RESTORE: Retrospective Fault Localization Enhancing Automated Program Repair

Tongtong Xu, Liushan Chen, Yu Pei, Tian Zhang, Minxue Pan, and Carlo A. Furia

Abstract—Fault localization is a crucial step of automated program repair, because accurately identifying program locations that are most closely implicated with a fault greatly affects the effectiveness of the patching process. An ideal fault localization technique would provide precise information while requiring moderate computational resources—to best support an efficient search for correct fixes. In contrast, most automated program repair tools use standard fault localization techniques—which are not tightly integrated with the overall program repair process, and hence deliver only subpar efficiency. In this paper, we present *retrospective fault localization*: a novel fault localization technique geared to the requirements of automated program repair. A key idea of retrospective fault localization is to reuse the outcome of failed patch validation to support mutation-based dynamic analysis—providing accurate fault localization information without incurring onerous computational costs. We implemented retrospective fault localization in a tool called RESTORE—based on the JAID Java program repair system. Experiments involving faults from the DEFECTS4J standard benchmark indicate that retrospective fault localization can boost automated program repair: RESTORE efficiently explores a large fix space, delivering state-of-the-art effectiveness (41 DEFECTS4J bugs correctly fixed, 8 of which no other automated repair tool for Java can fix) while simultaneously boosting performance (speedup over 3 compared to JAID). Retrospective fault localization is applicable to any automated program repair techniques that rely on fault localization and dynamic validation of patches.

1 INTRODUCTION

AUTOMATED program repair has the potential to transform programming practice: by automatically building fixes for bugs in real-world programs, it can help curb the large amount of resources—in time and effort—that programmers devote to debugging [1]. While the first viable techniques tended to produce patches that overfit the few tests typically available for validation [2], [3], automated program repair tools have more recently improved precision (see Section 5.2 for a review) to the point where they can often produce genuinely correct fixes—equivalent to those a programmer would write.

A crucial ingredient of most repair techniques—and especially of so-called *generate-and-validate* approaches [4]—is *fault localization*. Imitating the debugging process followed by human programmers, fault localization aims to identify program locations that are implicated with a fault and where a patch should be applied. Fault localization in program repair has to satisfy two apparently conflicting requirements: it should be accurate (leading to few locations highly suspicious of error), but also efficient (not taking too much running time).

In this paper, we propose a novel fault localization approach—called *retrospective fault localization*, and presented in Section 3—that improves accuracy while simultaneously boosting efficiency by *integrating* closely within standard automated program repair techniques. By providing a more effective fault localization process, retrospective fault localization expands the space of possible fixes that can be searched practically. Retrospective fault localization leverages mutation-based fault localization [5], [6] to boost localization accuracy. Since mutation-based fault localization is notoriously time consuming, a key idea is to perform it as a *derivative* of the usual program repair process. Precisely, retrospective fault localization introduces a *feedback loop* that reuses, instead of just discarding them, the candidate fixes that fail validation to enhance the precision of fault localization. Candidate fixes that pass some tests that the original (buggy) program failed are probably closer to being correct, and hence they are used to refine fault localization so that other similar candidate fixes are more likely to be generated.

We implemented retrospective fault localization in a tool called RESTORE, built on top of JAID [7], a recent generate-and-validate automated program repair tool for Java. Experiments with real-world bugs from the DEFECTS4J curated benchmark [8] indicate that retrospective fault localization significantly improves the overall effectiveness of program repair in terms of correct fixes (for 41 faults in DEFECTS4J, 8 more than any other automated repair tool for

- *Tongtong Xu is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, Jiangsu 210023, China, and also with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong. E-mail: dz1633014@smail.nju.edu.cn.*
- *Liushan Chen and Yu Pei are with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Hong Kong. E-mail: {cslschen, csypei}@comp.polyu.edu.hk.*
- *Tian Zhang is with the State Key Laboratory for Novel Software Technology of Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: ztluck@nju.edu.cn.*
- *Minxue Pan is with the State Key Laboratory for Novel Software Technology and the Software Institute of Nanjing University, Nanjing, Jiangsu 210023, China. E-mail: mxp@nju.edu.cn.*
- *Carlo A. Furia is with the Software Institute of USI Università della Svizzera italiana, 6904 Lugano, Switzerland. E-mail: furiac@usi.ch.*

Manuscript received 27 May 2019; revised 22 Mar. 2020; accepted 7 Apr. 2020. Date of publication 0 . 0000; date of current version 0 . 0000.
(Corresponding authors: Yu Pei, Tian Zhang, Minxue Pan, and Carlo A. Furia) .
Recommended for acceptance by K. Sen.
Digital Object Identifier no. 10.1109/TSE.2020.2987862

Java at the time of writing) and boosts its efficiency (cutting JAID's running time to a third or less). Other measures of performance, discussed in detail in Section 4, suggest that retrospective fault localization improves the efficiency of automated program repair by supporting accurate fault localization with comparatively moderate resources.

Generality. While our prototype implementation is based on the existing tool JAID, retrospective fault localization should be applicable to any program repair tools that use fault localization and rely on validation through testing. To demonstrate the approach's generality, we extended SimFix [9]—another state-of-the-art automated repair tools for Java—with retrospective fault localization. The experimental results comparing SimFix with and without retrospective fault localization (reported in Section 4.2.3) indicate that retrospective fault localization is applicable also to different implementations, where it similarly brings considerable performance improvements without decreasing effectiveness.

