Program Repair with Repeated Learning

Liushan Chen, Yu Pei, Minxue Pan, Tian Zhang, Qixin Wang, Carlo A. Furia

Abstract—A key challenge in generate-and-validate automated program repair is directing the search for fixes so that it can efficiently find those that are more likely to be correct. To this end, several techniques use machine learning to capture the features of programmer-written fixes. In existing approaches, fitting the model typically takes place *before* fix generation and is independent of it: the fix generation process uses the learned model as one of its inputs. However, the intermediate outcomes of an ongoing fix generation process often provide valuable information about which candidate fixes were "better"; this information could profitably be used to retrain the model, so that each new iteration of the fixing process would also learn from the outcome of previous ones.
In this paper, we propose the LIANA technique for automated program repair, which is based on this idea of *repeatedly* learning the features of generated fixes. To this end, LIANA uses a fine-grained model that combines information about fix characteristics, their relations to the fixing context, and the results of test execution. The model is initially trained offline, and then repeatedly updated online as the fix generation process unravels; at any step, the most up-to-date model is used to guide the search for fixes—prioritizing those that are more likely to include the right ingredients. In an experimental evaluation on 732 real-world Java bugs from 3 popular benchmarks, LIANA built correct fixes for 134 faults (83 ranked as first in its output)—improving over several other generate-and-validate program repair tools according to various measures.

1 INTRODUCTION

Automated program repair (APR) techniques rely on highly-specialized heuristics to efficiently sift through the huge space of all possible program modifications (the so-called "fix space") looking for suitable repairs [18], [60], [13]. A critical choice in designing these heuristics is selecting the right "fix ingredients" [74], [54], [40]: which program elements should be considered for generating the repairs. An effective APR technique must achieve a delicate balance between the size of the fix space (equivalently, the available fix ingredients) and the efficiency with which it can be searched.

This problem is a natural fit for machine learning: rather than selecting fix ingredients explicitly, one can train a statistical model that summarizes the features of programmer-written repairs, and then use the trained model to select repairs using ingredients classified as similar to those previously learned. As we discuss in Sec. 5, numerous recent advances in APR are based on applications of machine learning along these lines [47], [44], [39], [21].

It goes without saying that the effectiveness of a machine-learning model strongly depends on the data that one uses to train it [1], [5]. Applications of machine learning to APR usually train the model on programmer-written fixes *offline*, that is before the program repair process runs and independent of it. This proved to be an effective choice in many situations, but it also limits the flexibility of the resulting APR process. Whereas programmer-written fixes are the gold standard for what a suitable fix should look like [27], collecting a sufficiently varied collection of such fixes may be time-consuming [50]; and the fitted model may be unsuitable for fixing automatically any programs but those that are sufficiently similar to those used for training.

In this paper, we propose LIANA: a novel APR technique that is based on the idea of *repeatedly* updating a statistical model *online* based on the intermediate validation results of an ongoing program repair process. At its core, LIANA (Learn It AgaiN for APR) implements a fairly standard test-driven generate-and-validate (G&V) program repair loop, which explores the fix space incrementally and uses the available tests to validate any candidate fixes it generate. On top of that, LIANA introduces a learning-to-rank model to capture the features of fixes that are similar to those deployed by programmers (or at least that make some progress towards passing all available tests), and uses the model's information to guide the search towards generating candidate fixes that have similar characteristics. For example, if candidate fixes that introduce a conditional statement tend to pass tests that the original program failed, LIANA will prioritize generating new candidates that also introduce a similar conditional.

As in other APR approaches that use machine learning, the statistical model can be trained offline on any collection of fixes and given to LIANA as input. The key novelty of LIANA is that it also *updates* the model based on the outcome

- *Liushan Chen is with ByteDance Ltd. This work was done during her PhD at the Department of Computing, The Hong Kong Polytechnic University. E-mail: ls.chen@connect.polyu.hk.*
- *Yu Pei is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China. E-mail: csypei@comp.polyu.edu.hk.*
- *Minxue Pan is with the State Key Laboratory for Novel Software Technology and the Software Institute of Nanjing University, China. E-mail: mxp@nju.edu.cn.*
- *Tian Zhang is with the State Key Laboratory for Novel Software Technology and the Department of Computer Science and Technology of Nanjing University, China. E-mail: ztluck@nju.edu.cn.*
- *Qixin Wang is with the Department of Computing, The Hong Kong Polytechnic University, Hong Kong, China. E-mail: csqwang@comp.polyu.edu.hk.*
- *Carlo A. Furia is with the Software Institute of USI Università della Svizzera italiana, Lugano, Switzerland. Homepage: https://bugcounting.net/.*
- *Yu Pei is the corresponding author.*

Received: revised:

of validating any newly generated candidate fixes. More precisely, any candidate fix that passes *some* tests that were previously failing is considered a step in the right direction; LIANA automatically uses it to update its machine-learning model. The idea of using the results of intermediate validation to guide successive iterations of APR was also used by earlier, influential APR techniques such as GenProg [72] and HDA [30]. LIANA extends this idea by introducing a fine-grained statistical model that ranks candidate fixes in terms of features combining static and dynamic information about them. For example, it measures the number of variables that a candidate fix uses that are also declared locally to the method under fix; the number of tests that the originally faulty program fails that a candidate fix passes instead; and the similarity between method invocations in a candidate fix and other candidate fixes that pass validation. LIANA's approach also supports reusing the model learned while fixing a bug to repair another bug in a later run of the algorithm.

We implemented the LIANA technique into a tool with the same name based on the RESTORE generate-and-validate APR tool [80]. LIANA inputs a buggy Java program, a set of test cases (including at least one that triggers the bug under repair), and an initial fix ranking model, which was trained offline on any reference fixes or online during previous runs of the tool. It outputs a list of valid fixes (patches to the program that make all tests pass) and the latest fix ranking model, updated based on the information collected automatically while running the tests.

To evaluate the effectiveness and efficiency of LIANA, we ran it to repair 732 bugs from 3 popular benchmarks in APR, namely DEFECTS4J [27], INTROCLASSJAVA [14], and QUIXBUGS [36]. LIANA built valid fixes for 219 bugs and correct fixes (semantically equivalent to those written by programmers for the same bugs) for 134 bugs; specifically, it correctly fixed 53 bugs in DEFECTS4J, and a correct fix was ranked first in the output for 28 DEFECTS4J bugs. As we discuss in detail in Sec. 4.4—where we compare LIANA to 21 other state-of-the-art APR tools for Java—these results put LIANA among the most effective tools in terms of number of bugs fixed. To demonstrate the generality of the LIANA technique, we also implemented it atop the SIMFIX APR technique [26]. SIMFIX with LIANA fixes only two more DEFECTS4J bugs than "plain" SIMFIX, but does so considerably more quickly (2.7x speedup on average).

In summary, this paper makes the following contributions:

1) The LIANA technique for improving the effectiveness and efficiency of search-based automated program repair by repeatedly learning a fix ranking model based on the intermediate results of the fixing process;
2) An implementation of the LIANA in a tool based on the RESTORE APR tool;
3) A large-scale experimental evaluation of LIANA on 732 bugs from 3 different benchmarks.

Terminology. We use the terms "defect", "bug", "fault", and "error" as synonyms. We also use "fix", "patch", and "repair" as synonyms to denote changes to a program's source code.

Availability. The LIANA tool and a replication package are available at http://www4.comp.polyu.edu.hk/~csypei/download/LIANA-TSE.zip.

```
1  public static double linearCombination(double[] a, double[] b)
2        throws DimensionMismatchException {
3     final int len = a.length;
4     if (len != b.length) {
5        throw new DimensionMismatchException(len, b.length);
6     }
7
8     final double[] prodHigh = new double[len];
9
10    for (int i = 0; i < len; i++) {
11       final double ai = a[i];
12       final double bi = b[i];
13       prodHigh[i] = ai * bi;
14       ...
15    }
16    ...
17    double prodHighNext = prodHigh[1];
18    ...
19 }
```

(a) Faulty method `linearCombination` from class `MathArrays` in project *Commons-Math*.

```
if (len == 1)
   return a[0] * b[0];
```

(b) Fix written by developers (inserted at line 7 in Fig. 1a).

```
if (prodHigh.length == 1)
   return prodHigh[i];
```

(c) Correct fix generated by LIANA (inserted before line 15 in Fig. 1a).

Fig. 1: A faulty method in project *Commons-Math* that LIANA repairs with a fix functionally equivalent to that written by the developer.

```
if (a[i] != b[i])
   return prodHigh[i];
```

(a)

```
if (a[i] == a[i])
   return prodHigh[i];
```

(b)

Fig. 2: Two "progress" fixes generated by LIANA (inserted before line 15 in Fig. 1a), which are incorrect but help direct the search in the space of fixes towards the correct one.

2 AN EXAMPLE OF LIANA IN ACTION

Apache Commons Math's class `MathArrays` provides a wide range of functions to work with numerical arrays. Method `linearCombination` of the class inputs two **double** arrays `a` and `b` of the same length, and returns the scalar product $\sum_k$ `a[k] * b[k]`.

Fig. 1a outlines the implementation of this method in an older release of the library. The method's body first checks if `a` and `b` have the same length; if not, it returns with an exception. Otherwise, the method implements specific multiplication and addition algorithms to preserve accuracy and reduce the chance of numerical errors. However, Fig. 1a's implementation is faulty [27, Fault *Math3*]: when `a` and `b` are singletons, the statement on line 17 triggers an `ArrayIndexOutOfBoundsException`. To correct the bug in later versions of the library, the developers inserted at line 7 the conditional statement in Fig. 1b.

The LIANA technique described in this paper can automatically produce the fix in Fig. 1c, which also correctly fixes the bug in Fig. 1a. While LIANA's fix is applied at a later location—and is therefore slightly less efficient—it is semantically equivalent to the programmer-written one.

Automatically generating a correct fix for this bug is challenging: the fix requires to find both a conditional ex-

pression and a returned expression; the method is fairly large (60 lines of code), which means it includes a high number of "fix ingredients" to choose from for constructing fixes. For example, 15 variables are in scope at line 15; simply enumerating all possible combinations of `boolean` expressions (for the `if` condition) and `double` expressions (for the `return` statement) built out of these variables would be infeasible.

Indeed, to our knowledge, only three APR tools can correctly fix this bug at the time of writing. LIANA outputs a correct fix in around 70 minutes; JAID can also build a correct fix, but takes over 14 hours to do so. This difference in efficiency is even more striking considering that LIANA's search space is a considerably larger superset of JAID's; and that RESTORE includes this fix in its fix space, but runs out of time before its heuristics can find it. ACS is the third tool capable of fixing this bug; key to its success is that it specializes in synthesizing conditional fixes, like the one required to correct Fig. 1a's bug. The behavior of these three tools illustrates the relation between fix space, search efficiency, and overall effectiveness of an APR technique. ACS targets a smaller fix space, so that it can quickly find fixes such as those in the example; this choice also entails that generating different kinds of fixes are outside its capabilities. JAID targets a broader fix space, but it may struggle to sift through it in a reasonable amount of time; LIANA can generate several different kinds of fixes, and still often manages to search efficiently an heterogeneous and large fix space.

To this end, LIANA builds a *fix ranking model*, which identifies fix ingredients that are more likely to be useful. Crucially, the model is updated as LIANA's search progresses, so that it can learn from previous, partially successful, fixing attempts.

When running on the code in Fig. 1a, LIANA initially generates a batch of candidate fixes that includes, among others, the two in Fig. 2. Neither of them is correct; however, they are a step in the right direction since they pass a test that the buggy implementation fails, thus hinting at a suitable fixing location and at the necessary fixing ingredients (including using a conditional and the correct `return` expression). LIANA uses such "progress fixes" (incorrect but passing *some* originally failing tests) to retrain the fix ranking model so that similar candidate fixes will be generated in the following iterations. Indeed, LIANA's second batch of candidate fixes includes Fig. 1c's correct fix and ranks it in the top position. Since this fix also passes all available tests, it is output to the user. Overall, the ranking information learned by repeated iterations of LIANA's algorithm is useful both to direct the search towards candidate fixes that are more likely to be correct and to output to the user such fixes in a prominent position.

