# Better Automatic Program Repair by Using Bug Reports and Tests Together

Manish Motwani and Yuriy Brun
University of Massachusetts
Amherst, Massachusetts 01003-9264, USA
{mmotwani, brun}@cs.umass.edu

***Abstract*—Automated program repair is already deployed in industry, but concerns remain about repair quality. Recent research has shown that one of the main reasons repair tools produce incorrect (but seemingly correct) patches is imperfect fault localization (FL). This paper demonstrates that combining information from natural-language bug reports and test executions when localizing faults can have a significant positive impact on repair quality. For example, existing repair tools with such FL are able to correctly repair 7 defects in the Defects4J benchmark that no prior tools have repaired correctly.**

**We develop, Blues, the first information-retrieval-based, statement-level FL technique that requires no training data. We further develop RAFL, the first unsupervised method for combining multiple FL techniques, which outperforms a supervised method. Using RAFL, we create SBIR by combining Blues with a spectrum-based (SBFL) technique. Evaluated on 815 real-world defects, SBIR consistently ranks buggy statements higher than its underlying techniques.**

**We then modify three state-of-the-art repair tools, Arja, SequenceR, and SimFix, to use SBIR, SBFL, and Blues as their internal FL. We evaluate the quality of the produced patches on 689 real-world defects. Arja and SequenceR significantly benefit from SBIR: Arja using SBIR correctly repairs 28 defects, but only 21 using SBFL, and only 15 using Blues; SequenceR using SBIR correctly repairs 12 defects, but only 10 using SBFL, and only 4 using Blues. SimFix, (which has internal mechanisms to overcome poor FL), correctly repairs 30 defects using SBIR and SBFL, but only 13 using Blues. Our work is the first investigation of simultaneously using multiple software artifacts for automated program repair, and our promising findings suggest future research in this directions is likely to be fruitful.**

## I. Introduction

Automated program repair (APR) aims to reduce the cost of fixing bugs by automatically producing patches [27], [51]. APR tools have been successful enough to be used in industry [7], [66], [72], [41]. Unfortunately, repair tools patch only a small fraction of defects correctly [72], [71], [86] and industrial deployments require significant manual oversight. Recent studies show that accuracy of the fault localization (FL) used by APR has a significant effect on APR's success [2], [60], [35], [96], [6], [107], and manually improving FL can correctly patch more defects [2], [62]. Some APR tools, such as SimFix [36] use tool-specific methods to address inaccurate FL; however, these methods are tightly coupled to the specific repair technique and not reusable by other tools.

Existing APR techniques use either developer-written test suites or natural-language bug reports. For the former, spectrum-based fault localization (SBFL) executes the tests and collects coverage information to identify suspicious statements. For the latter, information-retrieval-based fault localization (IRFL) computes suspiciousness from the similarity between bug reports and program source. The defects these two types of APR tools repair tend to be complementary: For example, IRFL-based iFixR patches defects that 16 SBFL-based repair tools cannot, and vice versa [44]. Further, combining multiple FL techniques can improve localization [117], [53], [48]. We, therefore, hypothesize that combining SBFL and IRFL can improve APR. To test this hypothesis, we develop a novel IRFL technique and a novel method for combining FL techniques in an unsupervised fashion, and evaluate an SBFL, our IRFL, and the combined techniques in three state-of-the-art APR tools that have varied sensitivity to FL accuracy.

> Our main contribution is SBIR, a novel, reusable FL technique that combines bug reports and tests. The use of SBIR in APR is the first instance of APR simultaneously using multiple software artifacts, suggesting a promising new research direction. Our main finding is that the answer to the question "Does FL that combines bug reports and tests improve APR performance?" is a resounding yes, for many APR techniques. For example, on the latest Defects4J benchmark, we correctly repair 7 defects that none of 14 prior APR tools could repair correctly [60].

**Contributions on using bug-reports.** We create Blues, (Section II-A), the first reusable, APR-agnostic, unsupervised, statement-level IRFL technique that localizes defects using bug reports. Prior IRFL techniques are either file- or method-level [115], [82], [99], [111], [98], [43], or is the technique used internally by iFixR [44]. iFixR's FL requires hard-to-get training data and is tightly coupled to its APR implementation [44]. Unlike iFixR's FL, Blues can localize defects to all 57 kinds of Java AST expressions, (iFixR only handles 5 [44]). We empirically demonstrate that Blues outperforms iFixR's FL (Section III-B).

**Using tests.** Our SBFL technique is not novel. We implement SBFL using the latest version (v1.7.2) of GZoltar, and the Ochiai ranking strategy, which is one of the most effective ranking strategies in object-oriented programs [104], [117], and is used by most test-suite-based repair tools [60] (Section II-B).

**Contributions on using bug reports and tests together.** To combine FL techniques, we develop RAFL, (Section II-C), a novel approach inspired by search-based software engineering [31] that uses rank aggregation algorithms [57] to combine multiple ranked lists of top-k suspicious statements obtained by different FL techniques. While RAFL can combine any FL techniques, we focus on combining SBFL and IRFL, which are used separately by existing repair tools. We use RAFL to develop SBIR that uses the cross-entropy Monte Carlo algorithm [81] and the Spearman Footrule distance [9] to combine our SBFL and Blues. We evaluate our SBFL, Blues, and SBIR on 815 real-world defects in the Defects4J (v2.0) [26] benchmark (out of the benchmark's 835 defects, 18 have no bug reports, and 2 have irrelevant test execution information) and find that SBIR consistently outperforms the underlying techniques (Section III-C1). While one could use existing *supervised* combining FL techniques (e.g., CombineFL [117], DeepFL [53], Fluccs [87], Savant [48], Multric [104], and TraPT [54]), our study elects to use a new, *unsupervised* method because the prior techniques were trained on Defects4J and thus cannot be applied to an evaluation on Defects4J. Retraining the supervised techniques poses complex technical challenges, requires a large, independent, annotated dataset that simply doesn't exist today, and does not guarantee previously observed performance. We demonstrate that our unsupervised technique outperforms existing supervised ones (Section III-C2).

> Importantly, existing supervised methods for both IRFL and for combining multiple FL techniques require extensive training data, which is expensive (sometimes prohibitively so) to obtain. Our evaluation shows that our *unsupervised* methods (Blues and RAFL) consistently perform as well as or better than the supervised methods, without needing the expensive training data.

**Contributions on effect on APR.** To study the effect of combining FL on repair quality (Section III-D), we select Arja [113], SequenceR [14], and SimFix [36], three state-of-the-art APR tools that have varied FL sensitivity [62], are applicable to general defects, use varied repair approaches, and have public implementations. We evaluate these tools using our SBFL, Blues, and SBIR FL techniques on the 689 single-file-edit defects in the Defects4J (v2.0) benchmark, and find that SBIR enables APR to repair more defects correctly. For tools that have been shown to be more sensitive to FL [62], SBIR significantly improves patch quality (Section III-E).

Our evaluation answers four research questions:

> **RQ1. Does Blues localize defects better than existing approaches?** Yes. Blues consistently ranks buggy statements higher than state-of-the-art iFixR's supervised IRFL technique (Section III-B).
>
> **RQ2. Does SBIR improve FL over the techniques it is composed of?** Yes. SBIR consistently ranks buggy statements higher than its underlying SBFL and Blues (Section III-C1).
>
> **RQ3. Does SBIR outperform state-of-the-art FL?** Yes. SBIR consistently ranks buggy statements higher than 9 standalone FL techniques and an existing supervised FL-combining method (Section III-C2).
>
> **RQ4. Does SBIR improve repair quality?** Yes. SBIR enables repairing more defects correctly for Arja and SequenceR (the more FL-sensitive tools). For example, Arja using SBIR correctly repairs 28 defects, but only 21 using SBFL, and only 15 using Blues. In fact, using SBIR, Arja repairs 7 defects it cannot repair with either SBFL or Blues, suggesting that the combination of bug reports and tests is even more useful, at times, than using both types of information separately. SimFix already has internal mechanisms for dealing with poor FL, and correctly repairs 30 defects using both SBIR and SBFL, but only 13 using Blues. We empirically show that SBIR significantly reduces repair failures due to localization errors. Finally, using SBIR, these tools correctly repair 7 defects that none of prior 14 APR tools repaired correctly, representing a 7.5% improvement in the number of defects ever repaired correctly by APR (Section III-E).

APR has already shown effectiveness in real-world scenarios, but producing correct repairs is one of the remaining hurdles preventing wide deployment in industry [27]. This paper makes progress addressing this challenge by (1) developing a new FL technique suitable for APR that uses both bug reports and tests, demonstrating that it localizes defects better than techniques that use only bug reports or only tests, and (2) demonstrating that with this new FL, APR tools can repair more defects correctly.

We make all of our data, source code, and documentation to reproduce our results publicly available [70]. The rest of this paper is organized as follows. Section II describes our FL techniques and Section III evaluates the FL techniques, and their effect on APR. Section IV places our work in the context of related research, and Section V summarizes our contributions.

## II. Combining FL for Program Repair

This section describes our Blues and SBFL techniques, our method for combining FL techniques called RAFL, and using RAFL to combine Blues and SBFL into SBIR.

### A. Blues: Localizing Bugs Using Bug Reports

We design Blues, an IRFL technique that uses bug reports to localize defects at the statement level. We create our own technique because existing techniques [115], [82], [99], [111], [98], [43] localize defects at the file or method level, while APR tools require statement-level localization. We do not use

iFixR's [44] IRFL (the only existing statement-level IRFL) because its pre-trained model uses projects [52] that overlap with the Defects4J and retraining on independent projects poses complex technical challenges and requires *another* large annotated dataset of real-world defects. Further, iFixR ignores `for` and `while` loops, which Blues handles. Blues builds on BLUiR [82], an unsupervised file-level IRFL technique that uses structured information retrieval to compute the similarity between bug reports and source code files. We select BLUiR because it is efficient, does not require training data, and performs comparably to other state-of-the-art file-level IRFL techniques [52]. Algorithm 1 describes our Blues approach.