Contributions. This paper makes the following contributions:

1) Retrospective fault localization: a novel fault localization approach tailored for automated program repair techniques based on validation;
2) RESTORE: a prototype implementation of retrospective fault localization, demonstrating how retrospective fault localization can work in practice;
3) An experimental evaluation of RESTORE on real-world faults from DEFECTS4J, showing that retrospective fault localization significantly improves the efficiency by boosting effectiveness and, simultaneously, performance.
4) An implementation of retrospective fault localization atop the SimFix program repair technique, indicating that it is viable to improve also other generate-and-validate repair techniques.

Replication. A replication package with RESTORE's implementation and all experimental data is publicly available at: http://tiny.cc/9xff3y.

2 AN EXAMPLE OF RESTORE IN ACTION

The *Closure Compiler* is an open source tool that optimizes JavaScript programs to achieve faster download and execution times. One of the refactorings it offers—renaming classes so that namespaces are no longer needed— is based on class `ProcessClosurePrimitives` whose methods modify calls to common namespace manipulation APIs. In particular, method `processRequireCall` processes calls to the `goog.require` API and determines if they can be removed without changing program behavior.

Listing 1 shows part of the method's implementation, which is defective:[1] according to the tool documentation, a call to `goog.require` should be removed (lines 6 and 7) if *(i)* the required namespace can be resolved successfully (`provided != null`), *or(ii)* the tool is configured to remove all the calls to `goog.require` unconditionally (`requiresLevel.isOn()`). But the code in Listing 1 only checks condition *(i)* on line 5, and hence does not remove unresolvable calls even when condition *(ii)* holds.

1. Fault *Closure113* in DEFECTS4J [8] and Table 3.

Listing 1: Faulty method `processRequireCall` from class `ProcessClosurePrimitives` in project *Closure Compiler*.

```
1 private void processRequireCall(NodeTraversal t,
2         Node n, Node parent) {
3   ProvidedName provided = providedNames.get(...);
4   ...
5   if (provided != null) {
6     parent.detachFromParent();
7     compiler.reportCodeChange();
8   }
9 }
```

Listing 2: Fix written by tool developers (replacing line 5 in Listing 1), and also produced by RESTORE.

```
if (provided != null || requiresLevel.isOn()) {
```

Using some of the tests that come with *Closure Compiler*'s source code, the RESTORE tool described in the present paper produces the fix shown in Listing 2, which is identical to the one written by *Closure Compiler*'s tool developers—and completely fixes the bug. At the time of writing, RESTORE is the only automated program repair tool capable of correctly fixing this bug[2].

The features of method `processRequireCall` and its enclosing class `ProcessClosurePrimitives` contribute to making the bug challenging for generate-and-validate automated repair tools. First, class and method are relatively large (Class `ProcessClosurePrimitives` has 1233 lines and method `processRequireCall` has 40 lines), which is a challenge in and of itself for precise fault localization. Second, attribute `requiresLevel` is never referenced in the faulty version of `processRequireCall` and is used only once after initialization in the whole class; thus, expression `requiresLevel.isOn()`—which is needed for the fix—is unlikely to be selected by techniques that look for fixing "ingredients" mainly in a fault's context.

RESTORE's retrospective fault localization is crucial to ensure that the necessary fixing expression is found in reasonable time: RESTORE takes around 32 minutes to produce the fix in Listing 2) and to rank it first in the output. This indicates that RESTORE's search for fixes is not only efficient but also effective.

In the rest of the paper we explain how RESTORE works (Section 3), and demonstrate its consistent performance improvements on standard benchmarks of real-world bugs (Section 4).

3 HOW RESTORE WORKS

Retrospective fault localization is applicable in principle to any generate-and-validate automated program repair technique to improve its efficiency. To make the presentation more concrete, we focus on how retrospective fault localization is applicable on top of the JAID [7] automated program repair tool. We call the resulting technique, and its supporting tool, RESTORE.

2. Nopol was able to produce a valid, but incorrect, fix to the fault [10].

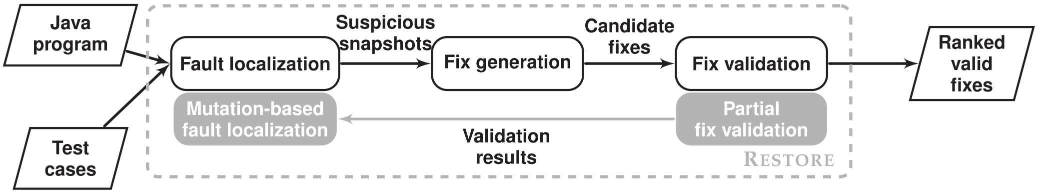


Fig. 1. An overview of how RESTORE works. RESTORE can improve the performance of any generate-and-validate automated program repair tool. Such a tool inputs a faulty program and some test cases exercising the program. The first, crucial, step of fixing is *fault localization*, which determines a list of snapshots: program states that are indicative of error; for each suspicious snapshot, *fix generation* builds a number of candidate fixes of the input program by exploring a limited number of program mutations that may avoid the suspicious states; *fix validation* reruns the available tests on each candidate built by fix generation; only candidates that pass all tests are *valid fixes*, which are the tool's output to the user. RESTORE kicks in during the first run of such a program repair tool, by introducing a feedback loop (in grey) that improves the effectiveness of fault localization. RESTORE performs a *partial fix validation*, whose goal is quickly identifying candidate fixes that fail validation—which are treated as *mutants* of the input program; information about how mutants' behaviors differ from the input program supports a *mutation-based fault localization* step that sharpens the identification of suspicious snapshots. As we demonstrate in Section 4, RESTORE's feedback loop significantly improves effectiveness and efficiency of automated program repair.