3 HOW LIANA WORKS

Fig. 3 gives an overview of the main steps of LIANA's program repair process. The process's backbone—corresponding to the blue boxes in the top row of Fig. 3—is similar to any standard G&V program repair technique: given a Java program P and a test suite T that triggers (at least) one failure in P, LIANA first performs fault localization to identify suspicious entities (locations or expressions) in P that are implicated with the failure. Then, it generates a number of candidate fixes; each candidate fix targets one of the suspicious entities in P identified by fault localization. Fix generation is based on heuristics, and hence it's *best effort*; therefore, it is followed by a fix *validation* step, where LIANA reruns all tests on all candidates. Valid fixes V are LIANA's final output to the user.

Validity and correctness. In the remainder, we call a candidate fix *valid* if it passes all the available tests for the program under repair. Valid fixes are also called *plausible* [46] or *test-suite adequate* [50]. A valid fix is *correct* if it is also semantically equivalent to the one written and deployed by the project maintainers for the same bug.

Fix space exploration. To efficiently explore the space of all possible candidate fixes (the so-called *fix space*), LIANA guides the fix generation process by means of a *fix ranking model*: a statistical model that ranks a candidate fix the higher the more likely it is valid. As shown by the green boxes in the bottom row of Fig. 3, each run of LIANA starts with an initial ranking model R (which typically comes from previous runs of the technique on other bugs), and uses it to generate a first batch of candidate fixes. Then, it iteratively updates the ranking model based on the features of the candidates it has generated and validated. To this end, it learns the features of all candidates that make some *progress* towards building a valid fix (that is, such that they pass some more tests in T), so that additional similar fixes are likely to be generated. As we detail in Sec. 3.3, "similarity" is measured in terms of 17 different features that mainly capture the syntactic "distance" between two programs using static and dynamic measures. In every round of fix generation and validation, LIANA always uses the most up to date ranking model R—which incorporates all information about the progress made so far.

The repair process terminates after a timeout (or when the fix space is exhausted). In addition to any valid fixes, LIANA's final output also includes the latest fix ranking model, which can be used to start future program repair runs.

The rest of this section describes the main steps of this process in detail, with a focus on those that are specific to LIANA's approach. Alg. 1 outlines in pseudo code the same steps and how they are combined. The presentation targets a run of LIANA on a program P, given a collection T of tests. T is partitioned into failing tests $T_{\text{✖}}$ and passing tests $T_{\checkmark}$; we assume that $T_{\text{✖}}$ is not empty—that is, P is buggy.

3.1 Fault Localization, Fix Generation, and Validation

The backbone of LIANA's process covers the usual steps of a G&V program repair technique: fault localization, fix generation, and fix validation. We refer to these steps as the "standard (repair) process". The specific contributions of LIANA complement the standard process with information that helps explore the fix space more effectively. These contributions are largely independent of the details of how the standard process works; hence, we only describe the standard process succinctly and at a high level, focusing instead on LIANA's peculiarities in the rest of the section. This also implies that LIANA is applicable in principle on top of a wide range of G&V program repair techniques;

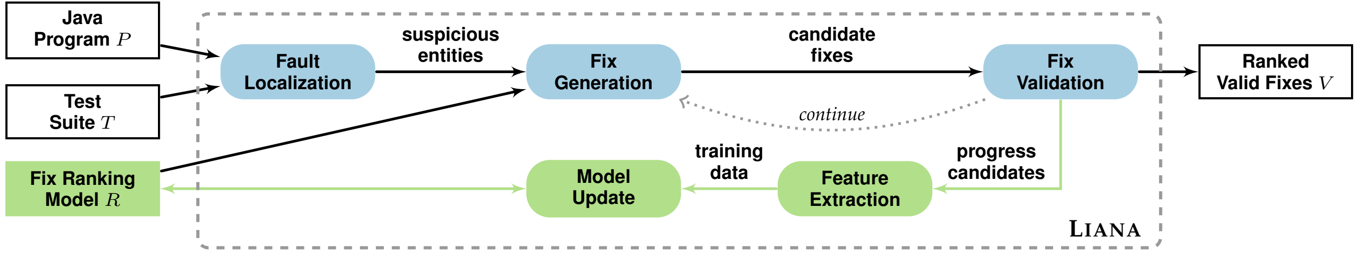


Fig. 3: Overview of how LIANA works. Given as input a Java program P and a test suite T, LIANA goes through the three main steps of fault localization, fix generation, and fix validation that are common in all G&V program repair techniques. Fix generation is done incrementally and guided by a *fix ranking model* R: a statistical model that assigns higher ranks to fixes that are more likely to be valid (and possibly correct). After each round of fix validation, LIANA updates the ranking model to incorporate the outcome of validation; and it uses the most up-to-date ranking model R in each round of fix generation. The final outputs are any valid fixes V (which pass all tests in T), as well as the latest update of the fix ranking model R.

Algorithm 1 How LIANA works in pseudo code.

```
Input
    R: an existing fix ranking model
    P: a buggy program to repair
    T: passing T✔ and failing T✖ tests for P
Output
    V: a (possibly empty) list of valid fixes
    R: an updated fix ranking model

 1: candidates ← []          /* list of candidate fixes, initially empty */
 2: V ← []                       /* list V of valid fixes, initially empty */
 3:                             /* list ℒ of suspicious program entities */
 4: ℒ ← FAULT_LOCALIZATION(P, T)
 5:          /* while there are suspicious entities or candidate fixes */
 6: while ℒ ≠ ∅ ∨ candidates ≠ ∅ do
 7:          /* select a new batch of n suspicious entities L from ℒ */
 8:     L ← PICK(ℒ, n)
 9:                      /* enumerate all candidate fixes for new entities */
10:     fixes ← FIX_GENERATION(P, L)
11:     candidates ← candidates + fixes
12:                      /* sort candidates according to ranking model */
13:     ranked ← RANK(candidates, R)
14:              /* progress fix discovered in this round of validation */
15:     progress ← []
16:                          /* for each of the top-m ranked candidates */
17:     for all f in ranked[0:m] do
18:         if TIME() > timeout then return ⟨V, R⟩ end if
19:         candidates ← candidates − [f]
20:             /* if f passes at least one failing test ("progress fix") */
21:         if PROGRESS?(f, P, T✖) then
22:             progress ← progress + [f]
23:                             /* if f passes all tests T it is valid */
24:             if VALID?(f, T) then
25:                 V ← V + [f]
26:             end if
27:         end if
28:     end for                 /* if there is at least one progress fix */
29:     if progress ≠ ∅ then
30:         /* update ranking model R by learning from 'progress' */
31:         R ← UPDATE(R, progress)
32:     end if
33: end while
34: return ⟨V, R⟩
```

the experiments in Sec. 4.4.4 will substantiate this claim of generality.

3.1.1 Fault Localization

Fault localization produces a list $\mathcal{L}$ of program entities (statements or expressions) that are suspicious: they are possible causes of the fault that is being repaired. Changing any of the entities in $\mathcal{L}$ has a chance of repairing the program. Precisely, a program entity $\ell \in \mathcal{L}$ corresponds to a triple $\langle n, e, v\rangle$: n denotes a statement within the program by its location number; e is an expression that is in scope at n; and v is a value of that expression. Intuitively, the triple points to a program run where $e = v$ when statement n executes. In the running example, the list $\mathcal{L}$ of entities identified by LIANA's fault localization includes various triples $\langle n_{15}, e, v\rangle$, where n_{15} identifies line 15 in Fig. 1a.

The current implementation of LIANA uses a spectrum-based fault localization algorithm to identify suspicious entities in the input program P. More specifically, LIANA reuses JAID's fault localization [7], [8], which implements Wong et al.'s Heuristic III [15], and combines static (expression dependence, which gives higher suspiciousness values to program entities that are syntactically similar to those implicated with a failure) and dynamic (spectral information, which gives higher suspiciousness scores to program entities that are evaluated more often in failing than in passing runs) information about the program. In LIANA, fault localization plays a less central role than in JAID, since LIANA only uses it to boostrap the fix generation process, after which it relies on the ranking model to select program entities.

3.1.2 Fix Generation

Given a batch $L \subseteq \mathcal{L}$ of suspicious program entities (selected out of all suspicious entities $\mathcal{L}$), fix generation enumerates all repair actions $\mathcal{A}$ that are applicable to any of the entities in L. Repair actions are transformations that try to change program behavior at the location where they are applied. Applying a repair action $a \in \mathcal{A}$ to an entity $\ell \in L$ gives a *candidate fix*: a patched version of program P. The set of all candidate fixes that could be generated is the *fix space* of P. Thus, the fix space depends on the fault localization technique (which populates $\mathcal{L} \supseteq L$) and on the repair actions $\mathcal{A}$ that are available.

LIANA only generates fixes that modify a single entity ℓ in program P: while a repair action may introduce complex

```
Schema A: snippet; oldStatement;
Schema B: if(e == v) { snippet; } oldStatement;
Schema C: if(e != v) { oldStatement; }
Schema D: if(e == v) { snippet; } else { oldStatement; }
Schema E: /* oldStatement; */ snippet;
```

Fig. 4: Schemas to build candidate fixes given a fix action a derived from entity $\langle n, e, v \rangle$.

modifications to its target entity, LIANA does not currently support multi-hunk (multi-location) fixes [68].

Fix ingredients. LIANA's generation of candidate fixes from suspicious entities follows a process that extends RESTORE's [80][1] with a new fix ingredient α_5. First, given a suspicious entity $\ell = \langle n, e, v \rangle$, LIANA enumerates (repair) actions (called *fix actions* in [80]) that:

- α_1: modify the state of an object referenced by the suspicious program entity e (for example, if e is an integer variable `x`, an action of this kind is `x = x + 1`);
- α_2: modify a subexpression of e (for example, if e is the expression `x == 1`, an action of this kind changes that expression to `x <= 1`);
- α_3: if the statement at n is a conditional `if (c)`, modify its condition `c`;
- α_4: modify the control flow at n (for example, adding a `return` statement at n);
- α_5: if e includes a call to some method m with actual arguments a, replace it with a call to another method that is type-compatible (for example, if e is the expression `y.get(2)` that retrieves the element at index 2 in a list `y`, an action of this kind changes that expression to `y.remove(2)`).

An action a determines a *new statement* `snippet` as follows: if a is of kinds α_1 and α_4, `snippet` is the action itself (for example, an assignment for α_1 and a `return` for α_4); if a is of kinds α_2, α_3, and α_5, `snippet` is the statement at location n modified according to a.

After enumerating several repair actions, LIANA generates candidate fixes by weaving the actions into the program under repair. To this end, it instantiates any[2] of the five schemas listed in Fig. 4, where `oldStatement` is the statement at location n in the buggy program, and `snippet` is the new statement determined by the action as described above. The candidate fix consists of the whole instantiated schema replacing the statement at location n in the program.

The combinations of actions and schemas determine the so-called *fix ingredients*, that is the possible modifications of the program that LIANA can introduce.

Fix space. In turn, the fix ingredients determine LIANA's space of all possible fixes, which is a strict superset of RESTORE's: LIANA supports all of RESTORE's repair actions (repair actions α_1, α_2, α_3, α_4), but can also generate fixes that replace method calls (repair action α_5). Incorrect method calls are a relevant category of bugs targeted by automated program repair [67], which LIANA is equipped for.

In the running example, the first iteration of LIANA generates several repair actions for location n_{15}, including statement `return prodHigh[i]` of kind α_4. Weaving this action into method `linearCombination` using schema B produces, for program entities $\langle n_{15}, $ `a[i] == b[i]`$, $ `false` $\rangle$ and $\langle n_{15}, $ `a[i] == a[i]`$, $ `true` $\rangle$, the two candidate fixes in Fig. 2.

1. Sec. 3.5.2 details the differences between LIANA and RESTORE.
2. The exceptions are actions of kind α_5: since they already target a conditional, LIANA only uses schema E to weave them into the program.