**Ranking Suspicious Files.** For each defect, Blues' inputs are the bug report URL, the source files, the number of top ranked files to consider, the number of top ranked statements per file to consider, and a function to combine statement and file suspiciousness scores. Blues crawls the bug report from the input URL and parses the bug report to extract identifiers from the *summary* and *description* fields, and stores the information in a separate structured XML document (line 2 in Algorithm 1). Next, Blues processes the abstract syntax tree (AST) of source files to extract identifiers associated with comments and with class, method, and variable names, and stores them in separate XML documents (line 3). Blues preprocesses the terms stored in all the XML documents using CamelCase splitting, which improves matching recall, text normalization (removes punctuation, performs case-folding, tokenizes terms), stopword removal (removes extraneous terms), and stemming (conflates variants of the same underlying term) (lines 4–5). Blues then feeds the bug report and source file XML documents to BLUiR to compute ranked lists of suspicious files (line 6). BLUiR uses an IR model (TF-IDF formulation based on the BM25 (Okapi) model [80]) to search and rank the files based on their similarity with the bug report. Blues uses the same tuning parameters as BLUiR, which prior work [82] tuned using AspectsJ that does not overlap with Defects4J.

**Algorithm 1** Blues: Statement-level IR-based FL.

```
    Input: br: a bug report URL
    Input: srcFiles: collection of source files
    Input: irTool: file-level IRFL tool
    Input: f: number of suspicious files to consider
    Input: m: number of suspicious statements per file to consider
    Input: ScoreFn: function to combine file and statement scores
    Output: rankedStmtList: ranked list of suspicious statements
 1: function MAIN (br, srcFiles, irTool, f, m, ScoreFn)
 2:     br_xml ← ParseBugReportAndConvertToXML(br)
 3:     src_files_xml ← ParseSrcFilesAndConvertToXML(srcFiles)
 4:     PreProcess(br_xml)
 5:     PreProcess(src_files_xml)
 6:     ranked_files ← Okapi(br_xml, src_files_xml, irTool)
 7:     ranked_stmts ← LocalizeStatements(br_xml, ranked_files, irTool)
 8:     rankedStmtList ← Ranker(ranked_files, ranked_stmts, f, m, ScoreFn)
 9:     return rankedStmtList
10:
11: function LOCALIZESTATEMENTS (br_xml, ranked_files, irTool)
12:     src_stmts_xml ← [ ]                ▷ stores XMLs of parsed source statements
13:     for f ∈ ranked_files do
14:         S ← extractASTStatements(f)  ▷ extract 57 kinds of Java AST statements
15:         for ast_stmt ∈ S do
16:             stmt_xml ← ParseStmtAndConvertToXML(ast_stmt, f)
17:             src_stmts_xml.append(stmt_xml)
18:     PreProcess(src_stmts_xml)
19:     ranked_stmts ← Okapi(br_xml, src_stmts_xml, irTool)
20:     return ranked_stmts
```

**Ranking suspicious statements.** To rank suspicious statements from the top-ranked suspicious files, Blues parses the ASTs of the top-ranked suspicious files to extract 57 types of Java AST statements (lines 13–14). Prior work [59] shows that localizing bugs at the expression-level can improve repair tools. Therefore, unlike iFixR [44], which only extracts five kinds of AST statements (If, Return, Expression, FieldDeclaration, and VariableDeclaration), Blues extracts 32 AST expressions [79], 3 AST nodes (SingleVariableDeclaration, AnonymousClassDeclaration, Annotation), and 22 AST statements [19], 17 of which iFixR ignores, including `for` loops, `while` loops, `do` statements, etc. For readability, we refer to the AST expression, AST node, and AST statement as *statement*.

For each statement, Blues identifies its line number in the associated source file along with the file name, extracts identifier terms, and stores this information in an XML document (lines 15–16). Blues creates these XML documents for all the statements extracted from the ranked source files (line 17), and preprocesses these XMLs (line 18) in the same way it pre-processes source file XMLs. Next, Blues feeds these statement and the bug report XMLs to BLUiR that outputs a ranked list of the statements. Blues extracts the line number, source file name, and suspiciousness scores from the output to create a ranked list of suspicious statements (line 19). Note that these ranked statements do not consider the ranks of their associated source files. Real-world projects contain many source files, and our experiments show that treating all statements in a higher-ranked file to be more suspicious than the ones in lower-ranked files is sometimes suboptimal, so we also explore other strategies. To combine the ranked suspicious files and statements, Blues provides a ranker module that uses the three parameters: $f$, the number of suspicious files to consider; $m$, the number of suspicious statements per file to consider; and $ScoreFn$, a function for combining the file and statement suspiciousness scores (line 8). We define two such functions: $Score_{high}$ ranks the $m$ most suspicious statements in the most suspicious file, followed by $m$ statements in the next file, and so forth. $Score_{wt}$ uses the files' scores as weights for the associated suspicious statements and recomputes the weighted suspiciousness scores by multiplying the scores of the statements with the score of the associated file. We set $f = 50$ based on the recommendation of a prior study [44]. We run Blues' ranker module using six different configurations: five ($m \in \{1, 25, 50, 100, all\}$) with $Score_{high}$, and one ($m = all$) with $Score_{wt}$. For each of the six configurations, Blues produces a ranked list of statements.

We found that the six configurations localize complementary defects, so we use Algorithm 2 to combine the six ranked lists into a single list, which we call *Blues ensemble*. The algorithm to combine lists sorts the statements using each statement's highest rank in the six lists, breaking ties using the number of lists in which the statement occurs (line 15 in Algorithm 2). To fairly compare suspiciousness scores across lists, the algorithm normalizes the scores first (line 5). Note that computing the individual configurations and the ensemble is a relatively low-

**Algorithm 2** Combining ranked suspicious statement lists using suspiciousness scores and consensus.

```
   Input: rankedStmtLists ← [l_1, l_2, ...]
   Output: combinedStmtList
 1: function CombineLists(rankedStmtLists)
 2:     stmt_maxscore ← {}        ▷ max susp. score of stmt from all the lists
 3:     stmt_listcount ← {}       ▷ number of lists in which a stmt occurs
 4:     for l_k ∈ rankedStmtLists do
 5:         list_n ← NormalizeScoresInList(l_k)
 6:         for (stmt, score) ∈ list_n do
 7:             if score > 0.0 then
 8:                 if stmt ∉ stmt_listcount then
 9:                     stmt_listcount[stmt] = 1
10:                     stmt_maxscore[stmt] = score
11:                 else          ▷ stmt seen before, update maxscore if needed
12:                     stmt_listcount[stmt] += 1
13:                     if score > stmt_maxscore[stmt] then
14:                         stmt_maxscore[stmt] = score
15:     combinedStmtList ← SORT(stmt_maxscore, stmt_count)
16:     return combinedStmtList
```

cost process. One only needs to rerun Blues's ranker module (line 8 in Algorithm 1) and Algorithm 2, not the entire Blues pipeline. From here on, we use only the ensemble and refer to it as just Blues.

### B. Spectrum-Based Fault Localization

We do not create a new SBFL technique, but combine existing tools to produce a state-of-the-art implementation. SBFL compares program spectra — measurements of the runtime behavior of a program, such as code covered by tests [32] — of passing and failing developer-written tests to rank program elements, such as statements. SBFL calculates suspiciousness scores using a ranking strategy that considers four values collected from the spectrum: the number of failing tests that do ($e_f$) and do not ($n_f$) execute the element, and the number of passing tests that do ($e_p$) and do not ($n_p$) execute the element. While there are multiple ranking strategies, including Ochiai [1], DStar [100], and Tarantula [38], empirical studies [104], [117] have found that Ochiai is more effective for object-oriented programs. Most SBFL-based APR tools use Ochiai, and so does our study.

There exist multiple frameworks that APR tools use to compute code coverage, including JaCoCo [33], GZoltar [13], and Cobertura [15]. Our study uses GZoltar because most APR tools use it, and a recent study comparing 14 APR tools used multiple GZoltar versions, showing that the latest-at-the-time version (v1.6.0) significantly improved FL results and repair performance [60]. We use the latest version (v1.7.2) of GZoltar available at the time of running our experiments. GZoltar's inputs are the source code and test suite and its outputs are each statement's $e_f$, $n_f$, $e_p$, and $n_p$. We use the Ochiai ranking formula to compute suspiciousness scores: $score = \frac{e_f}{\sqrt{(e_f+n_f)(e_f+e_p)}}$.

To validate our SBFL implementation, we compare it to previously reported results [60] on Defects4J (v1.2.0) for SBFL implemented using Ochiai and older versions of Gzoltar. Figure 1 shows our SBFL implementation localizes 13 more defects than the best prior version.

| project | Chart | Closure | Lang | Math | Mockito | Time | Total |
|---|---|---|---|---|---|---|---|
| #defects | 26 | 133 | 65 | 106 | 38 | 27 | 395 |
| GZ v0.1.1 | 22 | 78 | 29 | 91 | 21 | 22 | 263 |
| GZ v1.6.0 | 24 | 95 | 57 | 100 | 23 | 22 | 321 |
| **GZ v1.7.2** | **25** | **101** | **53** | **96** | **36** | **23** | **334** |

Fig. 1. Our SBFL (implemented using GZoltar (v1.7.2) and Ochiai), in **bold**, localizes more defects than prior SBFLs using older versions of Gzoltar [60].

In the remainder of this paper, when we refer to our SBFL, we are referring to this particular implementation.

### C. Combining FL Techniques

Existing approaches to combining multiple FL techniques [53], [117], [87], [48], [104] typically use *learning to rank* [12] supervised machine learning. These techniques use multiple FL techniques' suspiciousness scores as *features* to train a model to rank buggy statements higher than non-buggy ones. Such approaches require a training dataset of program statements annotated with suspiciousness scores from multiple FL techniques, and the manually labeled ground truth "buggy" or "not-buggy". Such training data is hard to create because of the required manual effort, and the performance of trained models depends heavily on its data and features [65].