3.1 Overview

Fig. 1 illustrates how RESTORE works at a high level, and how it enhances a traditional automated program repair technique by retrospective fault localization (boxes in grey in Fig. 1).

Input. RESTORE inputs a Java program P (a collection of classes), with a faulty method `fixme`, and a set T of test cases exercising P; precisely, tests T are partitioned into *passing* tests $T^{\checkmark}$ and *failing* tests $T_{\times}$. Since each run of RESTORE actually only uses tests that exercise `fixme`, we assume, without loss of generality, that T only includes such tests.

Fault Localization identifies program locations and states (called *snapshots*) that are indicative of faulty behavior. According to heuristics based on dynamic and static measures, each snapshot receives a *suspiciousness score*—the higher, the more suspicious; snapshots ranked according to their suspiciousness score are input to the next step: fix generation.

Fix Generation builds several modifications of input program P for each snapshot in order of suspiciousness. The modifications try to mutate P's behavior in a way that avoids reaching the suspicious snapshot's state. Fix generation's output is a sequence of *candidate fixes* that needs to be validated.

(Full) Fix Validation tests each candidate fix to determine whether it actually fixes the fault exposed by $T_{\times}$. In traditional automated program repair, fix validation runs all available tests T against each fix candidate, and only outputs candidates that pass all tests—ranked according to the suspiciousness of the snapshots they were derived from. Hence, fix validation is often the most time-consuming step of traditional automated program repair. Since it is done downstream from fix generation—as the last step of the whole fixing process—validation requires a large number of fix candidates to maximize the chance of finding some valid, possibly correct, fixes, which exacerbates the performance problem.

Partial Fix Validation is the lightweight form of validation of candidate fixes used by RESTORE to support retrospective fault localization. By only running a subset of the available tests T, partial fix validation aims to quickly detect *behavioral changes* in some of the candidates with respect to the program P under fix.

Mutation-based fault localization improves the precision and effectiveness of fault localization by using *retrospective* information coming from partial validation. Based on this information, the suspiciousness score of snapshots is revised to become more discriminatory.

Exploring a Larger Fix Space. With retrospective fault localization, the top-ranked snapshots have a *higher chance* of leading to *valid fixes* when used in the following phases of the repair technique—and thus to correct fixes ranked high in the overall output. Conversely, a higher-precision fault localization technique means that *fewer candidates* need to be generated and (fully) validated, leading to an overall faster process. In turn, RESTORE's more efficient search of the fix space allows it to explore a *larger space* in comparable—often *shorter*—time, ultimately leading to discovering fixes that are outside JAID's fix space.

3.2 Basic Automated Program Repair

This section describes the basic process of automated program repair—as implemented in generate-and-validate repair tools such as JAID and RESTORE. Then, Section 3.3 presents retrospective fault localization in RESTORE, showing how it enhances the basic repair process described here.

3.2.1 State Abstraction: Snapshots

Snapshots are fundamental abstractions of a program's runs. A snapshot is a triple $\langle \ell, e, v \rangle$, where ℓ is a *location* in the program's control-flow graph, e is a Boolean expression, and v is a Boolean value (`true` or `false`). Intuitively, $\langle \ell, e, v \rangle$ records the information that a program's run reaches location ℓ with expression e evaluating to v.

RESTORE builds snapshots by enumerating different Boolean expressions e that refer to program features visible at ℓ, and by evaluating such expressions in all runs of tests T.

3.2.2 Fault Localization

Fault localization assigns a *suspiciousness score* $su(s)$ to each snapshot s. Intuitively, $su(s)$ should capture the likelihood that s is the source of failure.

Tools like JAID use a form of spectrum-based fault localization [11], which roughly corresponds to giving a higher suspiciousness to $s = \langle \ell, e, v \rangle$ the more often e evaluates to v at ℓ in runs of failing tests than in runs of passing tests. In RESTORE, we call JAID's fault localization *basic fault localization*;

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

Fig. 2. Schemas to build candidate fixes from a code snippet `action` built from snapshot $\langle \ell, e, v \rangle$, where `oldStatement` is the statement at ℓ in method `fixme` under fixing.

RESTORE uses it to determine a suspiciousness score $su_B(s)$ for each snapshot s—bootstrapping the fix generation phase.

More precisely, JAID applies Wong *et al.*'s Heuristic III [12] to classify the suspiciousness of *snapshots* rather than statements—as more commonly done in fault localization. A snapshot s's suspiciousness combines a static analysis score (measuring the syntactic similarity of the snapshot expression e and the code around location ℓ) and a dynamic score (measuring the relative frequency with which $e = v$ in a failing rather than in a passing test). Some recent experiments [13] indicate that JAID's effectiveness does not significantly depend on the details of the spectrum-based fault localization algorithm: running JAID using other common algorithms for fault localization (such as Ochiai [11] or Tarantula [14]) leads to very similar numbers of valid and correct fixes.

3.2.3 Fix Generation

For each snapshot $\langle \ell, e, v \rangle$, fix generation modifies P's method `fixme` (the one being fixed) in ways that affect the value of e at ℓ. Fix generation processes snapshots in decreasing order of suspiciousness, building multiple modifications of `fixme` for the same snapshot; each modification is a *fix candidate*.