3.1.3 Fix Validation

In G&V program repair, fix generation is "best effort": there is no guarantee that any of the candidate fixes is indeed an improvement over the original program P. Therefore, fix validation runs all available tests T on every candidate fix f. If f passes all tests, it is *valid* and output to the user.

In the running example, neither candidate fixes in Fig. 2 is valid. However, both pass *some* tests in $T_{\text{✖}}$—which were failing in the program P under repair. This information about "partial validation" is used by LIANA's ranking model as we describe below.

3.2 Fix Ranking Model

The fix space is usually too large to be generated exhaustively ("combinatorial explosion"); even if it were, exhaustive validation would be infeasible since rerunning all tests takes even more time than generating fixes. Therefore, a key component of any effective program repair technique is heuristics to effectively explore the most promising parts of the fix space. LIANA generates candidates in batches; a fix ranking model determines which candidates are generated in each batch.

A *fix ranking model* is a statistical classifier that can be used to rank candidate fixes. As we have discussed before, a candidate f is the result of applying a repair action $a \in \mathcal{A}$, using a certain schema $s \in \mathcal{S}$, to an entity $\ell \in \mathcal{L}$ identified by fault localization. Therefore, we can think of the fix ranking model R as a function $R\colon \mathcal{L} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$ that assigns a nonnegative score to any candidate fix $f = (\ell, a, s)$. R should capture how similar a candidate fix f is to a valid (ideally, correct) fix. In each iteration, LIANA generates a new batch of candidate fixes corresponding to those with the highest ranking score as determined by R.

Learning to rank model. LIANA implements the ranking model R using learning-to-rank techniques [42]. Learning to rank is a family of supervised machine learning algorithms to learn ranking functions. LIANA uses a listwise ranking algorithm and normalized discounted cumulative gain (NDCG) as a measure of ranking quality. Previous applications of learning to rank to fault localization used pairwise ranking [4], [34] to exhaustively rank each element compared to all others. For our purposes, we mainly need to distinguish between high-rank and low-rank candidates; therefore, a listwise algorithm is a natural choice. NDCG also fits well LIANA's application of learning to rank: under discounted cumulative gain, top-ranked candidates have a disproportionate impact on determining the ranking quality; and indeed, the rest of the fixing process will only use candidates that are ranked high enough.

3.3 Features and Feature Extraction

The fix ranking model R assigns a score to a candidate fix $f = (\ell, a, s)$ based on f's program *features*. LIANA currently extracts 17 different features, which characterize f according to how it modifies P and how it compares to the progress fixes that have been generated in previous batches.

TABLE 1: Features used by LIANA's fix ranking model to characterize candidate fixes. The description refers to a candidate fix $f = (\ell, a, s)$ that applies a repair action a, using schema s, to the entity ℓ (location or expression) reported by fault localization within the whole program P under repair. The repair action a is a collection of edits (each deleting, replacing, adding, or moving some code). "Progress fixes" are the candidate fixes generated in previous batches that pass at least one of the tests $T_{✖}$ that P fails. Column EX-VAL lists the concrete values of each feature for the correct fix of Fig. 1c after generating the first batch of progress fixes. Missing values are for features that are undefined in the specific example (e.g., φ_{17} is undefined since there are no method calls in any of those progress fixes).

#	FEATURE	DESCRIPTION	EX-VAL
φ_1	*n-calls*	normalized number of operators and method calls introduced by a	0.100
φ_2	*dep-vars*	fraction of all variables in P and f affected by a that are pairwise correlated	0.152
φ_3	*dep-edits*	fraction of all variables in each edit affected by a that are pairwise correlated	–
φ_4	*used-vars*	fraction of all variables introduced by a that are also elsewhere in the method under fix	1.000
φ_5	*nodes*	fraction of all AST nodes introduced by a that are also elsewhere in the method under fix	1.000
φ_6	*calls*	fraction of all operators introduced by a that are also elsewhere in the method under fix	0.055
φ_7	*sim-code*	similarity (as in [76]) between the code introduced by a and any code snippets in P	0.500
φ_8	*sim-calls*	similarity (as in [76]) between method calls introduced by a and any method invocations in P	–
φ_9	*susp*	suspiciousness score of ℓ (determined by fault localization)	1.002
φ_{10}	*progress*	progress score of ℓ (the higher, the more progress fixes target the location of ℓ)	1.000
φ_{11}	*prog-vars*	fraction of all variables used by any progress fixes that are also introduced by a	0.436
φ_{12}	*prog-expr*	fraction of all expressions used in any progress fixes that are also introduced by a	0.431
φ_{13}	*prog-edits*	fraction of all edit descriptors in any progress fixes that are also used by a	1.000
φ_{14}	*prog-modified*	fraction of all expressions modified by edits in any progress fixes that are also targeted by a	1.000
φ_{15}	*prog-removed*	fraction of all expressions removed by edits in any progress fixes that are also removed by a	–
φ_{16}	*prog-added*	fraction of all expressions added by edits in any progress fixes that are also added by a	0.610
φ_{17}	*prog-calls*	fraction of all method calls introduced by any progress fixes that are also introduced by a	–

Tab. 1 briefly describes the features; here, we discuss what information they capture and give a few details about how they are computed. The experiments described in Sec. 4.4.3 will empirically assess each feature's impact on LIANA's effectiveness.

The chosen features try to characterize the candidate fixes from different angles and at different levels of granularity (for instance, variables, expressions, and methods). At a high level, the features capture a notion of "syntactic distance" both between a candidate and other candidates (e.g., φ_{11} counts the variables that both use), and between a candidate and its "context" within the whole program (e.g., φ_7 measures the similarity between the code introduced by the candidate fix and the rest of the program). Some features (1–8) measure how a candidate fix relates to the different parts of the program it modifies; other features (9–17) compare a candidate fix to other candidates generated in previous batches. All features are based on static and dynamic measures that are commonly used for fault localization [62], program analysis [61], and program repair [51], [54]. Sec. 4.4.3 describes the results of an ablation study, which justifies the selection of features a posteriori and demonstrates that LIANA's overall effectiveness is generally robust with respect to the choice of features.

LIANA uses the GumTree algorithm [16] to represent a fix f as an *edit script*: a hierarchical collection of *edits* that, when applied to the entity ℓ, transform P into the fix f. A single edit modifies the program's abstract syntax tree (AST) by adding a new node, deleting an existing node, modifying an existing node, and moving an existing node to another part of the tree.

Fix-program features. Features 1–8 capture how the modifications introduced by f relate to (different parts of) the modified program P. Feature φ_1 simply counts the operators and method calls that appear in the code introduced by fix f into P. Features φ_2 and φ_3 consider all variable pairs v_1, v_2 such that v_1 appears in P at ℓ (before fix action a is applied) and v_2 appears in the corresponding spot in f (after fix action a is applied); φ_2 measures the fraction of such pairs for which v_1 and v_2 are *correlated*, that is, either there is a data dependency between the two variables, or they belong to the same basic statement or branching condition. φ_3 is a similar measure but for variable pairs that belong to each edit in f's edit script. Features φ_4, φ_5, and φ_6 measure different program elements that appear both in the code introduced by a in f and elsewhere in its fixing *context*—that is, in the rest of the method that is being fixed. φ_4 targets variables; φ_5 targets nodes in the AST representation of the code; and φ_6 targets operators. Features φ_7 and φ_8 measure syntactic similarity in an even broader context, namely between code introduced by a in f and the whole program P under repair (even beyond the method where fixing takes place). φ_7 considers all code snippets, whereas φ_8 zooms in on method calls; both use the same notion of syntactic similarity used by [76].

As a concrete example, take φ_4: this feature has value 1 for all three candidate fixes in Fig. 1c and Fig. 2, since all variables that appear in the candidates' code (`len`, `a`, `b`, and `prodHigh`) are declared within the method `linearCombination` under repair. Another example is φ_1, which has value 0.1 for Fig. 1c and Fig. 2 in the first batch of fixes: each candidate fix includes exactly one operator (`==` or `!=`), which gives a normalized score of $1/m$, where $m = 10$ is the total number of operators and method calls that appear in the candidate fixes of the first batch.

Progress fix features. In contrast to features 1–8, which compare the code modified by a candidate fix to the rest of the program, features 9–17 capture similarities between a candidate fix f and other progress fixes. Remember that a progress fix is one that passes at least one test in $T_{✖}$—tests which the program under repair P all fails. Feature φ_9 is simply the suspiciousness that fault localization computes for each program entity ℓ; all candidate fixes that modify ℓ have the same value. The remaining features all

measure the similarity between f and other progress fixes by determining how common the different features of f are in the pool of progress fixes found so far. Feature φ_{10} quantifies "how much progress" f makes: first, program locations are ranked according to how many candidate fixes that target each location are also progress fixes; then, φ_{10} is $1/N$ for f, where N is the position of the program location that f modifies in this ranking. Like for φ_9, all candidate fixes that modify the same location have the same value of φ_{10}. φ_{11} examines variables; φ_{12} examines individual sub-expressions; and φ_{17} examines method calls. Features 13–16 have a different level of granularity since they all examine the individual edits in f's edit script compared to the other progress fixes'. φ_{13} looks at the edit descriptors, which classify an edit according to: whether it adds, deletes, modifies, or moves a node; and the statement type it applies to (e.g., a conditional or an assignment). φ_{14} looks at the expressions that are modified by any edits (they change from P to f); φ_{15} at the expressions that are removed by any edits (they appear in P but not in f); and φ_{16} at the expressions that are added (they don't appear in P but do appear in f).

The rightmost column in Tab. 1 lists the concrete values of the features for the running example's fix in Fig. 1c, after generating the first batch of (progress) candidate fixes (on which several features are based). The shown values indicate that the candidate is quite similar to other progress fixes, and hence its components seem useful to direct the search towards fixes that pass more test; in addition, most progress fixes modify the same location (line 7 in Fig. 1a) that the running example's fix modifies (hence, $\varphi_{10} = 1$).

3.4 Model Update

The ranking model can be updated by providing a new candidate fix labeled with a ranking score. Precisely, a candidate fix f is characterized by a value for each of its features (Tab. 1); and the score is a nonnegative number r the higher the closer the fix is to being correct. Therefore, a labeled datapoint to update R is a tuple $\langle \varphi_1, \varphi_2, \ldots, \varphi_{17}, r \rangle$. The score r is: (a) 10 if f is correct (equivalent to the programmer-written fix); (b) 5 if f is valid (it passes all tests in T) but not correct; (c) 3 if f is progress (it passes at least one test in $T_{✖}$) but not valid; (d) 1 if f is not progress (it fails all tests in $T_{✖}$).

We chose these scores following some basic principles: (i) *qualitative order*: a correct fix should have a higher score than a merely valid one, which should have a higher score than a progress but invalid one, which should have a higher score than a non-progress one; (ii) *correct* $\gg$ *valid*: the score difference between a correct fix and a valid but incorrect one should be larger than the score difference between a valid fix and a progress, invalid one (after all, finding correct fixes is the ultimate goal of the ranking process); (iii) *standard range*: the overall range of scores should be close to the scores normally used in applications of the learning-to-rank algorithm that we use [25]. In Sec. 4.4.3, we discuss some experiments that indicate that LIANA's overall effectiveness is not affected by small changes in the absolute values of the scores.

As outlined in Fig. 3, LIANA updates the ranking model automatically when some progress fixes were found in the previous batch of fix validation. Namely, it computes the values $v_1, \ldots, v_{17}$ of Tab. 1's features for the new progress fix, assigns it a score r according to whether it is valid or not, and retrains the ranking model using the new labeled datapoint $\langle v_1, \ldots, v_{17}, r \rangle$. In this case, the score r assigned to a new progress fix can only be 5 or 3, since we cannot determine automatically which valid fixes are also correct. We only use progress fixes for updating the model since also including non-progress candidates (that still fail all originally failing tests) would drown the model with lots of "noise"; instead, focusing on the candidates that do pass more tests provides a more direct guidance about which candidates to generate next. In addition to the model updates that take place while LIANA runs, users can provide additional training data offline. This data may also include fixes that have been established to be correct, and hence are labeled with $r = 10$.