Instead, we propose RAFL, a novel unsupervised approach that requires no training. We formulate the problem of combining different FL techniques as a rank aggregation (RA) [57] problem. RA involves combining multiple ranked lists (base rankers) into one ranked list (aggregated ranker) [17]. The RA problem has been studied extensively in information retrieval [18], marketing and advertisement research [57], social choice (elections) [18], and genomics [42]. We propose to use RA algorithms to combine multiple FL techniques' ranked lists. We next describe our RAFL approach to combine FL techniques (Section II-C1) and using it to combine Blues and SBFL (Section II-C2). Section III-C2 will empirically show that our approach outperforms the supervised ones.

*1) RAFL: Rank Aggregation-based FL:* FL techniques typically assign suspiciousness scores to hundreds of program statements. Combining multiple ranked lists, which are often inconsistent, such that the result is as close as possible to the individual lists according to some distance metric, can become combinatorially intractable. We propose rank aggregation-based FL (RAFL), a novel approach that uses RA algorithms to combine FL. Our technique takes inspiration from the research in search-based software engineering [31], which involves applying metaheuristic search techniques to solve problems of balancing competing (and sometimes inconsistent) constraints. RAFL works as follows. Let $L_1, L_2, \ldots, L_m$ be $m$ ordered lists of suspicious statements (e.g., obtained using $m$ FL techniques). RAFL aims to create an ordered list $\delta$ of length $k \geq 1$ that combines the statements in the individual lists by minimizing the weighted sum of the distances between $\delta$ and the individual lists. Formally, RAFL minimizes the objective function $f(\delta) = \sum_{i=1}^{m} w_i d(\delta, L_i)$, where $w_i$ is the importance weight associated with list $L_i$, and $d$ is a distance metric.

| parameter | definition | SBIR value |
|---|---|---|
| k | size of the combined list | 100 |
| seed | seed specified for reproducibility | 1 |
| distance | Spearman or Kendall | Spearman |
| method | algorithm (CE or GA) | CE |
| maxIter | max #iterations allowed (default 1000) | 1000 |
| convIn | #consecutive iterations to decide if algorithm has converged (default: 7 for CE, 30 for GA) | 7 |
| importance | vector of weights ($w_i$) indicating the importance of each list (default: a vector of 1's (equal weights to all lists)) | default |
| N | #samples generated in each iteration. Used only by the CE (default: $10kn$, where $n$ is the #unique statements considering all ranked lists and $n >> k$, otherwise at least $k^2$) | 10,000 |
| $\rho$ | $(\rho \cdot N)$ is quantile of candidate lists sorted by the objective function scores. Used only by the CE. (default: 0.01 when $N \geq 100$ and 0.1 otherwise) | 0.01 |
| popSize | population size in each generation for the GA (default 100) | NA |
| CP | Cross-over probability for the GA (default 0.4) | NA |
| MP | Mutation probability for the GA | NA |

Fig. 2. RAFL configuration parameters.

To minimize the objective function, RAFL samples multiple lists of $k$ statements from the unique statements in the individual lists, using an algorithm-specific sampling strategy. RAFL computes the objective function for each sampled list. Iteratively, RAFL updates the sampled lists using the objective function computations, e.g., by adjusting the sampling probabilities or using genetic algorithms to select the next generation of sampled lists. This iteration continues until RAFL observes no change in the objective function scores for a fixed number of iterations, returning the lowest-scoring list.

Our RAFL implementation uses the RankAggreg [76] package, which implements several RA algorithms (cross-entropy Monte Carlo (CE), genetic algorithm (GA), and brute force) and provides distance metrics (Spearman Footrule [9], and Kendall's tau [8]). The left two columns in Figure 2 list RAFL configuration parameters, which can be used to select combinations of RA algorithms and distance metrics to combine FL.

*2) SBIR: Combining Blues and SBFL:* To combine the suspicious statement lists from Blues (Section II-A) and our SBFL (Section II-B), we use RAFL to develop SBIR using the cross-entropy Monte Carlo (CE) rank aggregation algorithm with the Spearman Footrule distance. We make these choices because prior work found CE to be typically more efficient than genetic algorithms [77] and than Borda count [16], [76], and because computing the Spearman Footrule distance is faster than Kendall's tau.

The CE algorithm represents an ordered list of $k$ statements using a 0–1 matrix of size $n \times k$, where $n$ is the total number of unique statements in the ranked lists and $k$ is the length of the desired combined list. The algorithm imposes two constraints: each column sums up to exactly 1, and each row sums up to at most 1. Under this representation, an ordered list of size $k$ is uniquely determined by reordering the matrix' rows (statements) such that the top $k$ rows form the identity matrix. For example, if the full list was [A, B, C], a $3 \times 2$ matrix, $\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}$ would translate into the candidate top 2 list of (C, B).

CE algorithm's goal is to identify a matrix that results in the minimum objective function score out of all possible matrices. The CE algorithm uses the following four steps: (1) **Initialization** creates an $n \times k$ matrix and assigns each cell a probability of $\frac{1}{n}$. This matrix represents the multinomial sampling probabilities of the statements: each statement (row) is equally likely to be in each of the $k$ positions (column). Next, CE runs steps 2 and 3 iteratively. (2) **Sampling** generates $N$ 0–1 matrices using the restricted (truncated) multinomial sampling [84] using the current probabilities. The output of this step are $N$ (new) randomly generated 0–1 matrices of size $n \times k$. (3) **Updating** computes the objective function scores for each of the $N$ sampled matrices, sorts the sampled matrices in the ascending order of the scores, and identifies $\rho$-quantiles $y^t$ of the sorted matrices. The algorithm uses the objective function scores of the matrices in iteration $t$ to update the multinomial cell probabilities of unique statements that tend to minimize the objective function scores of the matrices sampled in the next iteration, as follows:

$$p_{jr}^{t+1} = (1-w)p_{jr}^{t} + w\frac{\sum_{i=1}^{N} I(f(\delta_i) \leq y^t)x_{ijr}}{\sum_{i=1}^{N} I(f(\delta_i) \leq y^t)}$$

where $1 \leq j \leq n$, $1 \leq r \leq k$, $p_{jr}^{t}$ is the probability of the unique statement at the $jr^{\text{th}}$ position in the matrix at iteration $t$ and $p_{jr}^{t+1}$ is its updated value at iteration $t+1$; $f(\delta_i)$ is the objective function score of the $i^{\text{th}}$ sampled matrix and $x_{ijr}$ is the value of the $jr^{\text{th}}$ cell of the $i^{\text{th}}$ sampled matrix; $w$ is a weight parameter with a default value of 0.25 (tuned by prior work [77] on independent dataset) and $I$ is the indicator function. (4) **Convergence** stops the iteration when the minimum value of the objective function does not change in a preset number of iterations. The matrix with a minimum objective function score in the final iteration represents the final combined list of statements.

SBIR combines SBFL's and Blues' ranked suspicious statement lists to produce a single list of top-100 statements. The right column in Figure 2 shows the values of configuration parameters we used to develop SBIR. We select $k = 100$ because most APR tools consider at most 100 statements during repair. We set $w_i = 1.0$ to assign equal importance to SBFL and Blues and use default values of other parameters including $w$ (used in updating sampling probabilities), and $\rho$ that are tuned by prior work [77] on a dataset that does not overlap with Defects4J.

## III. Evaluation

We next evaluate our FL techniques and their effect on APR.

### A. FL Evaluation Dataset and Metrics

We use the Defects4J (v2.0) [26] benchmark to evaluate our FL techniques. Defects4J (v2.0) targets Java 8 and consists of 835 reproducible defects from 17 large open-source Java

| identifier | project | description | all | sfd | sld |
|---|---|---|---|---|---|
| Chart | jfreechart | framework to create charts | 8 | 8 | 4 |
| Cli | commons-cli | API for parsing command line options | 39 | 32 | 3 |
| Closure | closure-compiler | JavaScript compiler | 174 | 137 | 23 |
| Codec | commons-codec | implementations of encoders & decoders | 18 | 14 | 8 |
| Collections | commons-collections | Java Collections Framework extensions | 4 | 4 | 1 |
| Compress | commons-compress | API for file compression utilities | 47 | 43 | 4 |
| Csv | commons-csv | API to read and write CSV files | 16 | 15 | 5 |
| Gson | gson | API to convert Java Objects into JSON | 18 | 16 | 2 |
| JacksonCore | jackson-core | core part of the Java JSON API (Jackson) | 26 | 19 | 3 |
| JacksonDatabind | jackson-databind | data-binding package for Jackson | 111 | 91 | 13 |
| JacksonXml | jackson-dataformat-xml | data format extension for Jackson | 6 | 6 | 1 |
| Jsoup | jsoup | HTML parser | 93 | 75 | 18 |
| JxPath | commons-jxpath | XPath (an expression language) interpreter | 22 | 13 | 1 |
| Lang | commons-lang | extensions to Java Lang | 64 | 64 | 10 |
| Math | commons-math | library of math utilities | 106 | 98 | 23 |
| Mockito | mockito | a unit-test mocking framework | 38 | 33 | 7 |
| Time | joda-time | date and time library | 25 | 21 | 3 |
| total | | | 815 | 689 | 129 |

Fig. 3. The "all" column shows the 815 defects from the 17 real-world Java projects in the Defects4J (v2.0) benchmark we use to evaluate our FL techniques. The "sfd" column shows the 689 single-file-edit defects and the "sld" column shows the 129 single-line-edit defects we use for APR evaluations.

projects. Each defect comes with (1) one buggy and one developer-repaired version of the project code with the changes minimized to those relevant to the defect; (2) a set of developer-written tests, all of which pass on the developer-repaired version and at least one of which evidences the defect by failing on the buggy version; and (3) defect information, including the bug report URL. Out of the 835 defects, 817 have the bug report URL available, making IRFL possible. For 815 of the 817 defects, the test execution information was relevant to make SBFL possible. Figure 3 describes these 815 defects, which we use to evaluate our FL techniques.

We use two metrics, common to FL evaluations [117]: (1) $hit@k$ is the number of defects localized in the top-k ranked statements, and (2) EXAM is the fraction of ranked statements one has to inspect before finding a buggy statement. $hit@k$ tells us how useful an FL technique is for APR that uses the top $k$ statements, while EXAM tells us how highly the buggy statements are ranked, easing APR's job to produce correct patches.