RESTORE generates fix candidates in two steps. First, it enumerates code snippets (called *actions* in [7]) that (a) modify the state of an object referenced in e, (b) modify a subexpression of e in the statement at ℓ, (c) if ℓ is a conditional statement `if (c) ...`, modify expression `c`, or (d) modify the control flow at ℓ (for example with a `return` statement). Second, it injects a code snippet `action` into `fixme` using any of the five schemas in Fig. 2: `oldStatement` is the statement at ℓ in `fixme`, which the whole instantiated schema replaces to generate a fix candidate.

Each fix candidate C can be seen as a mutant of input program P that originates from one snapshot s; we write $\sigma(C) = s$ to denote the snapshot s that candidate C originates from. To cull the search space of generated fixes, it is customary to builds fix candidates for at most the top N snapshots in order of suspiciousness; in JAID, $N = N_S = 1500$.

3.2.4 Fix Validation (and Ranking)

Since fix generation is "best effort" and based on the partial information captured by snapshots, it is followed by a *validation* step that reruns all available tests. A fix candidate C is *valid* if it passes *all available tests* T: tests $T_\times$ failing on the input program are passing on C, and tests $T\checkmark$ passing on the input program are still passing on C (no regression errors).

Typically, more than one fix candidate C fixing the same input program P is valid; we *rank* all such valid fixes in decreasing order of suspiciousness of the snapshot used to generate C—that is in decreasing order of $su(\sigma(C))$. The overall output of automated program repair is thus a list of valid fixes ranked according to suspiciousness.

3.3 Retrospective Fault Localization in RESTORE

The ultimate goal of automated program repair is finding fixes that are not only valid—pass all available tests—but *correct*—equivalent to those a competent programmer, knowledgeable of the program P under repair, would write. The traditional automated program repair process presented in Section 3.2 can be quite effective at producing correct fixes but is limited in practice by two related requirements: 1) since the accuracy of fault localization greatly affects the chances of success of the whole repair process, we would like to have a fault localization technique that incorporates as much information as possible; 2) since the process is open loop (no feedback), we have to generate as *many candidate fixes* as possible to maximize the chance of finding a correct one. Improving accuracy and generating many candidate fixes both exacerbate the already significant problem of *long validation times* (for example, validation takes up 92.8 percent of JAID's overall running time [7]). More crucially, they require to bound the search space of possible fixes to a *size* that can be feasibly explored. But, by definition, shrinking the fix space makes some bugs impossible to fix.

Retrospective fault localization, as implemented in RESTORE, addresses these two requirements with complementary solutions: 1) it performs a preliminary *partial fix validation*, which runs much faster than full validation and whose primary goal is to supply more dynamic information to fault localization; 2) using the information from partial validation, it complements JAID's fault localization with precise *mutation-based fault localization*. Such a feedback-driven mutation-based fault localization drives more efficient further iterations of fix generation, producing a much smaller, often higher-quality, number of candidate fixes that can undergo full validation taking a reasonable amount of time. The greater efficiency is then traded off against fix space size: RESTORE can afford to explore a *larger space of candidate fixes*, thus ultimately fixing bugs that are out of JAID's (and other repair tools') capabilities.

3.3.1 Initial Fix Generation

The initial iteration of fix generation in RESTORE works similarly to basic automated program repair: fault localization (Section 3.2.2) assigns a basic suspiciousness score $su_B(s)$ to every snapshot s (using spectrum-based fault localization as in JAID); and fix generation (Section 3.2.3) builds fix candidates for the most suspicious snapshots.

As we have already remarked, JAID's spectrum-based fault localization often takes a major part of the total fixing time, as it involves monitoring the values of many snapshot expressions in every test execution; for example, it takes 51–99 percent of JAID's total time on 16 hard faults [7]. To cut down on this major time cost, RESTORE *selects* a subset T_B of all tests T to be used in basic fault localization using nearest neighbor queries [15]. The selected tests T_B include all failing tests $T_\times$ as well as the passing tests with the *smallest distance* to those failing. The distance between two tests t_1, t_2 is calculated as the Ulam distance[3] $U(\phi(t_1), \phi(t_2))$, where $\phi(t)$ is a sequence with all basic blocks of `fixme`'s control-flow graph sorted according to how many times each block is executed when running t. This way, passing tests that are behaviorally similar to failing tests are selected as "more useful" for fault localization since they are more likely to be sensitive to fixes of the fault. Take, for example, the conditional at lines 5–7 in Listing 2; two tests t_1 and t_2 such that `provided != null` at line 5 both execute the conditional block, and hence will have a shorter Ulam distance than t_1 and another test t_3 that skips the conditional block (such that `provided == null` at line 5). Subset T_B is used only to bootstrap RESTORE's initial fix generation without dominating the overall running times.

During initial fix generation, RESTORE builds fix candidates for the $N_1 = N_S \cdot N_P$ most suspicious snapshots (whereas JAID builds candidates for the N_S most suspicious snapshots). Parameter N_P is 10 percent (i.e., $N_P = 0.1$) by default; this works because retrospective fault localization can be as effective as JAID's basic fault localization with a fraction of the snapshots.

3.3.2 Partial Fix Validation

Partial fix validation aims at quickly extracting dynamic information about the many candidate fixes built by the initial iteration of fix generation. To strike a good balance between costs (time spent on running tests) and benefits (information gathered to guide mutation-based fault localization), partial fix validation follows the simple strategy of running only the tests $T_\times$ that were failing on the input program P. still has a good chance of providing valuable information for fault localization, since it detects whether the failing behavior has changed in some of the fix candidates.

If a candidate fix happens to pass all tests $T_\times$, it immediately undergoes full validation (Section 3.3.6) for better responsiveness of the fixing process (outputting valid fixes as soon as possible).

