugs in java programs. *TSE* 43, 1 (2017), 34–55.

[99] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *Proceedings of the 11th FSE*. ACM, 831–841.

[100] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *STVR* 22, 2 (2012), 67–120.

[101] Klaus Changsun Youm, June Ahn, and Eunseok Lee. 2017. Improved bug localization based on code change histories and bug reports. *IST* 82 (2017), 177–192.

[102] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2017. Test case generation for program repair: A study of feasibility and effectiveness. *arXiv preprint arXiv:1703.00198* (2017).

[103] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2018. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *EMSE Journal* (2018), 1–35.

[104] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *Proceedings of the 34th ICSE*. IEEE, 14–24.

[105] Thomas Zimmermann, Rahul Premraj, Nicolas Bettenburg, Sascha Just, Adrian Schroter, and Cathrin Weiss. 2010. What makes a good bug report? *TSE* 36, 5 (2010), 618–643.