Similar to prior studies [60], [117], [44], we consider a defect successfully localized when at least one of the buggy statements is in the top-k. Unlike studies that break ties by reassigning average rank [74] or expected rank [117], we rank same-suspiciousness statements in the order they appear in the FL results, as this is how APR tools process them.

### B. Blues' Evaluation (*RQ1*)

We next compare Blues' performance to the state-of-the-art (Section III-B1) and baseline (Section III-B2) IRFL techniques.

*1) Blues vs. State of the Art:* Figure 4 compares Blues with iFixR's internal statement-level IRFL technique [44] on the 171 Lang and Math defects in Defects4J on which iFixR was evaluated[1]. As shown in Figure 4, considering ranked lists of size $\geq 25$ (relevant for APR), Blues consistently localizes more defects (higher $hit@k$) than iFixR's IRFL. Comparing the ranks of buggy statements in localized defects, Blues places buggy statements higher (lowering EXAM) in the lists than iFixR. Blues' advantage of using a lightweight unsupervised approach outweighs iFixR's supervised technique that requires 6 file-level IRFL techniques.

| (171 defects) | $hit@k$ | | | | | EXAM |
|---|---|---|---|---|---|---|
| | $k=1$ | 25 | 50 | 100 | all | $k=$ all |
| iFixR | **26** | 74 | 95 | 106 | 135 | 0.048 |
| **Blues** | 11 | **79** | **97** | **108** | **151** | **0.034** |

Fig. 4. For ranked lists of size $\geq 25$, Blues localizes more defects ($hit@k$) and places buggy statements higher in the list (lower EXAM) than the state-of-the-art IRFL technique used in iFixR when evaluated on 171 Lang and Math defects in the Defects4J on which original iFixR was evaluated.

| (815 defects) | $hit@k$ | | | | | EXAM |
|---|---|---|---|---|---|---|
| | $k=1$ | 25 | 50 | 100 | all | $k=$ all |
| vanilla BLUiR | 26 | 143 | 192 | 245 | 611 | 0.159 |
| **Blues** | **27** | **184** | **241** | **306** | **611** | **0.111** |

Fig. 5. For all ranked list sizes, Blues consistently localizes more defects (higher $hit@k$) and ranks buggy statements higher (lower EXAM) than statement-level BLUiR that does not consider suspicious file scores when evaluated on the 815 defects available in the Defects4J v2.0.

*2) Blues vs. Baseline:* We implement a version of statement-level BLUiR (vanilla BLUiR) that does not consider the suspiciousness scores of the ranked suspicious files and instead ranks the suspicious statements only based on their similarity to the bug reports. Figure 5 compares Blues' and vanilla BLUiR performance on the 815 defects. For all list sizes, Blues consistently outperforms vanilla BLUiR with higher $hit@k$ and lower EXAM.

> For APR-relevant scenarios ($k \geq 25$), Blues consistently localizes more defects and ranks buggy statements higher than the state-of-the-art, *supervised*, statement-level IRFL technique used in iFixR. Blues also consistently outperforms a statement-level baseline that ignores suspicious files' ranks. **(RQ1)**

### C. SBIR's Evaluation

We next compare SBIR with its underlying SBFL and Blues (Section III-C1) and with state-of-the-art FL techniques (Section III-C2). As SBIR's ranked lists are at most 100 statements, our comparisons use that maximum. To account for the randomness in SBIR's Monte Carlo algorithm, we compute SBIR using 10 random seeds, reporting the mean, standard

[1]The iFixR FL results available at https://github.com/TruX-DTF/iFixR/tree/master/data/stmtLoc contain multiple statements with the same rank and multiple ranks for the same statement. We break ties by assigning the highest possible rank to each statement.

| (815 defects) | | *hit@k* | | | | EXAM | | |
|---|---|---|---|---|---|---|---|---|
| | | $k=1$ | 25 | 50 | 100 | $k=25$ | 50 | 100 |
| SBFL | | 88 | 408 | 475 | 549 | 0.287 | 0.240 | 0.220 |
| Blues | | 27 | 184 | 241 | 306 | 0.332 | 0.300 | 0.270 |
| **SBIR** | mean | **101** | **419** | **489** | **557** | **0.256** | **0.215** | **0.187** |
| (10 seeds) | stdev | 7.60 | 5.01 | 5.40 | 4.22 | 0.006 | 0.006 | 0.005 |
| | cv | 0.08 | 0.01 | 0.01 | 0.01 | 0.023 | 0.026 | 0.028 |

Fig. 6. Comparing SBIR, SBFL, and Blues FL performance on the 815 defects in Defects4J (v2.0). For all list sizes, SBIR consistently localizes more defects (higher $hit@k$) and places buggy statements higher in the list (lower EXAM) than underlying SBFL and Blues.

deviation (stdev), and coefficient of variation (cv $= \frac{\text{stdev}}{\text{mean}}$, which measures variability in relation to the mean of the population). A coefficient of variation less than 0.1 means the 10 seeds' results are tightly coupled [5].

*1) SBIR's FL Performance (**RQ2**):* Figure 6 shows the FL performance of SBIR, SBFL, and Blues for different list sizes. SBIR consistently localizes more defects (higher $hit@k$) and ranks buggy statements higher (lower EXAM) than SBFL and Blues. For example, considering top-100 statements, SBIR, on average, localizes 8 more defects than SBFL and 251 more defects than Blues. Comparing the ranks of buggy statements in the top-100 ranked lists, SBIR, on average, ranks buggy statements 19 (EXAM 0.187) while SBFL 22 (EXAM 0.220) and Blues 27 (EXAM 0.270). These results confirm prior findings suggesting that combining FL techniques can lead to better FL [53], [117], [34], [87], [48], [104]. Thus, an APR tool using SBIR gets earlier opportunities to patch the buggy statements and a more diverse set of localized defects than using SBFL or Blues.

> For all list sizes we consider, SBIR consistently localizes more defects and ranks buggy statements higher than underlying SBFL and Blues. **(RQ2)**

*2) SBIR vs. State of the Art (**RQ3**):* We compare SBIR to 9 standalone FL techniques and a supervised learning-to-rank approach [46] used by existing combining FL techniques.

**SBIR vs. Standalone FL.** Our evaluation considers techniques that were previously evaluated on Defects4J, make no assumptions about a priori knowing the buggy file, and localize buggy statements (as opposed to methods or files). We compare SBIR with 9 such standalone FL techniques used in a recent FL evaluation [117]: two SBFL — Ochiai and DStar; two mutation-based FL (MBFL) — Metallaxis and MUSE; three slicing — union, intersection, and frequency; one stack trace FL; and one predicate switching FL. The existing evaluation [118] provides a dataset of the 357 defects of Defects4J (v1.0) annotated with suspiciousness scores of the 9 techniques, but does not release the implementations of the individual techniques. We recreate ranked lists of the 9 techniques from the dataset. 334 of the 357 defects have bug reports available, making SBIR possible. We use these 334 defects for our analysis. Figure 7 compares the 9 techniques with SBIR. For all list sizes, SBIR consistently localizes more defects (higher $hit@k$) and ranks buggy statements higher (lower EXAM) than all of the 9 prior techniques.

| (334 defects) | | *hit@k* | | | | EXAM |
|---|---|---|---|---|---|---|
| family | technique | $k=1$ | 25 | 50 | 100 | $k=100$ |
| SBFL | Ochiai | 30 | 168 | 196 | 221 | 0.254 |
| | DStar | 32 | 169 | 199 | 222 | 0.254 |
| MBFL | Metallaxis | 40 | 154 | 175 | 195 | 0.238 |
| | MUSE | 26 | 96 | 104 | 118 | 0.193 |
| slicing | slicing-union | 21 | 87 | 100 | 111 | 0.462 |
| | slicing-intersection | 18 | 71 | 81 | 91 | 0.481 |
| | slicing-frequency | 21 | 86 | 100 | 112 | 0.458 |
| stack trace | stack trace | 16 | 28 | 28 | 28 | 0.663 |
| predicate switching | predicate switching | 9 | 24 | 24 | 24 | 0.662 |
| **SBIR** (10 seeds) | mean | **48** | **177** | **207** | **231** | **0.175** |
| | stdev | 4.31 | 4.16 | 2.92 | 2.32 | 0.006 |
| | cv | 0.09 | 0.02 | 0.01 | 0.01 | 0.034 |

Fig. 7. Comparing SBIR to 9 standalone FL techniques on 334 defects from Defects4J (v1.0). For all list sizes, SBIR consistently localizes more defects (higher $hit@k$) and places buggy statements higher in the ranked lists (lower EXAM) than each of the 9 techniques.

**SBIR vs. Supervised Combining FL Techniques.** Supervised learning-to-rank approaches (e.g., RankSVM [46], RankBoost [23], RankNet [12], FRank [92], LambdaRank [11]) can combine FL techniques. Most such state-of-the-art techniques (e.g., CombineFL [117], Fluccs [87], TraPT [54], Savant [48]) use RankSVM [46]. Thus, we compare our unsupervised RAFL with supervised RankSVM in combining SBFL and Blues.[2] We first create a dataset of the 815 defects by annotating program statements of each defect with normalized suspiciousness scores obtained using our SBFL and Blues, along with the ground truth information. We then use this annotated dataset to train the RankSVM model using SBFL's and Blues' scores as features. To evaluate the trained model, we use the CombineFL framework [118] that uses 10–fold cross validation and computes $E_{inspect}@k$ and EXAM metrics. The $E_{inspect}@k$ metric break ties by computing the expected rank of buggy statement in the ranked lists and then counts the number of defects whose buggy statements have expected rank $\leq k$. (As there are no ties in SBIR lists, $E_{inspect}@k$ is the same as the $hit@k$ for SBIR.) The EXAM scores are computed using the expected ranks of buggy statements in the lists therefore, we denote it as $\text{EXAM}_{inspect}$. Figure 8 compares SBIR (implemented using RAFL as described in Section II-C2) and SBIR (RankSVM) (the combination of SBFL and Blues, combined using RankSVM). For all lists of sizes, SBIR consistently localizes significantly more defects (higher $E_{inspect}@k$) and ranks buggy statements higher (lower $\text{EXAM}_{inspect}$) than RankSVM. The fact that SBIR is unsupervised and requires no training data is a further advantage over the supervised RankSVM approach.