In the running example, LIANA updates its fix ranking model with new datapoints in successive iterations. Some datapoints correspond to progress fixes, such as those in Fig. 2, which receive a score of 3. One datapoint is $\langle \text{EX-VAL}, 5 \rangle$, where EX-VAL are the values in Tab. 1's column with the same name, for the candidate fix in Fig. 1c which is also valid (hence, the score of 5). Since this fix is also the first candidate that passes validation, LIANA will output it to the user first.

3.5 Implementation Details

We implemented the LIANA technique into a tool with the same name, based on the RESTORE program repair tool [80]. RESTORE is a recent G&V program repair tool for Java, whose implementation is publicly available [66]. LIANA reuses RESTORE's implementation as its backbone (blue boxes in Fig. 3). While using RESTORE was a convenient choice, LIANA's technique is not tied to any of RESTORE's features; in fact, Sec. 4.4.4 discusses a different implementation of LIANA which is based on the SIMFIX repair tool [26].

As discussed in Sec. 3.1.1, LIANA reuses JAID's spectrum-based fault localization algorithm to perform fault localization. Then, it generates an initial batch of candidate fixes, which go through validation. Any progress candidates are used to update the fix ranking model. Each following round of fix generation generates candidate fixes targeting 10% more suspicious program entities (identified by fault localization); then, LIANA ranks all candidates using the ranking function and feeds the top-30% ranked candidates through validation.

In addition to a timeout, RESTORE's implementation also uses at most 1500 suspicious program entities in each run. LIANA's implementation lifts this cap, since we found that it can often search efficiently through a large number of entities thanks to the ranking model's guidance. This difference in the implementation of LIANA reflects its larger fix space compared to RESTORE's.

The implementation of LIANA uses the WALA framework [71] to perform static analysis.

3.5.1 Ranking Model Implementation

LIANA's ranking model is implemented using XGBoost [9], a widely-used open-source implementation of the gradient

TABLE 2: A high-level overview of the similarities and differences between the APR techniques JAID, RESTORE, and LIANA. Each tool is associated with a color (green for LIANA, red for RESTORE, and blue for JAID) and a column; an items in each tool's implementation is colored according to which techniques it originates from. For example, LIANA uses the same state abstraction as JAID, the same validation process as RESTORE, but its own original ranking model, which it also uses to extend JAID's fix generation process.

		JAID	RESTORE	LIANA
state abstraction		snapshots (entities)	= JAID	= JAID
fault localization	*input*	program + tests	JAID + partial validation	= JAID
	algorithm	spectrum-based	mutation-based	= JAID
fix ingredients	*from*	target method	JAID + target class	RESTORE + method calls
fix generation	*in order of*	suspiciousness (fault localization)	= JAID	JAID + ranking model
validation	*full/partial*	full (all tests)	JAID + partial (failing tests)	= RESTORE
ranking	*based on*	suspiciousness (fault localization)	= JAID	ranking model

boosting machine learning algorithm. In a nutshell, XGBoost uses decision tree ensembles to represent classifying information that it learns from labeled examples (supervised learning). This statistical model is highly customizable to encode different classification tasks and error models, including the listwise learning-to-rank model with normalized discounted cumulative gain (NDCG) used by LIANA for ranking candidate fixes. XGBoost also supports "training continuation", which refines a learned model with new datapoints without having to train it from scratch. LIANA uses this feature to update the fix ranking model (Sec. 3.4) with new progress candidate fixes incrementally as new candidates are generated.

We followed standard practices [65] to tune the hyperparameters to be used with XGBoost. First, we selected, using random search, a group of 30 hyperparameters that produced generally high NDCG scores. Then, using grid search [65], we looked for values adjacent to these hyperparameters that produced the highest NDCG score, and used the values in our experiments.[3]

3.5.2 LIANA, RESTORE, *and* JAID

LIANA 's implementation is based on RESTORE, and RESTORE extends JAID. After presenting the details of how LIANA works, we are in a good position to explain the relations between these three tools.

Tab. 2 gives an overview of the similarities and differences in terms of state abstraction, fault localization, fix ingredients and generation, validation, and ranking. All three tools use *program entities* (called *snapshots* in JAID [7]) to abstract program states in terms of a location and a (sub)expression to modify. JAID and LIANA both use a spectrum-based algorithm for fault localization, which summarizes the execution paths of the program under fixing on the given tests; RESTORE, in contrast, introduced a form of mutation-based fault localization (called retrospective fault localization), which also uses the information from partial validation.

The fix space of JAID is the smallest of the three, as it only includes fix ingredients from the target method (under repair); RESTORE extends it with fix ingredients from the whole target class; LIANA further extends it by supporting fixes that introduce new method calls (Sec. 3.1.2). The fix generation order is based on suspiciousness in both JAID and RESTORE (but their suspiciousness scores are in general different); LIANA refines the suspiciousness order using a fix ranking model, which is updated dynamically based on the information from partial validation (Sec. 3.2). Validation is full in JAID, as it always runs all available tests; RESTORE performs partial validation, using only failing tests, to support its fault localization process; LIANA also performs partial validation, but uses it to update its ranking model (Sec. 3.4). Valid fixes are ranked by both JAID and RESTORE according to their suspiciousness (which, again, differ in general in the two techniques); LIANA's key contribution is a fix ranking model that is used to guide fix generation and validation (Sec. 3.2).

4 EXPERIMENTAL EVALUATION

We experimentally evaluated the effectiveness and efficiency of the LIANA tool using Java bugs from 3 popular benchmarks. Our experiments address the following research questions:

RQ1: How effective and efficient is LIANA in repairing faulty Java programs? RQ1 assesses LIANA's effectiveness (correct fixes) and performance, and compare it directly to RESTORE (on which LIANA's implementation is based).

RQ2: How does LIANA compare with existing APR tools in repairing faulty Java programs? RQ2 compares LIANA's effectiveness to the state-of-the-art APR tools for Java.

RQ3: How useful are the individual features of LIANA's fix ranking model? RQ3 investigates the different impact of the 17 features used by LIANA's fix ranking model.

RQ4: Is LIANA generally applicable to G&V APR techniques? RQ4 looks for evidence that the LIANA technique is applicable to G&V APR techniques other than RESTORE.

3. LIANA uses XGBoost with the following non-default hyperparameter values: `objective = rank:ndcg`, `eval_metric = ndcg`, `max_depth = 8`, `min_child_weight = 0.1`, `min_split_loss = 0`, `num_boost_round = 200`, `reg_lambda = 1`, `reg_alpha = 0`, `learning_rate = 0.2`.

4.1 Subject Faults

Our experiments target all 732 faults from 3 widely used benchmark collections of Java bugs: DEFECTS4J [27], INTROCLASSJAVA [14], and QUIXBUGS [36].

DEFECTS4J [27] (revision #895c4e6) includes 395 bugs from 6 open source Java projects (*Chart*, *Closure*, *Lang*, *Math*, *Time*, and *Mockito*); the median size of a subject program in DEFECTS4J is 129 592 lines of code. DEFECTS4J has become the standard benchmark to evaluate APR tools.

INTROCLASSJAVA [14] is a Java translation of the IntroClass benchmark [33], including 297 buggy student-written solutions to 6 small assignments given in an introductory undergraduate programming course. Each program in INTROCLASSJAVA comes with two JUnit test suites: a black-box one written by the instructor based on the program's specification, and a white-box one generated using a symbolic execution tool. Following the practice of other studies [33], [31], [73], [8], LIANA had access only to the black-box tests; the white-box tests feature only in the experimental evaluation, where we used them to determine which of the fixes outputted by LIANA were correct. The average size of a subject program in INTROCLASSJAVA is 230 lines of code.

QUIXBUGS [36] collects 40 faulty programs from the Quixey Challenge [37], where programmers had to fix a single-line bug in a Java implementation of a classic algorithm; the average size of a subject program in QUIXBUGS is 190 lines of code. The diverse origins and complexity of the faults from the three benchmarks help ensure that the experiments are representative of LIANA's behavior in different conditions.

4.2 Experimental Setup

The benchmarks provide, for each bug (fault) b in their collection, a buggy program P_b and tests T_b such that P_b fails on at least one of the tests in T_b—thus triggering b. Each run of LIANA targets a specific bug b: it inputs P_b and T_b and eventually produces a (possibly empty) list of fixes V_b. All fixes V_b output by LIANA are *valid*, that is they pass all tests T_b.

As customary in the APR literature [41], we determined correct fixes to a bug in DEFECTS4J or QUIXBUGS by manually going through the valid fixes in V_b and comparing each of them to the manually-written fix for the fault under repair: a valid fix is *correct* if it is *semantically equivalent*, in terms of input/output behavior, to the fix manually written by the developers and included in the benchmark. "Semantically equivalent" means that: (i) the valid inputs (i.e., those for which the program terminates normally) and the exceptional inputs (i.e., those for which the program terminates with an exception) are the same for the two programs; (ii) the two programs return the same values for the same inputs; (iii) the two programs change the object state in the same way for the same inputs. Concretely, we inspect two programs (programmer-written and automatically repaired) and look for syntactic changes that preserve input/output behavior, along the lines of the rules discussed in [41]. Conservatively, we mark as incorrect fixes that we cannot conclusively establish as equivalent in a moderate amount of time (around 15 minutes per fix).

The fix of a bug in INTROCLASSJAVA is correct if it passes all available white-box tests (which LIANA had not access to), in addition to the black-box tests that LIANA uses directly for validation. Programs and bugs in INTROCLASSJAVA are sufficiently simple that passing all white-box tests provides high confidence in their correctness.

For each experiment, we record: the number of *valid* fixes in the output; the rank of the first *correct* fix (if any is found); the overall wall-clock running time; the running time until the first *valid* fix is found; and the number of fixes validated as well as the running time until the first *correct* fix is found (if any is found).

All the experiments ran on the cloud infrastructure provided by the authors' institution. Each experiment uses exclusively one virtual machine instance running Ubuntu 14.04 and Oracle's Java JDK 1.8 on one core of an Intel Xeon Processor E5-2630-v2 with 8 GB of RAM, and runs for up to 6 hours, after which it times out and is forcefully terminated. We chose a 6-hour timeout since it is similar to the time limits used in other tools' experimental evaluation [69], [48], [35], it is compatible with the typical running times of RESTORE,[4] and is consistent with our research questions (which do not assess the usage of LIANA as a real-time tool, which would require much shorter running times).

Initial fix ranking model. As discussed in Sec. 3, each run of LIANA also needs an *initial* fix ranking model R_0 as input, which it then updates based on the intermediate outcomes of validating the candidate fixes for the program currently under repair. To bootstrap this process, we built a stripped-down version of LIANA called LIANA$^-$. LIANA$^-$ basically implements only LIANA's "backbone" workflow (blue boxes in Fig. 3): it does not use a ranking model and simply enumerates all candidate fixes in batches based on the information that comes from fault localization.

Then, an experiment with LIANA targeting a bug b in some of the benchmarks uses an initial fix ranking model R_0 built with leave-one-out cross validation [22] as follows. (i) We run LIANA$^-$ on all bugs in DEFECTS4J except b, [5] with a timeout of 15 hours per bug.[6] (ii) We collect all candidate fixes generated during these runs, and manually inspect the valid ones to identify those that are correct (equivalent to the programmer-written fix in DEFECTS4J). (iii) If a run produced at least one correct fix, we use all the candidate fixes it generated to train a fix ranking model R_0 as in Sec. 3.4; if a run didn't produce any correct fixes, we do not use its output for training; (iv) Finally, we run LIANA on b using R_0 as initial fix ranking model.

This time-consuming bootstrapping process is useful to set up a uniform, informative input for all experimental runs of LIANA. However, it would not be needed if LIANA were used regularly to fix new bugs as they are discovered during

4. The original experiments with RESTORE [80] did not use timeouts but rather bounded the number of suspicious entities to be processed. In practice, this bound corresponds to total running times that rarely exceed 6 hours per bug; LIANA can process many more suspicious entities in the same time, and hence we use a timeout rather than a bound on the entities in its experiments.