3.3.3 Mutation-Based Fault Localization

In mutation-based fault localization [5], [6], we compare the dynamic behavior of many different *mutants* of a program.

A mutant is a program variant produced by changing the program's code in some ways—for example, by changing a comparison operator. A mutant M of a program P is *killed* by a test t when M behaves differently from P on t; that is, either P passes t while M fails it, or P fails t while M passes it. A killed mutant M indicates that the locations where M syntactically differs from P are likely (if M fails) or unlikely (if M passes) to be implicated with the failure triggered by t.

RESTORE's retrospective fault localization treats candidate fixes as *higher-order mutants*—that is, mutants of the input program P that may include *multiple* elementary mutations—and interprets partial fix validation results of those higher-order mutants in a similar way to help locate faults more accurately. In particular, adapting [6]'s heuristics to our context, we assign a suspiciousness score $su_M(C)$ to each *candidate fix* C:

$$su_M(C) = \frac{|T_\times \cap \mathit{killed}(C)|}{\sqrt{|T_\times| \cdot |\mathit{killed}(C)|}}, \tag{1}$$

where $\mathit{killed}(C) \subseteq T_\times$ is the set of all tests that kill C—and thus $T_\times \cap \mathit{killed}(C)$ are the tests that fail on input program P and pass on C. Formula (1) assigns a higher suspiciousness to a candidate fix the more failing tests it manages to pass, indicating that C might be closer to correctness than P.

In order to combine the output of mutation-based and basic fault localization, we assign a suspiciousness score $su_M(s)$ to each *snapshot* s based on the suspiciousness (1) of *candidates*. Each candidate fix D is generated from some snapshot $\sigma(D)$; let $SU(D)$ be the largest suspiciousness score of all candidate fixes E generated from the same snapshot $\sigma(D)$ as D:

$$SU(D) \;=\; \max_{E} \left\{ su_M(E) \,|\, \sigma(E) = \sigma(D) \right\}.$$

Then, the mutation-based suspiciousness score $su_M(s)$ of a *snapshot* $s = \langle \ell, e, v \rangle$ is the average of $SU(D)$ across all candidate fixes D generated from a snapshot with the same location ℓ as s (and any expression and value):

$$su_M(\langle \ell, e, v \rangle) \;=\; \operatorname*{mean}_{D} \left\{ SU(D) \,|\, \sigma(D) = \langle \ell, *, * \rangle \right\}. \tag{2}$$

The maximum selects, for each snapshot, the candidate fix generated from it that is more "successful" at making failing tests pass. Then, all snapshots with the same location get the same "average" suspiciousness score. Intuitively, the average pools the information from different fixes that target different locations and pass partial validation.

Finally, we combine the basic suspiciousness score su_B and the mutation-based suspiciousness score su_M into an overall total ordering of snapshots according to their suspiciousness:

$$s_1 \preceq s_2 \;\triangleq\; \begin{array}{c} \\ \vee \end{array} \begin{array}{l} \big(\ell_1 \neq \ell_2 \;\wedge\; su_M(s_1) \geq su_M(s_2)\big) \\ \big(\ell_1 = \ell_2 \;\wedge\; su_B(s_1) \geq su_B(s_2)\big) \end{array},$$

where $s_1 = \langle \ell_1, e_1, v_1 \rangle$ and $s_2 = \langle \ell_2, e_2, v_2 \rangle$. That is, snapshots referring to different locations are compared according to their mutation-based suspiciousness, and snapshots referring to the same location are compared according to their basic suspiciousness—because they have the same mutation-based suspiciousness score. As discussed in Section 3.2.2, RESTORE assigns a basic suspiciousness score to each *snapshot*; whereas the mutation-based suspiciousness score (2) is the same, by definition, for all snapshots with the same location.

An Example of How MBFL Works. To get a more intuitive idea of how mutation-based fault localization can help find suitable fix locations in RESTORE, let's consider again fault *Closure113* in DEFECTS4J—shown in Fig. 1 and discussed in Section 2.

A single failing test case $T_\times = \{t_\times\}$ triggers the fault by reaching line 5 with `provided == null`: execution skips the *then* branch (lines 6 and 7), which eventually leads to a failure.

During the initial round of fix generation, RESTORE does not produce any valid fix, because a key fix ingredient (expression `requiresLevel.isOn()`) is further out in the

3. The Ulam distance [16] of two sequences is the minimum number of delete, shift, and insert operations to go from one sequence to another. For example, the Ulam distance $U(s_1, s_2)$ of $s_1 = a\,b\,c\,t\,u$ and $s_2 = a\,b\,t\,c\,u$ is 2 (delete c from s_1 and insert it back after t).

fix search space. However, it generates 16 candidate fixes that happen to pass the originally failing $T_\times$ because they all force execution through lines 6 and 7 by changing condition `provided != null` on line 5. For example, one such fixes replaces it with `provided != null` ‖ `provided == null`. None of these 16 candidates is valid (because they all fail other, previously passing, tests) but, instead of simply being discarded, they all are reused as evidence—to increase the suspiciousness score of line 5: *(i)* $su_M(C) = 1$ for each of these 16 candidates, because $|T_\times| = 1$ and $killed(C) = T_\times$; *(ii)* $SU(C) = su_M(C)$ for the same candidates, because they all have the same (maximum) value of suspiciousness; *(iii)* $su_M(\langle \ell = 5, *, * \rangle) = 1$ for all snapshots that target line 5. Since no other candidates generated in this round change the suspiciousness of other locations, the net result is that the following iterations of fix generation will preferentially target fixes at line 5. This biases the search for fixes so that RESTORE goes deeper in this direction of the fix search space, which eventually leads to generating the correct fix shown in Listing 2—which indeed targets line 5 with a suitable condition.