> SBIR outperforms 9 standalone FL techniques and a supervised technique used by existing combiners. **(RQ3)**

[2]We could not compare RAFL to the deep learning-based DeepFL [53] because DeepFL's data is not public (https://github.com/DeepFL/DeepFaultLocalization/issues/4).

| (815 defects) | | $E_{inspect}@k$ | | | | EXAM$_{inspect}$ |
|---|---|---|---|---|---|---|
| technique | | $k=1$ | 25 | 50 | 100 | $k=100$ |
| SBIR (RankSVM) | | 50 | 270 | 328 | 396 | 0.236 |
| **SBIR (RAFL)** | mean | **101** | **419** | **489** | **556** | **0.187** |
| (10 seeds) | stdev | 7.60 | 5.01 | 5.41 | 4.22 | 0.005 |
| | cv | 0.08 | 0.01 | 0.01 | 0.01 | 0.027 |

Fig. 8. Comparing SBIR to a supervised-RankSVM combination of SBFL and Blues on 815 defects from Defects4J (v2.0). For all list sizes, SBIR consistently localizes more defects (higher $E_{inspect}@k$) and places buggy statements higher in the ranked lists (lower EXAM) than the RankSVM combination.

### D. APR Evaluation Tools, Dataset, and Metrics

We ran our experiments evaluating FL's effect on APR using a cluster of 50 compute nodes, each with a Xeon E5-2680 v4 CPU with 28 cores (2 processors, 14 cores each) running at 2.40GHz. Each node had 128GB of RAM and 200GB of local SSD. We launched multiple repair attempts in parallel, each requesting 4 cores on one compute node. We next describe the APR tools we evaluate, our dataset, and the metrics used.

*1) APR Tools Evaluated:* Instead of developing a new APR tool or arbitrarily selecting tools from state-of-the-art, we select Arja [113] and SimFix [36] that are the most ($Sen = 66.9\%$) and least ($Sen = 29.5\%$) FL-sensitive general purpose repair tools out of the 11 APR tools evaluated in a recent study [62] for their FL sensitivity. We select a third tool, SequenceR [14], which uses fundamentally different repair approach than Arja and SimFix, and whose FL-sensitivity ($Sen = 39.5\%$) lies between Arja and SimFix. Our tool selection criteria require that tools apply to general defects, rather than specialized, and have public implementations available so that they can be customized to take precomputed FL results. Arja, SequenceR, and SimFix use genetic-programming-[45], neural-machine-translation-[93], and fix-pattern-mining-based [61] repair approaches, respectively. Although there are more effective learning-based APR tools (e.g., CURE [37]) than SequenceR, which is only applicable to single-line-edit defects, we use SequenceR because its implementation is public and can be customized.

Using the dataset described next (Section III-D2), we use Arja and SimFix to repair 689 single-file-edit defects and SequenceR to repair 129 single-line-edit defects using SBFL, Blues, and SBIR for FL. We use the developer-written tests to validate the produced patches. As SBIR's ranked lists contain at most 100 suspicious statements, to fairly evaluate repair tools with respect to all three FL techniques, we limit repair tools to use top-100 suspicious program statements obtained by the three FL techniques. The original SequenceR evaluation [14] used (manually created) perfect FL and top-10 statements to repair a defect. We do the same for SequenceR. We do not otherwise modify the implementations of the three repair tools except customizing them to use our precomputed FL results.

*2) Dataset:* Manually assessing the correctness of patches that modify multiple files is error-prone and suffers from bias [49], [109]. To reduce errors and bias, we consider the 689 single-file-edit defects from the 815 defects from Section III-A. As SequenceR applies to single-line-edit defects, we use the 129 single-line-edit defects that are a subset of the 689 defects. Figure 3 shows the distribution of the 689 single-file-edit defects and 129 single-line-edit defects across the 17 projects in the Defects4J benchmark.

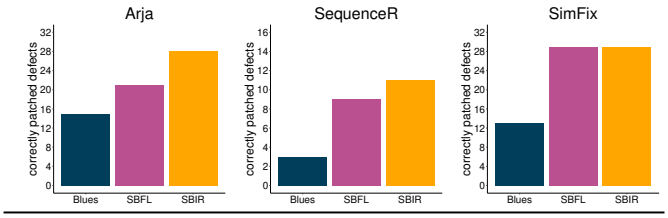
*3) Metrics:* Prior repair tools' evaluations that measure patch correctness use either manual inspection [67], [102], [49] or automatically-generated evaluation test suites [101], [103], [49], [71], [2]. While manual inspection is subjective and could be biased, using evaluation test-suites could inaccurately measure patch correctness [49]. Therefore, we propose a novel patch evaluation methodology that uses a hybrid of these methods to evaluate patch correctness.

For each patched defect, we use the developer-patched program (available for all Defects4J defects) as an oracle and use EvoSuite [22] to generate 10 held-out test suites using 10 seeds, a search budget of 12 minutes per seed, and a coverage criterion of maximizing line coverage of the developer-modified classes. We use EvoSuite because it is typically used to generate tests for regression oracles, and because prior studies [56], [71] preferred EvoSuite for this task. Most studies using EvoSuite use a 3 minute budget per seed, but using longer time budgets leads to better quality tests [71]. Therefore, we used 12 minutes (4 times what most prior studies use) per seed, for 10 seeds.

To check the correctness of an automatically produced patch, we first execute the held-out evaluation tests on the patch. If any test fails, we annotate such patch as *plausible* (the term used for a patch that passes developer-written tests but is incorrect [78]). This methodology is the state-of-the-art objective (but potentially incomplete [49]) automated test-driven patch correctness methodology [71]. If all the evaluation tests pass, we manually compare the patch against the developer's patch. If the patch is semantically equivalent to the developer's patch, we annotate it as *correct*. If it is not, we annotate it as *plausible*. If a patch is partially correct or we cannot determine its semantic equivalence because it requires extensive domain knowledge, which happens when the modifications are made to methods that are different from developer-modified ones, we conservatively annotate it as *plausible*. Thus, our patch evaluation methodology is conservative as it only considers a patch to be *correct* if it passes all held-out evaluation tests and is also semantically equivalent to the developer's patch. To study the effect of improving FL on APR, we compare the number of defects a repair tool correctly patches (*repair quality*), using different FL techniques. Since we had run SBIR with 10 seeds, we executed the APR tool experiments ten times, once for each SBIR result. We verified that the defects patched in each run is representative of all 10, but we only manually analyzed the patches for correctness for one run because of the significant manual effort involved.

### E. Effect of SBIR on APR quality (*RQ4*)

The top of Figure 9 compares repair quality of the three repair tools using the three FL techniques. Arja and SequenceR correctly patch more defects when using SBIR than when using




| | Arja (689 defects) | SequenceR (129 defects) | SimFix (689 defects) |
|---|---|---|---|
| repair quality assessment | | | |
| SBFL | 21 | 10 | 30 |
| Blues | 15 | 4 | 13 |
| SBFL ∪ Blues | 25 | 12 | 30 |
| **SBIR** | **28** | **12** | **30** |
| localization error assessment | | | |
| perfect FL | 21 | 24 | 21 |
| upper bound | 36 | 24 | 32 |
| ↓ # of defects not correctly patched due to localization error ↓ | | | |
| SBFL | 15 | 14 | 2 |
| Blues | 21 | 20 | 19 |
| **SBIR** | **8** | **12** | **2** |

Fig. 9. SBIR improves repair quality and reduces localization errors for more FL-sensitive APR tools. Arja and SequenceR, more FL-sensitive tools, correctly patch complementary defects using SBFL and Blues, and benefit more from using SBIR. SimFix, a less FL-sensitive repair tool, correctly patches the same number of defects using SBIR as SBFL but more than Blues.

SBFL or Blues. Specifically, Arja using SBIR correctly repairs 7 (33%) more defects than using SBFL and 13 (87%) more defects than using Blues. SequenceR using SBIR correctly patches 2 (20%) (out of a smaller subset of single-line defects) more defects than using SBFL and 8 (200%) more defects than using Blues. SimFix unsurprisingly correctly patches the same number of defects when using SBFL but 17 (131%) more defects than using Blues. More FL-sensitive repair tools, Arja and SequenceR, correctly patch complementary defects using SBFL and Blues, as evident by the row showing the union of defects they patch using SBFL and Blues. However, as the less FL-sensitive SimFix uses test case purification [105] and expands each suspicious statement by $\pm 5$ lines to address inaccurate FL, it does not patch complementary defects.

**Localization Error Analysis.** Multiple factors can prevent repair tools from producing correct patches. For example, if inaccurate FL ranks irrelevant non-buggy statements as more suspicious than buggy statements, the tool may produce plausible patches before having a chance to explore the buggy statement. This phenomenon is called APR localization error [44]. We next measure SBFL's, Blues', and SBIR's effect on localization error. We execute each of the three repair tools using perfect (manual) FL and measure the number of correctly patched defects ("perfect FL" row in Figure 9). We compute the "upper bound" [62] number of defects a repair tool can correctly patch as the union of defects correctly patched using the perfect FL and our three FL techniques. (Note that Arja and SimFix consider multiple suspicious statements and can

```
709   case Token.MOD:
710   if (rval == 0) {
711   error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right)
          ;     // Blues(38) SBIR(40)
712   return null;
713   }
714   result = lval % rval;
715   break;
716   case Token.DIV:
717   if (rval == 0) {
718   error(DiagnosticType.error("JSC_DIVIDE_BY_0_ERROR", "Divide by 0"), right)
          ;     // SBFL(1) Blues(36) SBIR(1)
719   return null;
720   }
```

Fig. 10. The two non-consecutive buggy statements (lines 711 and 718) that cause the Closure-78 defect. The annotations show which of the three FL techniques localize the buggy statements and their ranks in the respective lists.

patch more defects using SBIR than perfect FL. Their repair algorithms fail to construct patches for some defects when FL's ranked lists do not contain certain non-buggy statements adjacent to the buggy ones.) We then compute the localization error for each FL technique: the difference between the upper bound and the number of defects correctly patched using that FL technique. The bottom three rows of Figure 9 show that using SBIR significantly reduces the number of defects not patched due to localization error for the more FL-sensitive repair tools, compared to SBFL and Blues.