5. For simplicity, if b is from a benchmark other than DEFECTS4J, the initial ranking model is still based on all bugs in DEFECTS4J, which is the most realistic and widely-used benchmark; hence, information learned on it is also serviceable for fixing other programs.

6. This longer timeout helps generate an initial fix ranking model using LIANA$^-$: since LIANA's efficiency crucially depends on the ranking model, LIANA$^-$ needs considerably more time to explore a significant portion of the search space.

TABLE 3: Experimental results of LIANA and RESTORE in repairing faults from the 3 benchmarks. For each BENCHMARK, the table reports: the number of BUGS the benchmark includes, and the average size in LOC (lines of code) of a program in the benchmark. For each tool LIANA and RESTORE: the number # of bugs that the tool fixed with a VALID fix and the median time T2V to produce the first valid fix; the number of bugs that the tool fixed with a CORRECT fix ranked in FIRST, TOP-10, or ANY position among the output, and the median time T2C to produce the first correct fix. All times are in minutes.

BENCHMARK	BUGS	LOC	LIANA						RESTORE					
			VALID		CORRECT				VALID		CORRECT			
			#	T2V	FIRST	TOP-10	ANY	T2C	#	T2V	FIRST	TOP-10	ANY	T2C
DEFECTS4J	395	129 592	110	13.5	28	42	53	14.1	98	11.6	19	29	41	21.3
INTROCLASSJAVA	297	230	95	9.3	47	57	71	10.4	82	4.3	31	43	68	9.7
QUIXBUGS	40	190	14	5.5	8	9	10	5.5	11	8.9	4	8	9	15.1
Overall	732	–	219	9.6	83	108	134	10.4	191	6.6	54	80	118	14.9

development: in this usage scenario, one would simply use the latest fix ranking model (trained on all previous runs of LIANA) as input to the next run.

LIANA on SIMFIX. To support our claim that the LIANA technique is compatible with G&V APR techniques other than RESTORE, we developed SIMFIX+L: an extension of the SIMFIX APR tool [26] with LIANA's technique. SIMFIX is another state-of-the-art APR technique for Java that has been extensively evaluated on DEFECTS4J and whose source code and experimental artifacts are publicly available.[7]

SIMFIX generates candidate fixes in an order that depends on the suspiciousness of statements identified by its spectrum-based fault-localization algorithm. SIMFIX+L does the same but only for the first batch of candidate fixes—to bootstrap the learning process; successive iterations of fix generation follow the order given by a fix ranking model trained, as in LIANA, on the outcome of validation. More precisely, SIMFIX+L generates candidate fixes for 5 more suspicious locations, and validates the top 30% of the newly generated candidates in each iteration. By default, SIMFIX terminates as soon as it finds a valid fix; to better compare it with SIMFIX+L—which needs multiple validation runs to retrain its fix ranking model—we allowed SIMFIX to generate up to 10 valid fixes in our experiments. As in the original experiments [26], both SIMFIX and SIMFIX+L ran with a 5-hour timeout per bug.

In our experiments, we only ran SIMFIX and SIMFIX+L on DEFECTS4J bugs: some details of SIMFIX's implementation rely on DEFECTS4J scripts to compile fixes and run tests; thus, applying it to other benchmarks would require a non-trivial amount of tweaking, which is outside our evaluation's scope.

4.3 Comparison with Other Tools

We compared LIANA with all tools for Java that satisfy the following criteria, which enable a meaningful and fair comparison: (i) the tool has been evaluated on at least one of the 3 benchmarks; (ii) the tool's evaluation reports the number of *all* correct fixes produced by the tool; (iii) the tool does not rely on "perfect" fault localization information; (iv) the tool can produce correct fixes for at least 10 bugs from the benchmark when it comes to DEFECTS4J. Tools like Cardumen [53] and DeepRepair [74] are excluded because they don't fulfill (ii); this also excludes a recent experimental evaluation [13] which does not analyze correctness of fixes. Tools like CoCoNuT [48] and SketchFixPP [24] are excluded because they don't fulfill (iii). Earlier approaches such as xPAR [30] are excluded because they don't fulfill (iv)—outperforming earlier approaches would not be especially meaningful.

According to these criteria, we selected the 21 tools listed in Tab. 5. The table reports the data from each tool's main publication or replication package, which we used for our comparison without rerunning the tools. The exceptions are the rows about RESTORE, SIMFIX, and TBar. (i) RESTORE's original publication [80] only evaluated it on DEFECTS4J; since LIANA is based on RESTORE, we designed an extensive evaluation of the two tools, and hence we ran RESTORE on all of the bugs in DEFECTS4J, INTROCLASSJAVA, and QUIXBUGS bugs with a time out of 6 hours per bug. (ii) As we discuss in the next paragraph, we also reran SIMFIX and TBar on the benchmarks after modifying their stopping conditions to allow them to generate multiple valid fixes; this modification did not alter the tools' behavior, and in fact the number of fixed bugs that SIMFIX and TBar produced in our repetition (Tab. 5's columns FIRST) coincides with those in the tools' publications [69], [41].

Most evaluations only consider the first valid fix output by the tool; the exceptions are JAID, RESTORE, and HDA. In order to get a broader view of the capabilities of the tools, we also tried to assess their effectiveness when allowed to generate more than one valid fix per run. To keep this part of the evaluation manageable, we only considered tools that are above the median in terms of number of DEFECTS4J bugs fixed with a correct fix in first position: these are the tools CapGen, DLFix, Elixir, $\text{PAR}_{\text{FixMiner}}$, HERCULES, SIMFIX, SOFix, and TBar; among them, only SIMFIX and TBar have made their source code publicly available together with instructions on how to build the tool and run experiments. This allowed us to modify SIMFIX's and TBar's implementations so that they can generate up to 10 valid fixes per run; and to run both modified tools on DEFECTS4J bugs with a timeout of 6 hours per bug (the same as in our experiments with LIANA). Columns TOP-10 in Tab. 5 report the results of these experiments. Note that these modifications were limited to the tools' stopping conditions, and did not alter in any way their core program repair algorithms.

Most Java APR tools can only generate single-location fixes (also known as "single-hunk"); the exception is HER-

7. SIMFIX's characteristics make it a suitable tool to extend with LIANA's technique. However, other tools among those listed in Tab. 5 that are also publicly available—such as Avatar [38] and $\text{PAR}_{\text{FixMiner}}$ [29]—could be modified in a similar way.

CULES, which was designed specifically to generate multi-location fixes. To better understand this distinction, Tab. 5 separately reports the number of single-hunk fixes and the number of all fixes (including multi-location ones) that HERCULES generated.

4.4 Results

4.4.1 RQ1: Effectiveness and Efficiency

Table 3 summarizes the main experimental results with LIANA, and compares them with RESTORE. Overall, LIANA produced valid fixes for 219 bugs, and correct fixes for 134 bugs—thus significantly outperforming RESTORE with 15% more valid fixes and 14% more correct fixes. However, the bugs successfully fixed by LIANA are not a superset of those fixed by RESTORE: LIANA failed to produce valid fixes for 19 faults and correct fixes for 19 faults[8] where RESTORE succeeded; on the other hand, RESTORE failed to produce valid fixes for 48 faults and correct fixes for 35 faults where LIANA succeeded. While this confirms that LIANA's search strategy is generally more successful, it also indicates that RESTORE's heuristics may be more effective in certain cases. After manually inspecting these cases, we did not find a clear pattern for why RESTORE works better. It's plausible that some fix ingredients, which are available to both LIANA and RESTORE, are simply more likely to be identified as useful by RESTORE's fault localization process. Such corner cases are always possible when dealing with techniques that are ultimately best effort, with no absolute a priori guarantees of success. Combining RESTORE's and LIANA's approaches is thus a natural extension of this work that would further improve overall effectiveness in a practical way.

A correct fix is the more valuable the higher it is ranked in the output; in fact, numerous APR tool evaluations just consider the first fix in the output [50]. LIANA produced correct fixes ranked first for 83 bugs—54% more than RESTORE—and correct fixes ranked in the top 10 for 108 bugs—35% more than RESTORE. This is further evidence that LIANA's ranking model improves effectiveness not only by generating more correct fixes but also by ranking them higher.

The median time taken by LIANA to produce a valid fix is 9.6 minutes—45% slower than RESTORE's median time. In contrast, the median time taken by LIANA to produce a correct fix is 10.4 minutes, that is, 30% *faster* than RESTORE. This indicates that LIANA's search is generally more efficient to produce correct fixes. More generally, the running time performance is reasonable for a tool based on dynamic analysis.

The violin plots in Fig. 5 compare the distributions of ranks in the output as well as the time and numbers of fixes validated till the first correct fixes are found (for bugs on which LIANA or RESTORE were successful). LIANA's distribution of ranks (Fig. 5a) is widest around 1 (i.e., most correct fixes are ranked first), and quickly becomes narrow for higher values; in contrast, RESTORE still has a considerable fraction of bugs that have quite a high rank (around 100). The running times until a correct fix is found (Fig. 5b) vary in both tools; however, RESTORE's plot has more weight in the top part: since the time scale is logarithmic, this corresponds to a marked difference in practice. LIANA also tends to be more efficient in terms of number of candidate fixes that are generated and validated until a correct fix is found (Fig. 5c). Validation can be quite time consuming—since it may involve running a large number of tests—whereas fix generation is comparatively quick; therefore, the fewer candidates need to undergo validation, the more efficient the whole APR process usually is.

8. The two sets of 19 faults do not coincide.

TABLE 4: Numbers of correct fixes generated by LIANA using any of the available repair actions in any of the benchmarks DEFECTS4J, INTROCLASSJAVA, QUIXBUGS.

BENCHMARK	REPAIR ACTION α_1	α_2	α_3	α_4	α_5
DEFECTS4J	7	19	15	17	14
INTROCLASSJAVA	99	0	103	23	15
QUIXBUGS	6	0	7	2	2
OVERALL	112	19	125	42	31

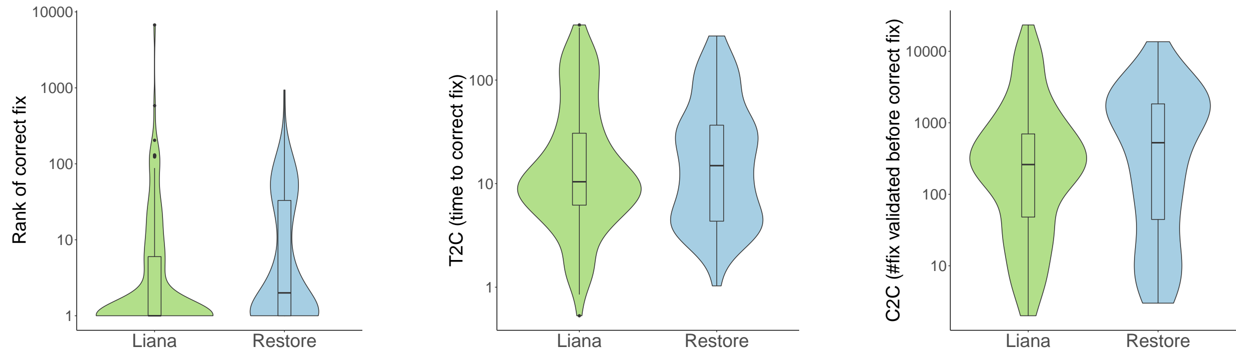
As explained in Sec. 3.1.2, LIANA uses five different kinds of repair actions—α_1 through α_5. To better understand the individual usefulness of these actions, we analyzed all correct fixes that LIANA produced and ranked in the top 10, and determined which kinds of actions they used; Tab. 4 shows the analysis results. Overall, all actions are necessary in some cases. If we only consider DEFECTS4J bugs, the distribution of used actions is fairly balanced; DEFECTS4J is a diverse and heterogeneous collection of real-world bugs, and hence it is not surprising that the more kinds of fixes we can generate the more bugs we can successfully patch. In contrast, the distribution of used actions is more skewed for the other two benchmarks. In INTROCLASSJAVA, in particular, actions α_1 and α_3 are disproportionately successful, whereas α_2 is never needed. Remember that INTROCLASSJAVA consists of simple programs written by students; apparently, forgetting an update statement (supplied by action α_1) or using an incorrect condition (corrected by action α_3) are the most common mistakes in these programs.