3.3.4 Retrospective Loop Iteration

Equipped with the refined fault localization information coming from mutation-based fault localization, RESTORE decides whether to iterate the retrospective fault localization loop—entering a new round of initial fix generation (Section 3.3.1)—or to just use the latest fault localization information to perform a final fix generation (Section 3.3.5). While the retrospective feedback loop could be repeated several times (until all snapshots are used to build candidates), we found that there are diminishing returns in performing many iterations. Thus, the default setting is to stop iterating as soon as mutation-based fault localization assigns a *positive* suspiciousness score $su_M(s)$ to *some* snapshot s; if no snapshot gets a positive score, we repeat initial fix generation.

3.3.5 Final Fix Generation

Snapshots ranked according to the $\preceq$ relation drive the final generation of fixes. Final fix generation runs when retrospective fault localization has successfully refined the suspiciousness ranking of snapshots (Section 3.3.4)—hopefully identifying few promising snapshots. Thus, final fix generation generates fixes *only* for snapshots corresponding to the N_L most suspicious locations—with $N_L = 5$ by default.

During final fix generation, RESTORE can even afford to trade off some of the greater precision brought by retrospective fault localization for a *larger fix space* to be explored: whereas JAID builds fix candidates based only on expressions found in method `fixme` (the method being fixed), RESTORE may also consider expressions found anywhere in `fixme`'s enclosing *class*. RESTORE can efficiently search such a larger fix space, thus significantly expanding its overall fixing effectiveness.

3.3.6 (Full) Fix Validation

The final validation is, as in basic automated program repair, full—that is, uses *all* available tests T and validates candidate fixes that pass all of them. This validation has a higher chance of being significantly faster than in basic automated program repair: first, it often has to consider fewer candidate fixes (Section 3.3.5) selected according to their mutation-based suspiciousness; second, several candidate fixes have already undergone partial validation against failing tests $T_\times$ (Section 3.3.2), and thus only need to be validated against the originally passing tests $T_\checkmark$.

Fixes that pass validation are output to the user in the same order of suspiciousness $\preceq$ as the snapshots used to generate them. Thus, RESTORE's overall output is a list of valid fixes ranked according to suspiciousness.

4 EXPERIMENTAL EVALUATION

We implemented the RESTORE technique in a tool, also called RESTORE, based on the JAID program repair system. Our experimental evaluation assesses to what extent RESTORE is an effective automated program repair tool by comparing: *(i)* RESTORE's results on high-level metrics, such as *bugs correctly fixed*, to other program repair tools for Java; *(ii)* RESTORE's results on fine-grained metrics, such as the effectiveness of *fault localization*, to JAID—a state-of-the-art repair tool for Java which RESTORE directly extends; *(iii)* the effects of extending SimFix—another recent generate-and-validate repair tool for Java—with *retrospective* fault localization (RESTORE's key technical improvement). Overall, the evaluation indicates that RESTORE is a substantial advance in general-purpose automated program repair for Java. Different parts of the evaluation have different levels of granularity, so that the we can also track *which* ingredients used by RESTORE are effective and which metrics they impact.

RQ1: What is RESTORE's *effectiveness* in fixing bugs?
In RQ1, we consider RESTORE from a user's perspective: how many valid and correct fixes it can generate.

RQ2: What is RESTORE's *performance* in fixing bugs?
In RQ2, we consider RESTORE's efficiency: how quickly it runs versus how large a fix space it explores.

RQ3: How well does retrospective *fault localization* (RFL) work in RESTORE?
In RQ3, we zoom in on RESTORE's fault localization technique to assess how efficiently it drives the search for a valid fix.

RQ4: How *robust* is RESTORE's behavior when its internal parameters are changed?
In RQ4, we evaluate the impact of disabling features like partial validation and of changing some parameters that regulate retrospective fault localization.

RQ5: Is retrospective fault localization *generally applicable* to generate-and-validate program repair techniques?
In RQ5, we look for evidence that retrospective fault localization is applicable not only to JAID but also to other automated program repair techniques.

TABLE 1
Basic Measures of size for projects in DEFECTS4J.

PROJECT	FULL NAME	KLOC	#TESTS	#FAULTS
Chart	JFreechart	96	2205	26
Closure	Closure Compiler	90	7927	133
Lang	Apache Commons-Lang	22	2245	65
Math	Apache Commons-Math	85	3602	106
Time	Joda-Time	27	4130	27
	TOTAL	320	20109	357

For each PROJECT in DEFECTS4J, its FULL NAME, the size KLOC in thousands of lines of code, the number of tests #TESTS, and the number of distinct faults #FAULTS.

Comparison to Other Tools. We compare RESTORE's results on high-level metrics to the 13 state-of-the-art automated program repair systems for Java listed in Table 2. To our knowledge these 13 tools include all recent Java repair tools evaluated on DEFECTS4J and published, at the time of writing, in major software engineering conferences in the last couple of years.

4.1 Subject Faults

As it has become customary when evaluating automated program repair tools for Java, our experiments use real-world faults in the DEFECTS4J curated collection [8]. DEFECTS4J includes hundreds of faults from open-source Java projects; each fault comes with at least one test triggering the failure—in addition to other passing or failing tests—as well as a programmer-written fix for the fault. Table 1 shows basic measures of size for DEFECTS4J's 357 faults in 5 projects.