Overall, Arja and SequenceR significantly benefit from SBIR. Arja using SBIR correctly patches 28 (78%) of the 36 upper bound defects, whereas using SBFL, it only patches 21 (58%) and using Blues only 15 (42%). SequenceR using SBIR correctly patches 12 (50%) of the 24 upper bound defects, whereas using SBFL, it patches 10 (42%) and using Blues 4 (17%). SimFix, correctly patches 30 (94%) of the 32 upper bound defects using both SBIR and SBFL, and patches 13 (41%) using Blues.

**Case Study Illustrating How SBIR Helps APR.** The three APR tools using SBIR correctly patched 7 defects (Chart-12, Closure-68, Closure-78, Closure-86, Closure-92, Lang-10, and Lang-20) in Defects4J that none of the existing 14 APR tools patch. That is a 7.5% improvement over the 93 defects in Defects4J (v1.0) that at least one of the 14 tools correctly patches [60]. Closure-78 and Lang-20 require editing multiple code locations in a single file, and most repair tools struggle to patch these kind of defects. For example, Arja using SBIR correctly patches Closure-78, whose repair involves deleting two non-consecutive statements (lines 711 and 718 in Figure 10) to fix a division-by-0 error. None of the existing 14 APR tools [60], nor Arja with our SBFL or Blues, patch this defect. For this defect, SBFL ranks only line 718 in the 1st position; Blues ranks line 718 36th and line 711 38th; and SBIR ranks line 718 1st and line 711 40th. Arja using SBFL produces a plausible, but only partially correct patch that deletes line 718 while Arja using Blues does not produce a patch, timing out trying to modify the 35 non-buggy statements ranked higher. Arja using SBIR produces a correct patch (identical to the developer patch) because it finds the buggy statement at the top of SBIR's list, and then fetches the second buggy statement because it is also in SBIR's list and because it uses the same variables and methods as the top-ranked line (Arja's *ingredient screening*

step [113]). Arja constructs a correct patch by deleting both the buggy statements. Thus, it is precisely the *combination* of the information from bug reports and test executions that enables Arja to localize and correctly repair this defect.

**SBIR vs. Union of SBFL and Blues.** Since APR using SBFL and IRFL often repairs complementary defects [44], we set out to measure how defects repaired with SBIR compare to the union of the defects repaired with Blues and SBFL. We find that while there is some complementarity, for Arja, SBIR repairs more defects than the union, suggesting that combining bug reports and tests not only captures most (though not all) of the benefits of the two, it is also able to extract a combined benefit where neither Blues nor SBFL alone leads to a repair. Arja patches 25 defects (row 3 in Figure 9) using SBFL and Blues, including 4 defects (Compress 27, Jsoup 33, Jsoup 55, and Time 15) that Arja could not patch using SBIR. However, Arja using SBIR patches 28 defects, including 7 defects (Closure 78, Gson 7, Jsoup 39, Jsoup 68, Jsoup 85, JxPath 5, and Lang 7) that Arja could not patch using SBFL or Blues. Thus, for Arja, SBIR is even more beneficial than using both SBFL and Blues. SequenceR patches 12 defects using both SBFL and Blues that include one defect (Cli 40) that SequenceR could not patch using SBIR. However, SequenceR using SBIR also patches 12 defects that include one defect (JacksonCore 25) that SequenceR could not patch using SBFL or Blues. Thus, SBIR provides the same benefit to SequenceR as using both SBFL and Blues. SimFix correctly patches the same 30 defects using SBIR as it does using both SBFL and Blues. Thus, for SimFix, using SBIR provides the same benefit as using just SBFL or both SBFL and Blues.

**SBIR vs. Original Published APR Versions.** We find that the three repair tools using SBIR correctly repair somewhat complementary defects to those the original published versions repaired. Arja using SBIR correctly patches 4 defects (Lang-7, Lang-10, Lang-59, and Math-35) original Arja did not. Of the defects in our dataset, the original Arja correctly patched 15 defects [113] (plus 3 others that either had no bug reports (Chart-3) or were multi-file-edit defects (Math-22 and Math-98)). Of these 15, Arja using SBIR correctly patches 12, but not the other 3 (Lang-35, Math-39, and Math-86). We examined the original evaluation's patches[3] and found that for these 3 defects, Arja had produced only a single patch, which is highly uncommon for Arja (it produced many patches for all other defects it patched), suggesting that there is something special about these defects or the process the Arja evaluation followed in repairing them. Overall, Arja with SBIR correctly patches 1 more defect than the original Arja. SimFix using SBIR correctly patches 3 defects (Closure-68, Closure 92, and Closure-126) original SimFix did not. Of the defects in our dataset, original SimFix correctly patched 21 defects [36] (plus 6 others that either had no bug reports (Chart-3, Chart-7, Chart-20) or were multi-file-edit defects (Closure-63, Math-71, and Math-98)). Of these 21, SimFix with SBIR correctly patches 19. (Note that the original evaluation [36] listed 7 more defects (Closure-115, Lang-16, Lang-27, Lang-39, Lang-41, Lang-50, and Lang-60) as patched correctly. The authors subsequently identified one of those (Lang-27) as incorrect,[4] and our analysis revealed that the six others are also incorrect. SimFix with SBIR could not patch the remaining two defects (Math-35 and Math-63). Overall, SimFix with SBIR correctly patches 1 more defect than the original SimFix. SequenceR's original evaluation used perfect FL [14], so a direct comparison is not appropriate. With perfect FL, original SequenceR patched 14 defects correctly, and with SBIR, it patches 6 (Chart-11, Closure-73, Closure-86, Lang-59, Math-58, and Math-75) of those 14.

> SBIR significantly improves repair quality and reduces localization errors for more FL-sensitive APR tools, and enables correctly repairing some defects that they cannot repair with other FL techniques. For less FL-sensitive APR, SBIR provides the same repair quality as SBFL. Using SBIR, we are able to correctly repair 7 defects never previously automatically repaired correctly by existing techniques. **(RQ4)**

### *F. Discussion and Threats to Validity*

Our approach requires a bug report and a bug-exposing test. This requirement is not always met: several defects in Defects4J (v2.0) have no documented bug reports, and prior studies [44], [39] show that for 92% of defects, bug-exposing tests are added after the bug is reported. However, most repair tools cannot function without either a failing test or a bug report, and existing repair tools that use only bug reports are not fully automatic (a human must validate the proposed patches). Meanwhile, while test-driven APR can be fully automated, most patches it produces are incorrect [72], [71], [86]. Our work extends APR to use SBIR, which uses both bug reports and test suites, enabling repair tools to be fully automated and to produce higher-quality patches. This is a worthwhile achievement even if not all defects in industrial settings have the requisite artifacts, and may motivate developers to create the artifacts in the future, which reinforces an already recommended practice. Further, combining the available user inputs to improve APR can foster trust in the generated patches [73], thereby helping the adoption of repair techniques.

Blues' effectiveness depends on the quality of bug reports. For example, Blues could not localize the Chart 2 defect[5] because its bug report only contains a URL and no description. This caused SBIR to lower the rank of the buggy statement in its ranked list of suspicious statements.

It is not a goal of our study to develop the best way to combine FL techniques. Instead, because existing combining techniques are trained on Defects4J, we could not use them in our evaluation, so we created RAFL, an unsupervised combining method. We show that RAFL outperforms RankSVM, a state-of-the-art supervised combining method, and that it is

[3]https://github.com/yyxhdy/defects4j-patches/tree/master/Arja

[4]https://github.com/xgdsmileboy/SimFix/tree/master/final/result

[5]https://sourceforge.net/p/jfreechart/bugs/959/

sufficient to demonstrate improvement in APR performance. However, comparing RAFL with *all* supervised methods is out of scope of this study.

Our evaluation aims to measure the impact of combined IRFL and SBFL on APR in a way that will generalize to a wide range of APR techniques. That is why our evaluation uses three diverse APR techniques. The design of our study allows estimating the SBIR's impact on a repair technique based on its FL-sensitivity. For example, a recent technique Recoder [116] has an FL-sensitivity of 34.5%, similar to SequenceR's 39.5%. Thus, we expect SBIR's impact on Recoder will be similar to that on SequenceR, but smaller than that on Arja (sensitivity 66.9%).

Arja and SequenceR are stochastic and results may vary across executions. We address this threat by using a large-scale dataset. Executing our study is highly computationally intensive and required eight weeks of wall-clock time on a 50-node cluster. To enable others to independently reproduce our results, and reuse our FL techniques in improving APR, we make all code and data available.

We address threats to internal validity by reusing publicly available implementations of repair tools instead of reimplementing them. We address threats to external validity by selecting diverse APR tools and using Defects4J (v2.0) that has significantly more projects and defects than earlier versions.

## IV. Related Work

**Improving APR Performance.** Program repair tools typically follow a three step process: identifying the location of a defect, producing candidate patches, and validating those patches. The method used for each of these steps can significantly affect the tool's success. To improve APR, researchers have proposed to use different kinds of FL strategies [6], [107], [88], [44], [35], [64], patch generation algorithms (e.g., heuristic-based [50], [63], [91], [97], [36], [75], constraint-based [2], [94], [29], [68], [40], and learning-based [14], [30], [83]), and patch validation methodologies [95], [108], [112], [90], [109]. Assuming perfect FL, recent study [62] shows that modern repair tools can patch significantly more defects. However, assuming perfect FL is unrealistic and therefore we propose to improve automated FL used in APR.

Recent APR research has used formal constraints derived by program analyzers instead of test suites [28], [25]. These techniques patch specific families of defects, such as security vulnerabilities and exception-causing defects, and our approach to improving general-purpose APR is complementary.

APR's fundamental challenge is generating fewer incorrect patches [86], [71], [78]. In some domains, e.g., formal verification, an oracle exists to determine patch correctness [21], [20], [85], [3], overcoming this problem, though better FL can still lead to the production of more patches.