LIANA *produced valid fixes for 219 bugs and correct fixes for 134 bugs. For 83 bugs, the correct fix is the first in the output. The median running time to build a correct fix is 10.4 minutes.* LIANA *consistently outperforms* RESTORE.

4.4.2 RQ2: Comparison

Tab. 5 compares LIANA with 21 recent APR tools for Java on bugs from the 3 benchmarks in terms of number of valid and correct fixes, and the corresponding measures of precision (number of bugs with correct fix / number of bugs with valid fix) and recall (number of bugs with correct fix / number of bugs in the benchmarks).

Comparison results: DEFECTS4J. When only the first valid fix produced by a tool is considered, which is how most tools are evaluated, LIANA produced more correct fixes for DEFECTS4J bugs than all the other tools except DLFix (2 more fixes), SIMFIX (5 more fixes) and HERCULES (18 more fixes, only if we consider multi-location fixes). In contrast to recall, LIANA's precision is somewhat lackluster—even though it clearly improves over RESTORE's precision, which



(a) Distribution of the position (rank) of the first correct fix in the output.

(b) Distribution of T2C: time until the first correct fix is built.

(c) Distribution of C2C: number of fixes validated until the first correct fix is found.

Fig. 5: Violin plots comparing the distributions of different measures in LIANA vs. RESTORE for all bugs that the tools can correctly fix. Vertical axes use logarithmic scales.

TABLE 5: A comparison of LIANA with 21 other APR tools for Java in fixing bugs from different BENCHmarks. For each tool, the table reports the number of bugs for which the tool produced VALID fixes, and CORRECT fixes in FIRST, TOP-10, or ANY position in the output; it also shows the corresponding PRECISION and RECALL with respect to the benchmark's bugs. Finally, column UNIQUE lists the number of bugs that each tool can correctly fix that no other tools in the table can (this data is available only for DEFECTS4J). Question marks indicate data that was not available in the tool's cited publication or replication package.

BENCH	TOOL	VALID	FIRST			TOP-10			ANY			UNIQUE
			CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	
DEFECTS4J	LIANA	110	28	25%	7%	42	38%	11%	53	48%	13%	3
	ACS [79]	23	18	78%	5%	?	?	?	?	?	?	4
	ARJA [84]	59	18	31%	5%	?	?	?	?	?	?	3
	Avatar [41]	57	19	33%	5%	?	?	?	?	?	?	2
	CapGen [73]	25	21	84%	5%	?	?	?	?	?	?	0
	DLFix [35]	65	30	46%	8%	?	?	?	?	?	?	0
	Elixir [67]	41	26	63%	7%	?	?	?	?	?	?	?
	$\text{PAR}_{\text{FixMiner}}$ [29]	32	26	81%	7%	?	?	?	?	?	?	?
	HDA [30]	?	13	?	4%	23	?	6%	?	?	?	1
	HERCULES (single hunk) [68]	43	26	60%	7%	?	?	?	?	?	?	1
	HERCULES (all) [68]	63	46	73%	12%	?	?	?	?	?	?	8
	LSRepair [40]	38	19	50%	5%	?	?	?	?	?	?	10
	JAID [8]	94	14	15%	4%	25	27%	6%	35	37%	9%	0
	RESTORE	98	19	20%	5%	29	30%	7%	41	42%	10%	3
	SIMFIX [69]	55	33	60%	8%	34	62%	9%	?	?	?	7
	SketchFix [24]	26	19	73%	5%	?	?	?	?	?	?	0
	SOFix [43]	?	23	?	6%	?	?	?	?	?	?	1
	ssFix [77]	60	20	33%	6%	?	?	?	?	?	?	0
	TBar [41]	72	24	33%	6%	28	39%	7%	?	?	?	1
INTROCLASSJAVA	LIANA	95	47	49%	16%	57	60%	19%	71	75%	24%	13
	JAID [8]	84	30	36%	10%	43	51%	14%	69	82%	23%	1
	CapGen [73]	?	25	?	8%	?	?	?	?	?	0	0
	JFix [31]	?	19	?	6%	?	?	?	?	?	?	?
	RESTORE	82	31	38%	10%	43	52%	14%	68	83%	23%	0
	S3 [32]	?	22	?	7%	?	?	?	?	?	?	?
QUIXBUGS	LIANA	14	8	57%	20%	9	64%	23%	10	71%	25%	3
	JAID [8]	11	4	36%	10%	9	82%	23%	9	82%	23%	0
	Astor [52]	11	6	55%	15%	?	?	?	?	?	?	?
	Nopol [82]	4	1	25%	3%	?	?	?	?	?	?	?
	RESTORE	11	4	36%	10%	8	73%	20%	9	82%	23%	0

is the most direct comparison yardstick. All tools that outperform LIANA in terms of precision incorporate knowledge learned from human-written fixes mined from code repositories. This suggests a similar approach to improve LIANA's precision, which we plan to investigate in future work.

If we consider up to 10 valid fixes per bug (for the tools for which we could collect this information, under columns TOP-10 in Tab. 5), SIMFIX built correct fixes for 34 bugs (1 fix ranked below first); TBar for 28 bugs (4 ranked below first). LIANA outperforms both of them in terms of recall, as it fixes 42 DEFECTS4J bugs with a fix that is ranked in the top 10 (14 fixed ranked below first); in contrast, SIMFIX and TBar still outperform LIANA in terms of precision. LIANA finds even more correct fixes (for 53 DEFECTS4J bugs) if we consider all valid fixes it can generate during a run, regardless of their position in the output. It is clear that SIMFIX and TBar were

designed with a focus on generating the first valid fix; they do a very good job at that, and hence their success does not increase much if they are allowed to continue past that. On the other hand, there may be value in generating more valid fixes for the same bug—especially given that the fixes produced by APR tools are typically succinct; hence, analyzing more than one suggestion requires moderate effort.

Comparison results: other benchmarks. For benchmarks INTROCLASSJAVA and QUIXBUGS, LIANA consistently produced correct fixes for more faults than all other tools evaluated on these benchmarks. In particular, LIANA correctly fixed 47 INTROCLASSJAVA bugs ranked in first position, 16 more than the runner-up RESTORE; if we disregard the rank and consider all valid fixes produced by the tools, LIANA still fixes more bugs correctly but its advantage over the other tools becomes quite narrow (LIANA's 71 correctly fixed bugs vs. JAID's 69 and RESTORE's 68). This illustrates once again the relation between fix space and search heuristics, which was also apparent in Sec. 2's example: not only does LIANA offer a larger fix space; crucially, its heuristics can search it effectively. On the INTROCLASSJAVA and QUIXBUGS benchmarks, LIANA is also the most precise tool—at least among those that provide data about their precision. Together, these results confirm that LIANA manages to improve both precision and recall by combining an extended fix space with a ranking model that tends to perform well on it in practice.

> LIANA *is among the most effective APR tools for Java in terms of number of correct fixes it produces: 7% recall on* DEFECTS4J *bugs, 16% recall on* INTROCLASSJAVA *bugs, and 20% recall on* QUIXBUGS *bugs.* LIANA*'s precision is not as competitive on* DEFECTS4J *bugs (25% precision), whereas it outperforms other tools on* INTROCLASSJAVA *(49% precision) and* QUIXBUGS *(57% precision) bugs.*

4.4.3 RQ3: Feature Usefulness

To empirically assess the usefulness of each individual feature used by LIANA's fix ranking model, we perform a so-called ablation study. Tab. 6 shows how LIANA's effectiveness changes if we remove any of the 17 features described in Sec. 3.2. More precisely, in each experiment: (i) we remove one feature from the ranking model; (ii) we run LIANA with this modified ranking model on all bugs in DEFECTS4J, with the same experimental protocol as for the main experiments; (iii) we collect LIANA's output, and determine how many correct fixes it produced and in which position in the output.

The most common outcome of removing a feature is that LIANA's effectiveness worsens. For very specific scenarios it may slightly improve: for example, LIANA can rank two more correct fixes as first in the output if we remove features 4, 7, or 14; in these cases, however, it's only the rank of a few fixes that improves, whereas LIANA finds fewer correct fixes overall. On the other hand, the differences caused by removing any one feature are small in absolute value; thus, the ranking model is relatively robust and no feature has a uniquely crucial role.

> *All features of* LIANA*'s ranking model contribute somewhat to* LIANA*'s overall bug-fixing effectiveness.*

Another component of LIANA's ranking model that needs an empirical validation are the scores (labels) that

TABLE 6: Impact of features used by LIANA's fix ranking model. Each column labeled k reports the number of bugs in DEFECTS4J that LIANA could fix with a correct fix using a fix ranking model *without* feature φ_k. Different rows consider correct fixes in the FIRST position, in a TOP-10 position, or in ANY position in the output.

	OMITTED FEATURE φ																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
FIRST	20	29	27	30	24	24	30	22	27	27	27	26	21	30	26	24	26
TOP-10	37	40	39	43	43	39	44	37	39	40	37	39	40	42	39	39	37
ANY	51	51	51	52	53	50	52	50	52	53	52	50	51	54	52	51	49

TABLE 7: A comparison of SIMFIX and SIMFIX+L in repairing bugs from DEFECTS4J. For each tool, the table reports the number of bugs for which the tool produced a Valid fix, and a Correct fix in FIRST or TOP-10 position in the output.

TOOL	V	C (FIRST)	C (TOP-10)
SIMFIX	56	33	34
SIMFIX+L	65	35	36

we give to new datapoints, corresponding to whether a fix is progress, valid, or correct (Sec. 3.4). We took all 53 DEFECTS4J bugs that LIANA can correctly fix, and trained a model using slightly different scores: in addition to the original $(1, 3, 5, 10)$, we also tried $(2, 6, 10, 20)$, $(1, 3, 5, 14)$, and $(2, 6, 10, 20)$. Changing scores did change the ranking of candidate fixes in a few cases; however, the changes are unlikely to affect LIANA's overall effectiveness (which bugs it can successfully fix). We also tried the much larger scores $(10, 30, 50, 100)$; this drastic change degraded LIANA's behavior, as training its ranking model sometimes failed. Indeed, we found out that the developers of XGBoost recommend[9] using scores in the range 0–10; using much larger numbers may introduce numerical overflows, which makes XGBoost's implementation misbehave. In all, there is some leeway in choosing different ranking scores, provided they remain within a small range to avoid numerical errors.

4.4.4 RQ4: Generality

Tab. 7 shows that adding LIANA's technique on top of SIMFIX improves its effectiveness only slightly: more precisely, it increases recall (more correct fixes) but decreases precision. In contrast, LIANA brought clear improvements to performance. Fig. 6 shows a scatterplot comparing SIMFIX and SIMFIX+L in terms of the time it took them to produce the first correct fix for each DEFECTS4J bug. The regression line has a slope of about 0.36, which means that SIMFIX+L was $2.7 = 1/0.36$ times faster on average.