4.2 Experimental Protocol

Each experiment runs RESTORE, JAID, or another tool to completion on a fault in DEFECTS4J. In each run we record several measures such as:

#V: number of *valid* fixes in the output;
C: rank of the first *correct* fix in the output;
T: overall wall-clock running *time*;
T2V: wall-clock *time* until the first *valid* fix is found;
T2C: wall-clock *time* until the first *correct* fix is found;
C2V: number of fixes that are *checked* (generated and validated) until the first *valid* fix is found;
C2C: number of fixes that are *checked* (generated and validated) until the first *correct* fix is found.

Measures C2V and C2C include all kinds of validation. For example, RESTORE performs partial and full validation (see Section 3.3.2 and Section 3.3.6); JAID uses only one kind of (full) validation.

Correctness. We determined correct fixes by manually going through the output list of valid fixes and comparing each of them to DEFECTS4J's manually-written fix for the fault under repair: a valid fix is correct if it is *semantically equivalent* to the fix manually written by the developers and included in DEFECTS4J. Conservatively, we mark as incorrect fixes that we cannot conclusively establish as equivalent in a moderate amount of time (around 15 minutes per fix).

Hardware/software setup. All the experiments ran on the authors' institution's cloud infrastructure. Each experiment used 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.

4.2.1 Statistics

Table 4 reports detailed *summary statistics* directly comparing RESTORE to JAID. For each measure m taken during the experiments (e.g., time T), let $J_{m,k}$ and $R_{m,k}$ denote the value of m in JAID's and in RESTORE's run on fault k. We compare RESTORE to JAID using these metrics (illustrated and justified below) [17]:

$\frac{\sum Restore}{\sum Jaid}$: the ratio $\sum_k J_{m,k} / \sum_k R_{m,k}$ expressing the *relative cost* of RESTORE over JAID for measure m.

TABLE 2
A Quantitative Comparison of RESTORE With 13 Other Tools for Automated Program Repair on DEFECTS4J Bugs

TOOL	VALID	ANY POSITION			FIRST POSITION			TOP-10 POSITION			UNIQUE
		CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	CORRECT	PRECISION	RECALL	
RESTORE	98	41	42%	11%	19	20%	5%	29	30%	8%	8
ACS [19]	23	18	78%	5%	18	78%	5%	18	78%	5%	12
CapGen [20]	25	22	88%	6%	21	84%	6%	22	88%	6%	3
Elixir [21]	41	26	63%	7%	26	63%	7%	26	63%	7%	0
HDA [22]	?	23	?	6%	13	?	4%	23	?	6%	3
JAID [7]	31	25	81%	7%	9	29%	3%	15	48%	4%	1
jGenProg [23]	27	5	19%	1%	5	19%	1%	5	19%	1%	1
jKali [23]	22	1	5%	0%	1	5%	0%	1	5%	0%	0
Nopol [23]	35	5	14%	1%	5	14%	1%	5	14%	1%	2
SimFix [9]	56	34	61%	10%	34	61%	10%	34	61%	10%	12
SketchFix [24]	26	19	73%	5%	9	35%	3%	?	?	?	0
SketchFixPP [24]	?	34	?	10%	?	?	?	?	?	?	2
ssFix [25]	60	20	33%	6%	20	33%	6%	20	33%	6%	1
xPar [19], [22]	?	4	?	1%	?	?	?	4	?	1%	0

For each program repair TOOL, the table references the source of its experimental evaluation data reported here: the number of bugs that the tool could fix with a VALID fix; the number of bugs that the tool could fix with a CORRECT fix; and the resulting PRECISION (CORRECT/VALID) and RECALL (CORRECT/357, where 357 is the total number of DEFECTS4J faults used in the experiments). For tools whose data about the POSITION of fixes in the output ranking is available, the table breaks down the data separately for fixes ranked in ANY POSITION, in the FIRST POSITIONS, and in the TOP-10 POSITION. (These measures do not change for tools that output at most one fix per fault.) The rightmost column UNIQUE lists the number of distinct bugs that only the tool can correctly fix. Question marks represent data not available for a tool.

mean(JAID- RESTORE): the *mean difference* (using arithmetic mean) $\mathrm{mean}_k(J_{m,k} - R_{m,k})$ expressing the *average additional cost* of JAID over RESTORE for measure m.

$b_l, \widehat{b}, b_h$: the estimate $\widehat{b}$ and the 95 percent probability interval (b_l, b_h) of the *slope* b of the linear regression $R_{m,k} = a + b \cdot J_{m,k}$ expressing RESTORE's measure m as a linear function of JAID's.

$\widehat{\chi}, \chi_h$: for the same linear regression, the estimate $\widehat{\chi}$ and the 95 percent probability upper bound χ_h of the crossing ratio (where the regression line crosses the "no effect" line).

Each summary statistics compares RESTORE to JAID on faults on which the statistics is defined for both tools; for example, the mean difference of measure C (rank of first correct fix) is over the 23 faults that *both* RESTORE and JAID can correctly fix.

Interpretation of Linear Regression. A linear regression $y = a + b \cdot x$ estimates coefficients a (intercept) and b (slope) in a way that best captures the relation between x and y. A linear regression algorithm outputs *estimates* $\widehat{a}$ and $\widehat{b}$ and *standard errors* ϵ_a and ϵ_b for both coefficients: the "true" value of a coefficient c lies in interval (c_l, c_h), where $c_l = \widehat{c} - 2\,\epsilon_c \leq \widehat{c} \leq \widehat{c} + 2\,\epsilon_c = c_h$, with 95 percent probability.