**Improving FL.** Techniques to improve FL can be classified into two categories. The first category is the standalone techniques. For example, PRoFL [64] improves SBFL using patch execution results from APR, PREDFL [34] uses runtime statistics from statistical debugging to improve SBFL, PRFL [114] uses the PageRank algorithm, XGB-FL [106] uses a classifier to learn the importance of program statements and features, such as execution sequence and semantics, UniVal [47] uses execution profiles and the success and failure information from program executions, in conjunction with statistical inference, and DeepRL4FL [55] formulates FL as pattern recognition and uses code coverage representation learning to improve SBFL and MBFL techniques. The second category (e.g., CombineFL [117], DeepFL [53], Fluccs [87], Savant [48], Multric [104], and TraPT [54]) uses learning-to-rank [12] machine learning approaches such as RankSVM [46] to combine multiple FL techniques. RAFL outperforms RankSVM, the state-of-the-art supervised method.

Property-based testing, e.g., for software fairness [24], [89], [4], [10], and automated oracle generation [69] can synthesize additional tests to improve FL in ways complementary to our approach.

**FL in Program Repair.** Most repair tools use SBFL implemented using off-the-shelf coverage tracking tools and the Ochiai ranking strategy [50], [63], [91], [30], [83], [97], [36], [94], [68], [29], [2], [14], [110]. R2Fix [58] and iFixR [44] are the only two IRFL-based repair tools, and no prior repair tool uses combined SBFL and IRFL. Although, using patch-execution results from repair tools to refine FL results can outperform state-of-the-art SBFL and MBFL techniques [64]. Recent studies have shown the effect of using different technologies, assumptions, and adaptations of test-suite-based FL techniques on the performance of repair tools [2], [60], [35], [88], [107], [96], [6]. Often, APR researchers omit FL tuning used by their repair tools while presenting repair performance, which leads to bias in comparing performance of different repair tools [62], [60]. Further, the tuned FL implementations are often tightly coupled to the repair tool implementations, which makes it hard to reuse them for other repair tools. Our FL techniques can be used to mitigate this bias as they can serve as a plugin by future repair tools to decouple their FL implementations from their repair algorithm implementation, as is done in some frameworks, including JaRFly [71].

## V. Contributions

We have developed SBIR, an FL technique that uses both bug reports and tests to localize defects, and showed that it helps improve APR quality for FL-sensitive tools, repairing more defects correctly than by using other FL techniques. Along the way, we also created Blues, the first statement-level, information-retrieval-based FL that outperforms the state of the art without needing ground truth data for training, and RAFL, a novel unsupervised method for combining arbitrary FL techniques. Our results demonstrate that combining bug reports and tests leads to better FL, and enables higher-quality APR. Our findings support further research into improving APR by combining bug-report-based and test-based information.

## Data Availability

All of our data, source code, and documentation to reproduce our results are publicly available [70].

## ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under grants no. CCF-1763423 and CCF-2210243.

## REFERENCES


[1] R. Abreu, P. Zoeteweij, and A. J. V. Gemund. On the accuracy of spectrum-based fault localization. In *TAIC-PART*, pages 89–98, 2007.

[2] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues. SOSRepair: Expressive semantic search for real-world program repair. *TSE*, 47(10):2162–2181, October 2021.

[3] A. Agrawal, E. First, Z. Kaufman, T. Reichel, S. Zhang, T. Zhou, A. Sanchez-Stern, T. Ringer, and Y. Brun. Proofster: Automated formal verification. In *ICSE Demo*, May 2023.

[4] R. Angell, B. Johnson, Y. Brun, and A. Meliou. Themis: Automatically testing software for discrimination. In *ESEC/FSE Demo*, pages 871–875, November 2018.

[5] S. Aronhime, C. Calcagno, G. H. Jajamovich, H. A. Dyvorne, P. Robson, D. Dieterich, M. I. Fiel, V. Martel-Laferriere, M. Chatterji, H. Rusinek, and B. Taouli. DCE-MRI of the liver: Effect of linear and nonlinear conversions on hepatic perfusion quantification and reproducibility. *Journal of Magnetic Resonance Imaging*, 40(1):90–98, 2014.

[6] F. Y. Assiri and J. M. Bieman. Fault localization for automated program repair: Effectiveness, performance, repair correctness. *Software Quality Journal*, 25(1):171–199, 2017.

[7] J. Bader, A. Scott, M. Pradel, and S. Chandra. Getafix: Learning to fix bugs automatically. *PACMPL OOPSLA*, 3, October 2019.

[8] F. J. Brandenburg, A. Gleißner, and A. Hofmeier. Comparing and aggregating partial orders with kendall tau distances. *Discrete Mathematics, Algorithms and Applications*, 5(02):1360003, 2013.

[9] F. J. Brandenburg, A. Gleißner, and A. Hofmeier. The nearest neighbor spearman footrule distance for bucket, interval, and partial orders. *Journal of Combinatorial Optimization*, 26(2):310–332, 2013.

[10] Y. Brun and A. Meliou. Software fairness. In *ESEC/FSE NIER*, 2018.

[11] C. Burges, R. Ragno, and Q. Le. Learning to rank with nonsmooth cost functions. In *NeurIPS*, pages 193–200, 2006.

[12] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender. Learning to rank using gradient descent. In *ICML*, pages 89–96, 2005.

[13] J. Campos, A. Riboira, A. Perez, and R. Abreu. Gzoltar: An Eclipse plug-in for testing and debugging. In *ASE*, pages 378–381, 2012.

[14] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *TSE*, 47(9):1943–1959, 2019.

[15] S. Christou. Cobertura code coverage tool. https://cobertura.github.io/cobertura/, 2015.

[16] V. Debroy and W. E. Wong. A consensus-based strategy to improve the quality of fault localization. *Software: Practice and Experience*, 43(8):989–1011, 2013.

[17] K. Deng, S. Han, K. J. Li, and J. S. Liu. Bayesian aggregation of order-based rank data. *JASA*, 109(507):1023–1039, 2014.

[18] C. Dwork, R. Kumar, M. Naor, and D. Sivakumar. Rank aggregation methods for the web. In *WWW*, pages 613–622, 2001.

[19] Eclipse JDT API specification. https://ibm.co/3orMarh. [accessed 4-March-2022].

[20] E. First and Y. Brun. Diversity-driven automated formal verification. In *ICSE*, pages 749–761, May 2022.

[21] E. First, Y. Brun, and A. Guha. TacTok: Semantics-aware proof synthesis. *PACMPL OOPSLA*, 4:231:1–231:31, November 2020.

[22] G. Fraser and A. Arcuri. Whole test suite generation. *TSE*, 39(2):276–291, February 2013.

[23] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer. An efficient boosting algorithm for combining preferences. *JMLR*, 4:933–969, Nov. 2003.

[24] S. Galhotra, Y. Brun, and A. Meliou. Fairness testing: Testing software for discrimination. In *ESEC/FSE*, pages 498–510, September 2017.

[25] X. Gao, B. Wang, G. J. Duck, R. Ji, Y. Xiong, and A. Roychoudhury. Beyond tests: Program vulnerability repair via crash constraint extraction. *TOSEM*, 30(2):14:1–14:27, Feb. 2021.

[26] G. Gay and R. Just. Defects4J as a challenge case for the search-based software engineering community. In *SSBSE*, pages 255–261, 2020.

[27] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *TSE*, 45(01):34–67, 2019.

[28] D. Ginelli, O. Riganelli, D. Micucci, and L. Mariani. Exception-driven fault localization for automated program repair. In *QRS*, 2021.

[29] S. Gulwani, I. Radiček, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. In *PLDI*, 2018.

[30] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. DeepFix: Fixing common C language errors by deep learning. In *AAAI*, 2017.

[31] M. Harman and B. F. Jones. Search-based software engineering. *Information and Software Technology*, 43(14):833–839, 2001.

[32] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *STVR*, 10(3):171–194, 2000.

[33] M. R. Hoffmann, B. Janiczak, E. Mandrikov, and M. Friedenhagen. JaCoCo code coverage tool. https://www.jacoco.org/jacoco/, 2009.

[34] J. Jiang, R. Wang, Y. Xiong, X. Chen, and L. Zhang. Combining spectrum-based fault localization and statistical debugging: An empirical study. In *ASE*, pages 502–514, 2019.

[35] J. Jiang, Y. Xiong, and X. Xia. A manual inspection of Defects4J bugs and its implications for automatic program repair. *Science China Information Sciences*, 62(10):200102, 2019.

[36] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *ISSTA*, 2018.

[37] N. Jiang, T. Lutellier, and L. Tan. CURE: Code-aware neural machine translation for automatic program repair. In *ICSE*, 2021.

[38] J. A. Jones and M. J. Harrold. Empirical evaluation of the tarantula automatic fault-localization technique. In *ASE*, pages 273–282, 2005.

[39] R. Just, C. Parnin, I. Drosos, and M. D. Ernst. Comparing developer-provided to user-provided tests for fault localization and automated program repair. In *ISSTA*, pages 287–297, July 2018.

[40] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun. Repairing programs with semantic code search. In *ASE*, pages 295–306, November 2015.

[41] S. Kirbas, E. Windels, O. McBello, K. Kells, M. Pagano, R. Szalanski, V. Nowack, E. R. Winter, S. Counsell, D. Bowes, T. Hall, S. Haraldsson, and J. Woodward. On the introduction of automatic program repair in bloomberg. *Software*, 38(4):43–51, 2021.

[42] R. Kolde, S. Laur, P. Adler, and J. Vilo. Robust rank aggregation for gene list integration and meta-analysis. *Bioinformatics*, 28(4):573–580, 2012.

[43] A. Koyuncu, T. F. Bissyandé, D. Kim, K. Liu, J. Klein, M. Monperrus, and Y. L. Traon. D&C: A divide-and-conquer approach to IR-based bug localization. *CoRR*, abs/1902.02703, 2019.

[44] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. L. Traon. iFixR: Bug report driven program repair. In *ESEC/FSE*, pages 314–325, 2019.

[45] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[46] T.-M. Kuo, C.-P. Lee, and C.-J. Lin. Large-scale kernel ranksvm. In *SIAM Intl. Conf. on data mining*, pages 812–820. SIAM, 2014.