LIANA could have a stronger impact on SIMFIX if we searched a larger (or different) fix space rather than just searching the same space with different heuristics: in our experiments, SIMFIX+L used the same "catalog" of fixes as SIMFIX (which was generated by mining programmer-written repairs, and we took from SIMFIX's replication package without changes). SIMFIX is quite effective at searching this fix space, but several bugs have fixes that fall outside it; for example, SIMFIX couldn't fix DEFECTS4J's bugs *Lang45*

9. https://discuss.xgboost.ai/t/very-large-ndcg-result/1712/4

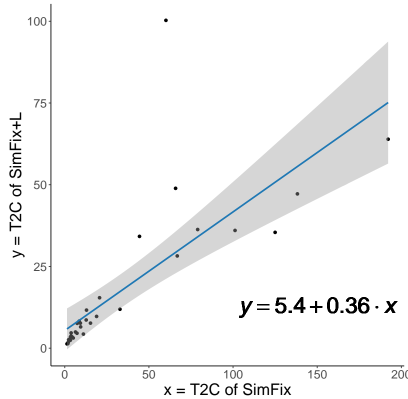


Fig. 6: Performance comparison of SIMFIX and SIMFIX+L. A dot with coordinates (x, y) corresponds to a DEFECTS4J bug that both tools could fix: SIMFIX produced a correct fix in x minutes, whereas SIMFIX+L produced it in y minutes. The regression line indicates that SIMFIX+L is faster on average.

and *Math80* (both of which LIANA can correctly fix) even though it exhaustively searched all the fix space. If it could target a larger fix space, SIMFIX+L would automatically adjust the search strategy to this larger space based on validation results. This is an interesting direction for future work.

Applying LIANA's technique to SIMFIX does not noticeably affect its effectiveness (2 more correct fixes are generated), but it clearly improves its efficiency (a 2.7x speedup on average).

4.5 Threats to Validity

In this section, we outline possible threats to the validity of our experimental findings, and discuss how we mitigated them.

Construct validity concerns whether the measures used in the experiments capture salient features. We classify a fix as *correct* if it is equivalent to a given programmer-written fix. Since we assess equivalence manually, others may disagree with some of our assessments. To mitigate this threat, we were conservative in evaluating equivalence: if a fix does not clearly introduce the same behavior as the reference fix, we classify it as incorrect. In addition, one author reviewed the classification made by another. This approach is consistent with how other researchers in this area assess correctness.

Internal validity concerns whether the experiments controlled for possible confounders. One obvious threat comes from possible mistakes in our implementation of LIANA. To mitigate this threat, we reviewed each other's code and experimental scripts to ensure their correctness before conducting the final experiments. Another threat has to do with the initial fix ranking models used in the experiments (Sec. 4.2). It is possible that training the initial ranking model with different or more data may lead to different results. We applied a form of leave-one-out validation, which is a standard approach and ensures that the initial ranking model is not cherry-picked. In the future, we may carry out more experiments to further assess LIANA's sensitivity to the initial fix ranking model.

External validity concerns whether the experimental findings generalize to other contexts. We mitigated these threats by conducting a large-scale experimental evaluation involving 3 different benchmarks. We also implemented LIANA's technique on top of two different G&V APR tools (RESTORE and SIMFIX), so as to ascertain that the technique is successfully applicable to more than one existing approach.

5 RELATED WORK

Following the influential work of GenProg [72], many APR techniques that target general classes of faults are *test-driven*: they use tests as examples of correct (passing tests) and incorrect (failing tests) behavior. Since tests cannot completely capture program correctness, a fix that passes all available tests may still be incorrect. This gap between passing all tests and being actually correct is referred to as the *overfitting* problem in APR [63], [59], [75]. A lot of recent work in this area tried to better characterize the problem [64], [55], [12] and to limit its practical negative impact [17], [78], [83], [75].

According to how they search through the space of all possible fixes, test-driven APR techniques can be classified as constraint-based or heuristic [20], [13]. *Constraint-based* techniques [57], [81], [58], [56] replace suspicious expressions in the faulty program with symbolic variables, build constraints on those variables, and then solve the constraints to find concrete fixes that achieve that program behavior by construction. Since a constraint must encode program behavior in purely logic form, practical challenges faced by constraint-based techniques include handling expressions with side effects and generating patches that introduce complete statements.

The majority of test-driven APR techniques are *heuristic* [3], [72], [28], [63], [59], [45], [30], [47], [77], [44], [7], [79], [73], [24], [19], [26], [68], [80]: they follow a process that starts with fault localization, continues with fix generation, and concludes with fix validation. In contrast to constraint-based techniques, fix validation is necessary in heuristic APR techniques, since fix generation is driven by heuristics and provides no guarantee that the generated candidate fixes are adequate. The space of all possible fixes is typically huge, whereas the correct fixes are rather sparse in that space [46]—a "needle in a haystack" problem. Thus, identifying information that is useful to effectively search this space is a crucial challenge of heuristic APR. In the rest of this section we outline various such sources of information.

Program context. It is common that the "ingredients" to build a correct fix can be found around the failure location [11], [54]. For example, GenProg [72] is based on the assumption that defects can be repaired by taking code elements from other locations in the program under repair. ssFix [77] matches contextual information at the fixing location to a database of human-written fixes, and uses the matching code elements to drive fix generation. SimFix [26] combines the information extracted from existing patches and snippets similar to the code under fix to make the search for correct fixes more efficient. CapGen [73] prioritizes ingredients based on a notion of programming context. Hercules [68] can build multi-hunk fixes by first identifying a set of repair locations with similar code—which are therefore expected to need similar changes—and then generating fixes that modify all those locations consistently.

Learning from correct fixes. Another popular approach is summarizing the characteristics of correct fixes, and then generating fixes automatically that have similar characteristics. PAR [28] is based on ten manually-generated patterns, which reflect common characteristics of human-written fixes. This approach is inflexible as it is based on a fixed hardcoded set of patterns [59]. More recent approaches are instead completely automated (called "learning-aided repair" in [20]), and hence are applicable to different programs with consistent performance. SPR [45] is a technique that systematically generates candidate fixes according to a set of predefined transformation functions. Prophet [47] adds on top of SPR a probabilistic model learned from human-written fixes, which is effective in prioritizing the generated candidate fixes. Genesis [44] infers code transformations with template variables from human-written patches. Elixir [67] specializes in repairing buggy method invocations; to this end, it machine learns a model that captures the characteristics of correct fixes of this kind. TBar [39] systematically applies fix patterns summarized from the literature to generate fixes. FixMiner [29] extracts patterns of correct fixes based on their abstract syntax trees.

Dynamic analysis. Some search-based APR techniques reuse some of the information that comes from dynamic analysis (i.e., from the *validation* phase) to guide the fix generation phase. In approaches based on genetic algorithms [3], [2], [72], the fitness function typically includes information that comes from validation: the more tests a candidate fix passes, the more "fit" it is. The history-driven approach [30] is based on a stochastic search process that views candidates fix candidates as mutants, and selects those mutants that match the characteristics learned from fix history.

LIANA is based on the design idea of combining different sources of information about the characteristics of suitable fixes into a single model, so that complementary information can inform the generation of new fixes. LIANA's model includes measures of code similarity, it can learn from correct (or valid) fixes, and it is updated based on the outcome of validation.

Deep learning techniques have been successfully applied to countless application domains in recent year—including APR. DeepRepair [74] uses deep learning models to capture the ingredients of programmer-written fixes without explicitly selected features. DeepFix [21] predicts locations responsible for compilation errors in C programs along with suitable patches. Several applications of deep learning to APR [23], [70], [10], [6], [48], [35] treat program repair as a machine translation task that converts buggy code to correct code. The techniques differ in the kinds of deep neural network architecture they use, how they process the input data, and how they encode programs as text. As we discussed in the rest of the paper, a key difference with LIANA is that all these techniques train models offline, whereas LIANA continues the learning phase while the program repair is ongoing.

Benchmarks such as DEFECTS4J [27], INTROCLASSJAVA [14], QUIXBUGS [36], and Bears [49] have been instrumental in enabling quantitative comparisons of APR tools, and in motivating the development of new capabilities. As we mentioned in Sec. 4, DEFECTS4J has become the de facto standard to evaluate APR for Java [13]; using additional benchmarks helps avoid overfitting it, and can identify specialized techniques that are especially effective in certain domains.

6 CONCLUSIONS AND FUTURE WORK

We presented the LIANA technique for APR. LIANA incorporates the essential information about the quality of fixes in a fix ranking model, which it updates online during its runs based on the outcome of validation. LIANA can build correct fixes for 134 bugs in 3 popular benchmarks, placing it among the most competitive G&V APR tools for Java.

LIANA improves over RESTORE in its overall effectiveness in fixing bugs; nevertheless, there remain a few cases (19 faults in our experiments) where RESTORE is successful whereas LIANA is not. More generally, it is often the case that each program repair tool has unique capabilities of fixing certain kinds of bugs (column UNIQUE in Tab. 5), whereas it is less effective on others. These observations suggest that *combining* the ingredients of different program repair techniques is a natural direction for future work.

By implementing the LIANA technique on top of SIMFIX (Sec. 4.4.4), we demonstrated that it can be usefully deployed on different APR systems. In our experiments, however, we did not modify SIMFIX's search space; as a result, even though LIANA considerably sped up the search in that space, it found only two more correct fixes. In future work, we would like to extend this experiment: after applying LIANA to another APR system, and ascertaining that it helps speed up its search for correct fixes, we will also extend the system's fix space to determine if more correct fixes can be found within the same time budget.

As we mention in Sec. 4.2, SIMFIX is not the only recently-developed open-source APR tool for Java that could be extended with the LIANA technique. On the other hand, the experience with SIMFIX suggests that implementing LIANA's ranking model on top of an existing tool would have a greater effectiveness if we could tailor LIANA's techniques to the characteristics of the tool (for example, to its actual fix space). As future work, we could investigate the implementations of other open-source Java APR tools, such as Avatar [38] and $\text{PAR}_{\text{FixMiner}}$ [29], looking for opportunities of applying LIANA's techniques in the most effective way.

LIANA's fix ranking model uses 17 features that mainly capture "syntactic distance" from different angles (Tab. 1). These features are fairly common measures used in program analysis; as we demonstrated in Sec. 4.4.3, all features contribute to LIANA's effectiveness, even though omitting any one feature has at most a modest negative impact on effectiveness. Nevertheless, any choice of features is heuristic and may overlook characteristics that become relevant in certain contexts. This suggests to experiment with deep learning models as fix ranking models. The appeal of these models is their capability of identifying features in the training data that are useful to improve the accuracy of classification. In future work, we plan to look into these machine learning techniques.

Acknowledgments

Yu Pei was supported in part by the Hong Kong RGC General Research Fund (GRF) under grant PolyU 152002/18E.

Minxue Pan was supported in part by the National Natural Science Foundation of China (No. 61972193). The research project of Qixin Wang related to this paper was supported in part by the Hong Kong Research Grants Council (RGC) Theme-based Research Scheme T22-505/19-N (P0031331, RBCR, P0031259, RBCP), in part by RGC GRF under Grant PolyU 152002/18E (P0005550, Q67V) and Grant PolyU 152164/14E (P0004750, Q44B), in part by RGC Germany/HK Joint Research Scheme under Grant G-PolyU503/16, and in part by the Hong Kong Polytechnic University fund under Grant P0033695 (ZVRD), Grant P0013879 (BBWH), Grant P0031950 (ZE1N), Grant P0036469 (CDA8), and Grant 8B2V. Carlo A. Furia was supported in part by the Swiss National Science Foundation (SNF) under grant *Hi-Fi 200021-182060*.

REFERENCES

[1] A. Agrawal and T. Menzies. Is "better data" better than "better data miners"?: on the benefits of tuning SMOTE for defect prediction. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1050–1061. ACM, 2018.

[2] A. Arcuri. Evolutionary Repair of Faulty Software. *Appl. Soft Comput.*, 11(4):3494–3514, June 2011.

[3] A. Arcuri and X. Yao. A Novel Co-evolutionary Approach to Automatic Software Bug Fixing. In *In Proceedings of the IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[4] T.-D. B. Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188, New York, NY, USA, 2016.

[5] J. Chakraborty, S. Majumder, and T. Menzies. Bias in machine learning software: why? how? what to do? In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 429–440. ACM, 2021.

[6] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. Codit: Code editing with tree-based neural models. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[7] L. Chen, Y. Pei, and C. A. Furia. Contract-Based Program Repair without the Contracts. In *Proceedings of the 2017 32th IEEE/ACM International Conference on Automated Software Engineering*, pages 637–647, Urbana-Champaign, IL, USA, 2017.

[8] L. Chen, Y. Pei, and C. A. Furia. Contract-based program repair without the contracts: An extended study. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.

[9] T. Chen and C. Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794, New York, NY, USA, 2016. Association for Computing Machinery.

[10] 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. *IEEE Transactions on Software Engineering*, pages 1–1, 2019.

[11] Z. Chen and M. Monperrus. The remarkable role of similarity in redundancy-based program repair. Technical Report 1811.05703, arXiv, 2018.

[12] X. B. Dinh Le, L. Bao, D. Lo, X. Xia, S. Li, and C. Pasareanu. On reliability of patch correctness assessment. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 524–535, May 2019.

[13] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu. Empirical review of Java program repair tools: A large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2019, page 302–313, New York, NY, USA, 2019. Association for Computing Machinery.

[14] T. Durieux and M. Monperrus. IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs. Research Report hal-01272126, Universite Lille 1, 2016.

[15] W. Eric Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *Journal of Systems and Software*, 83(2):188–208, Feb. 2010.

[16] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering*, ASE '14, page 313–324, New York, NY, USA, 2014. Association for Computing Machinery.

[17] X. Gao, S. Mechtaev, and A. Roychoudhury. Crash-avoiding program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 8–18, New York, NY, USA, 2019. ACM.

[18] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Transactions on Software Engineering*, 45(1):34–67, Jan 2019.

[19] A. Ghanbari, S. Benton, and L. Zhang. Practical program repair via bytecode mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 19–30, New York, NY, USA, 2019. ACM.

[20] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, nov 2019.

[21] R. Gupta, S. Pal, A. Kanade, and S. Shevade. DeepFix: Fixing common C language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, AAAI'17, page 1345–1351. AAAI Press, 2017.

[22] J. Han and M. Kamber. *Data mining : concepts and techniques*. Kaufmann, San Francisco [u.a.], 2005.

[23] H. Hata, E. Shihab, and G. Neubig. Learning to generate corrective patches using neural machine translation, 2019.

[24] J. Hua, M. Zhang, K. Wang, and S. Khurshid. Towards practical program repair with on-demand candidate generation. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 12–23, 2018.

[25] K. Järvelin and J. Kekäläinen. Cumulated gain-based evaluation of ir techniques. *ACM Trans. Inf. Syst.*, 20(4):422–446, oct 2002.

[26] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309, 2018.

[27] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440, New York, NY, USA, 2014.

[28] D. Kim, J. Nam, J. Song, and S. Kim. Automatic Patch Generation Learned from Human-written Patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811, Piscataway, NJ, USA, 2013.

[29] A. Koyuncu, K. Liu, T. F. Bissyandé, D. Kim, M. Monperrus, J. Klein, and Y. Le Traon. FixMiner: Mining relevant fix patterns for automated program repair. *Empirical Software Engineering*, 25(3):1980–2024, 2020.

[30] X. B. D. Le, D. Lo, and C. L. Goues. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 1, pages 213–224, 2016.

[31] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. JFIX: semantics-based repair of Java programs via symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017*, pages 376–379, 2017.

[32] X. D. Le, D. Chu, D. Lo, C. Le Goues, and W. Visser. S3: syntax- and semantic-guided repair synthesis via programming by examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 593–604, 2017.

[33] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. T. Devanbu, S. Forrest, and W. Weimer. The ManyBugs and IntroClass benchmarks for automated repair of C programs. *IEEE Trans. Software Eng.*, 41(12):1236–1256, 2015.

[34] X. Li and L. Zhang. Transforming programs and tests in tandem for fault localization. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):92:1–92:30, Oct. 2017.

[35] Y. Li, S. Wang, and T. N. Nguyen. Dlfix: Context-based code transformation learning for automated program repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*,

ICSE '20, page 602–614, New York, NY, USA, 2020. Association for Computing Machinery.

[36] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. Quixbugs: A multi-lingual program repair benchmark set based on the quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56, New York, NY, USA, 2017.

[37] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama. QuixBugs: a multi-lingual program repair benchmark set based on the Quixey challenge. In *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, pages 55–56. ACM, 2017.

[38] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. AVATAR: fixing semantic bugs with fix patterns of static analysis violations. In X. Wang, D. Lo, and E. Shihab, editors, *26th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2019, Hangzhou, China, February 24-27, 2019*, pages 456–467. IEEE, 2019.

[39] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. Tbar: Revisiting template-based automated program repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2019, pages 31–42, New York, NY, USA, 2019. ACM.

[40] K. Liu, A. Koyuncu, K. Kim, D. Kim, and T. F. Bissyandé. Lsrepair: Live search of fix ingredients for automated program repair. In *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, pages 658–662, 2018.

[41] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyandé, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon. On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for Java programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 615–627, New York, NY, USA, 2020. Association for Computing Machinery.

[42] T.-Y. Liu. Learning to rank for information retrieval. *Found. Trends Inf. Retr.*, 3(3):225–331, Mar. 2009.

[43] X. Liu and H. Zhong. Mining stackoverflow for program repair. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 118–129, 2018.

[44] F. Long, P. Amidon, and M. Rinard. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 727–739, New York, NY, USA, 2017.

[45] F. Long and M. Rinard. Staged Program Repair with Condition Synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 166–178, New York, NY, USA, 2015.

[46] F. Long and M. Rinard. An analysis of the search spaces for generate and validate patch generation systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering*, pages 702–713, 2016.

[47] F. Long and M. Rinard. Automatic patch generation by learning correct code. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312. ACM, 2016.

[48] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 101–114, New York, NY, USA, 2020. Association for Computing Machinery.

[49] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. BEARS: An extensible Java bug benchmark for automatic program repair studies. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 468–478, Feb 2019.

[50] 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. *Empirical Software Engineering*, 2016.

[51] M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, Nov. 2013.

[52] M. Martinez and M. Monperrus. ASTOR: A program repair library for Java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis (ISSTA) – Demonstrations Track*, pages 441–444. ACM, 2016.

[53] M. Martinez and M. Monperrus. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In T. E. Colanzi and P. McMinn, editors, *Search-Based Software Engineering - 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings*, volume 11036 of *Lecture Notes in Computer Science*, pages 65–86. Springer, 2018.

[54] M. Martinez, W. Weimer, and M. Monperrus. Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches. In *36th IEEE International Conference on Software Engineering - New Ideas and Emerging Results Track*, 2014.

[55] S. Mechtaev, X. Gao, S. H. Tan, and A. Roychoudhury. Test-equivalence analysis for automatic patch generation. *ACM Trans. Softw. Eng. Methodol.*, 27(4):15:1–15:37, Oct. 2018.

[56] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 129–139, New York, NY, USA, 2018. ACM.

[57] S. Mechtaev, J. Yi, and A. Roychoudhury. DirectFix: Looking for Simple Program Repairs. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, pages 448–458, 2015.

[58] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 691–701, New York, NY, USA, 2016.

[59] M. Monperrus. 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 International Conference on Software Engineering*, pages 234–242, New York, NY, USA, 2014.

[60] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, Jan. 2018.

[61] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[62] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620, Piscataway, NJ, USA, 2017.

[63] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering*, 40(5):427–449, 2014.

[64] Z. Qi, F. Long, S. Achour, and M. Rinard. An Analysis of Patch Plausibility and Correctness for Generate-and-validate Patch Generation Systems. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 24–36, New York, NY, USA, 2015.

[65] K. Ramasubramanian and J. Moolayil. *Applied Supervised Learning with R: Use machine learning libraries of R to build models that solve business problems and predict future trends*. Packt Publishing, 2019.

[66] RESTORE: implementation and replication package. http://tiny.cc/9xff3y.

[67] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad. Elixir: Effective object oriented program repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, pages 648–659, Piscataway, NJ, USA, 2017.

[68] S. Saha, R. K. Saha, and M. R. Prasad. Harnessing evolution for multi-hunk program repair. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 13–24, Piscataway, NJ, USA, 2019. IEEE Press.

[69] Simfix. https://github.com/xgdsmileboy/SimFix/. Accessed: 2021-04-15.

[70] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk. An empirical study on learning bug-fixing patches in the wild via neural machine translation. *ACM Trans. Softw. Eng. Methodol.*, 28(4), Sept. 2019.

[71] WALA: T.J. Watson libraries for analysis. http://wala.sf.net.

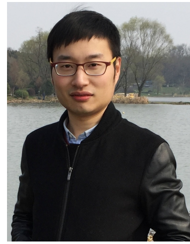
[72] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest. Automatically finding patches using genetic programming. In *Proceedings of the IEEE 31st International Conference on Software Engineering*, pages 364–374, 2009.

[73] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung. Context-aware patch generation for better automated program repair. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, pages 1–11, 2018.

[74] M. White, M. Tufano, M. Martínez, M. Monperrus, and D. Poshyvanyk. Sorting and transforming program repair ingredients via deep learning code similarities. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 479–490, 2019.
[75] Q. Xin and S. P. Reiss. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 226–236, New York, NY, USA, 2017.
[76] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 660–670. IEEE, 2017.
[77] Q. Xin and S. P. Reiss. Leveraging syntax-related code for automated program repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, pages 660–670, Piscataway, NJ, USA, 2017.
[78] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, pages 789–799, New York, NY, USA, 2018. ACM.
[79] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang. Precise condition synthesis for program repair. In *Proceedings of the 39th International Conference on Software Engineering*, pages 416–426, Piscataway, NJ, USA, 2017.
[80] T. Xu, L. Chen, Y. Pei, T. Zhang, M. Pan, and C. A. Furia. Restore: Retrospective fault localization enhancing automated program repair. *IEEE Transactions on Software Engineering*, pages 1–1, 2020.
[81] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering*, PP(99):1–1, 2016.
[82] H. Ye, M. Martinez, T. Durieux, and M. Monperrus. A comprehensive study of automatic program repair on the QuixBugs benchmark. https://arxiv.org/abs/1805.03454, 2019.
[83] 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. *Empirical Software Engineering*, 24(1):33–67, Feb 2019.
[84] Y. Yuan and W. Banzhaf. Arja: Automated repair of Java programs via multi-objective genetic programming. *IEEE Transactions on Software Engineering*, 46(10):1040–1067, 2020.

Liushan Chen is an R&D engineer with ByteDance Ltd. She received her B.A. degree from University of International Business and Economics, China, and her M.Sc. and Ph.D. degrees from the Department of Computing, The HongKong Polytechnic University. Her main research interests lie in automatic program repair, software fault localization, and automated testing for mobile apps.

Yu Pei is an assistant professor with the Department of Computing, The Hong Kong Polytechnic University, China. His main research interests include automated program repair, software fault localization, and automated software testing.



Minxue Pan is an associate professor with the State Key Laboratory for Novel Software Technology and the Software Institute of Nanjing University. He received his B.S. and Ph.D. degrees in computer science and technology from Nanjing University. His research interests include software modelling and verification, software analysis and testing, cyber-physical systems, mobile computing, and intelligent software engineering.

Tian Zhang is a professor with the Nanjing University. He received his Ph.D. degree in Nanjing University. His research interests include model driven aspects of software engineering, with the aim of facilitating the rapid and reliable development and maintenance of both large and small software systems.

Qixin Wang received the BE and ME degrees from the Department of Computer Science and Technology, Tsinghua University, Beijing, China, in 1999 and 2001, respectively, and the PhD degree from the Department of Computer Science of the University of Illinois at Urbana-Champaign in 2008. He joined the Department of Computing of the Hong Kong Polytechnic University in 2009 as an assistant professor, and is currently an associate professor. He has published about 20 first/lead author refereed papers in various top journals and conference proceedings, and over 50 papers/articles in various academic venues. He has won the IEEE Transactions on Industrial Informatics Best Paper Award in 2008, and has one paper chosen as the featured article by the IEEE Transactions on Mobile Computing 2008 May issue. His main research interests include cyber-physical systems, real-time and embedded systems, and computer networks. He is a member of the IEEE and the ACM.

Carlo A. Furia is an associate professor in the Software Institute part of the Faculty of Informatics of USI Università della Svizzera italiana.