[47] Y. Küçük, T. A. D. Henderson, and A. Podgurski. Improving fault localization by integrating value and predicate based causal inference techniques. In *ICSE*, pages 649–660, 2021.

[48] T. B. Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Intl. Symposium on Software Testing and Analysis*, pages 177–188, 2016.

[49] X. D. Le, L. Bao, D. Lo, X. Xia, S. Li, and C. S. Pasareanu. On reliability of patch correctness assessment. In *ICSE*, 2019.

[50] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. GenProg: A generic method for automatic software repair. *TSE*, 38:54–72, 2012.

[51] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *CACM*, 62(12):56–65, Nov. 2019.

[52] J. Lee, D. Kim, T. F. Bissyandé, W. Jung, and Y. L. Traon. Bench4BL: Reproducibility study on the performance of IR-based bug localization. In *ISSTA*, pages 61–72, 2018.

[53] X. Li, W. Li, Y. Zhang, and L. Zhang. DeepFL: Integrating multiple fault diagnosis dimensions for deep fault localization. In *ISSTA*, 2019.

[54] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *PACML OOPSLA*, 1:1–30, 2017.

[55] Y. Li, S. Wang, and T. Nguyen. Fault localization with code coverage representation learning. In *ICSE*, pages 661–673, 2021.

[56] R. Lima, J. F. Ferreira, and A. Mendes. Automatic repair of java code with timing side-channel vulnerabilities. In *International Workshop on Refactoring (IWOR)*, pages 1–8, 2021.

[57] S. Lin. Rank aggregation methods. *Wiley Interdisciplinary Reviews: Computational Statistics*, 2(5):555–570, 2010.

[58] C. Liu, J. Yang, L. Tan, and M. Hafiz. R2Fix: Automatically generating bug fixes from bug reports. In *ICST*, pages 282–291, 2013.

[59] K. Liu, D. Kim, A. Koyuncu, L. Li, T. F. Bissyandé, and Y. L. Traon. A closer look at real-world patches. In *ICSME*, pages 275–286, 2018.

[60] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. L. Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *ICST*, pages 102–113, 2019.

[61] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *ISSTA*, pages 31–42, 2019.

[62] K. Liu, L. Li, A. Koyuncu, D. Kim, Z. Liu, J. Klein, and T. F. Bissyandé. A critical review on the evaluation of automated program repair systems. *Journal of Systems and Software*, 171:110817, 2021.

[63] F. Long and M. Rinard. Automatic patch generation by learning correct code. In *POPL*, pages 298–312, 2016.

[64] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang. Can automated program repair refine fault localization? a unified debugging approach. In *ISSTA*, pages 75–87, 2020.

[65] S. Mahalakshmi and E. Sivasankar. Cross domain sentiment analysis using different machine learning techniques. In *Intl. Conf. on Fuzzy and Neuro Computing (FANCCO)*, pages 77–87, 2015.

[66] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott. SapFix: Automated end-to-end repair at scale. In *ICSE*, pages 269–278, 2019.

[67] M. Martinez, T. Durieux, R. Sommerard, J. Xuan, and M. Monperrus. Automatic repair of real bugs in Java: A large-scale experiment on the Defects4J dataset. *EMSE*, 22(4):1936–1964, April 2017.

[68] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *ICSE*, pages 129–139, 2018.

[69] M. Motwani and Y. Brun. Automatically generating precise oracles from structured natural language specifications. In *ICSE*, May 2019.

[70] M. Motwani and Y. Brun. Replication data for: Better automatic program repair by using bug reports and tests together. Harvard Dataverse, https://doi.org/10.7910/DVN/OHMYAK, 2023.

[71] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues. Quality of automated program repair on real-world defects. *TSE*, 48(2):637–661, February 2022.

[72] K. Noda, Y. Nemoto, K. Hotta, H. Tanida, and S. Kikuchi. Experience report: How effective is automated program repair for industrial software? In *SANER*, pages 612–616, 2020.

[73] Y. Noller, R. Shariffdeen, X. Gao, and A. Roychoudhury. Trust enhancement issues in program repair. In *ICSE*, 2022.

[74] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *ICSE*, pages 609–620, 2017.

[75] J. Petke and A. Blot. Refining fitness functions in test-based program repair. In *APR*, page 13–14, 2018.

[76] V. Pihur, S. Datta, and S. Datta. RankAggreg, an R package for weighted rank aggregation. *BMC bioinformatics*, 10(1):62, 2009.

[77] V. Pihur, S. Datta, and S. Datta. RankAggreg: Weighted rank aggregation. https://cran.r-project.org/web/packages/RankAggreg/, 2020.

[78] Z. Qi, F. Long, S. Achour, and M. Rinard. An analysis of patch plausibility and correctness for generate-and-validate patch generation systems. In *ISSTA*, page 24–36, 2015.

[79] Rational software architect 9.5.0. https://ibm.co/3HfEml7. [accessed 4-March-2022].

[80] S. E. Robertson, S. Walker, and M. Beaulieu. Experimentation as a way of life: Okapi at trec. *Information processing & management*, 36(1):95–108, 2000.

[81] R. Y. Rubinstein and D. P. Kroese. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation and Machine Learning*. Springer Science & Business Media, 2013.

[82] R. K. Saha, M. Lease, S. Khurshid, and D. E. Perry. Improving bug localization using structured information retrieval. In *ASE*, 2013.

[83] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. ELIXIR: Effective object oriented program repair. In *ASE*, pages 648–659, 2017.

[84] L. Sanathanan. Estimating the size of a truncated sample. *Journal of the American Statistical Association*, 72(359):669–672, 1977.

[85] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer. Passport: Improving automated formal verification using identifiers. *ACM TOPLAS*, 2023.

[86] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *ESEC/FSE*, pages 532–543, 2015.

[87] J. Sohn and S. Yoo. FLUCCS: Using code and change metrics to improve fault localization. In *ISSTA*, pages 273–283, 2017.

[88] S. Sun, J. Guo, R. Zhao, and Z. Li. Search-based efficient automated program repair using mutation and fault localization. In *COMPSAC*, volume 1, pages 174–183, 2018.

[89] P. S. Thomas, B. C. da Silva, A. G. Barto, S. Giguere, Y. Brun, and E. Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, November 2019.

[90] H. Tian, K. Liu, A. K. Kaboré, A. Koyuncu, L. Li, J. Klein, and T. F. Bissyandé. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *ASE*, 2020.

[91] Y. Tian and B. Ray. Automatically diagnosing and repairing error handling bugs in C. In *ESEC/FSE*, pages 752–762, 2017.

[92] M.-F. Tsai, T.-Y. Liu, T. Qin, H.-H. Chen, and W.-Y. Ma. Frank: a ranking method with fidelity loss. In *SIGIR*, pages 383–390, 2007.

[93] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk. On learning meaningful code changes via neural machine translation. In *ICSE*, page 25–36, 2019.

[94] K. Wang, R. Singh, and Z. Su. Search, align, and repair: Data-driven feedback generation for introductory programming exercises. In *PLDI*, 2018.

[95] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin. Automated patch correctness assessment: How far are we? In *ASE*, page 968–980. Association for Computing Machinery, 2020.

[96] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. An empirical analysis of the influence of fault space on search-based automated program repair. *CoRR*, abs/1707.05172, 2017.

[97] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung. Context-aware patch generation for better automated program repair. In *ICSE*, 2018.

[98] M. Wen, R. Wu, and S.-C. Cheung. Locus: Locating bugs from software changes. In *ASE*, pages 262–273, 2016.

[99] C.-P. Wong, Y. Xiong, H. Zhang, D. Hao, L. Zhang, and H. Mei. Boosting bug-report-oriented fault localization with segmentation and stack-trace analysis. In *ICSME*, pages 181–190, 2014.

[100] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The DStar method for effective software fault localization. *TR*, 63(1):290–308, 2013.

[101] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *ISSTA*, pages 226–236, 2017.

[102] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *ICSE*, 2018.

[103] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *ICSE*, 2017.

[104] J. Xuan and M. Monperrus. Learning to combine multiple ranking metrics for fault localization. In *ICSME*, pages 191–200, 2014.

[105] J. Xuan and M. Monperrus. Test case purification for improving fault localization. In *FSE*, pages 52–63, 2014.

[106] B. Yang, Y. He, H. Liu, Y. Chen, and Z. Jin. A lightweight fault localization approach based on xgboost. In *QRS*, pages 168–179, 2020.

[107] D. Yang, Y. Qi, and X. Mao. Evaluating the strategies of statement selection in automated program repair. In *SATE*. Springer, 2018.

[108] J. Yang, A. Zhikhartsev, Y. Liu, and L. Tan. Better test cases for better automated program repair. In *ESEC/FSE*, pages 831–841, 2017.

[109] H. Ye, M. Martinez, and M. Monperrus. Automated patch assessment for program repair at scale. *EMSE*, 26(2), 2021.

[110] H. Ye, M. Martinez, and M. Monperrus. Neural program repair with execution-based backpropagation. In *ICSE*, pages 1506—1518, 2022.

[111] K. C. Youm, J. Ahn, J. Kim, and E. Lee. Bug localization based on code change histories and bug reports. In *APSEC*, 2015.

[112] Z. Yu, M. Martinez, B. Danglot, T. Durieux, and M. Monperrus. Alleviating patch overfitting with automatic test generation: A study of feasibility and effectiveness for the Nopol repair system. *EMSE*, 24(1):33–67, 2019.

[113] Y. Yuan and W. Banzhaf. ARJA: Automated repair of Java programs via multi-objective genetic programming. *TSE*, 46(10):1040–1067, 2020.

[114] M. Zhang, X. Li, L. Zhang, and S. Khurshid. Boosting spectrum-based fault localization using pagerank. In *ISSTA*, pages 261–272, 2017.

[115] J. Zhou, H. Zhang, and D. Lo. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *ICSE*, pages 14–24, 2012.

[116] Q. Zhu, Z. Sun, Y. an Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *ESEC/FSE*, pages 341–353, 2021.

[117] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. An empirical study of fault localization families and their combinations. *TSE*, 2019.

[118] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang. Evaluating and combining fault localization techniques from different families. https://damingz.github.io/combinefl, 2019. [accessed 4-March-2022].