In our experiments, values of x measure JAID's performance and values of y measure RESTORE's;[4] thus, the linear regression line expresses RESTORE's performance as a linear function of JAID's. The line $y = x$ (that is, $a = 0$ and $b = 1$) corresponds to *no effect*: the two tool's performances are identical. In contrast, lines that lie *below* the "no effect" line indicate that RESTORE measures consistently *lower* than JAID; since for all our measures "lower is better", this means that RESTORE performs better than JAID. Plots such as those in Fig. 4 display the estimated regression line with a shaded area corresponding to the 95 percent probability error interval; thus we can visually inspect whether the difference with respect to the dashed "no effect" line is significant with 95 percent probability by checking whether the shaded area lies under the dashed line.

Analytically, RESTORE is *significantly better* than JAID at the 95 percent probability level if the 95 percent probability upper bound b_h on the regression slope's estimate satisfies $b_h < 1$: the slope is different from (in fact, less than) the "no difference" value 1 with 95 percent probability.

Since this notion of significant difference does not consider the intercept, it only indicates that RESTORE's is better *asymptotically*; to ensure that the difference is significant in the range of values that were actually measured, we consider the *crossing ratio* $\widehat{\chi} = (\overline{x} - \min(Jaid))/(\max(Jaid) - \min(Jaid))$, which expresses the coordinate $x = \overline{x}$ where the regression line $y = \widehat{a} + \widehat{b}x$ crosses the "no effect" line $y = x$ relative to JAID's range of measured values (the crossing ratio upper bound χ_h is computed similarly but using the upper bounds a_h and b_h of a's and b's 95 percent probability intervals). A large crossing ratio means that RESTORE is better than JAID only on "hard" faults, whereas a small crossing ratio means that RESTORE is consistently better across the experimented range, as illustrated in the example of Fig. 3.

4. In Section 4.3.5, x measures SimFix's performance and y measures the performance of SimFix+ (SimFix with retrospective fault localization).

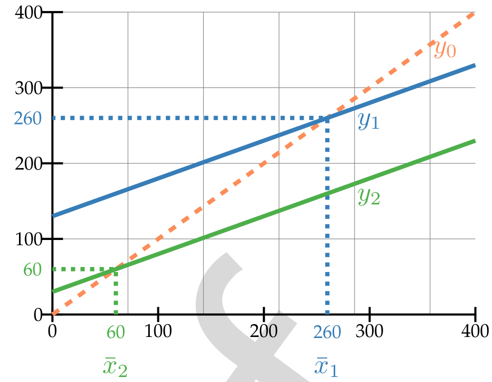


Fig. 3. Visual explanation of linear regression lines. The two regression lines $y_1 = 130 + 0.5\,x$ and $y_2 = 30 + 0.5\,y$ have the same slope but different intercepts. Therefore, y_2 crosses the "no effect" line $y_0 = x$ at $\bar{x}_2 = 60$, much earlier than y_1 that crosses it at $\bar{x}_1 = 260$. The crossing ratio scales the crossing coordinates $\bar{x}_1$ and $\bar{x}_2$ over the range of values on the x axis. If the range is the whole x axis from 0 to 400, the crossing ratios are simply $\chi_1 = \bar{x}_1/400 = 0.15$ and $\chi_2 = \bar{x}_2/400 = 0.65$, which indicate that y_1 is above y_0 for only 15 percent of the data, and y_2 for 65 percent of the data.

Summarizing Data With Linear Regression. Using linear regression to model data that doesn't "look" linear may seem unsound. However, it is not a problem in our case given how we use linear regression: not to *predict* the performance of RESTORE on yet to be seen inputs, but simply to *summarize* the experimental data in a way that accounts for some measurement errors (and hence is more robust than just summarizing the raw data). After all, the essence of linear regression is a mechanism to "learn about the mean and variance of some measurement, using an additive combination of other measurements" [18], which is all we use it for in analyzing our experimental data.

4.2.2 Robustness of Retrospective Fault Localization

As described in Section 3.3.2, retrospective fault localization initially performs a *partial* validation of candidate fixes—using only failing tests. To understand the usefulness of partial validation, we built RESTORE-FULL: a variant of RESTORE that only performs *full* validation—always using all available tests.[5] In Section 4.3.4, we compare RESTORE and RESTORE-FULL on DEFECTS4J faults.

In its current implementation, RESTORE's behavior depends on several parameters: it uses the $N_S = 1500$ most suspicious state snapshots for fixing (Section 3.2.3); it adds $N_P =$ 10 percent more snapshots in each iteration of retrospective fault localization, and performs $N_I = 0$ extra iterations after a new suspicious location has been found (Section 3.3); it targets the $N_L = 5$ most suspicious locations for final fix generation (Section 3.3.5). To understand whether these parameters influence RESTORE's behavior, we modified one of them at a time and ran RESTORE on the same DEFECTS4J faults with these different settings. In Section 4.3.4, we report how changing each parameters affects the number of faults repaired with valid fixes, the number of faults repaired with correct fixes, and the running time across all faults where RESTORE is able to produce at least one valid fix.

5. Since full validation may blow up the running time when many tests are available for a fault, we do not run RESTORE-FULL to completion but set a cut-off time equal to twice overall running time of RESTORE on the fault.

TABLE 3
Summary of the Experimental Results

FAULT ID			#TEST		RESTORE					JAID				
PROJECT	ID	LOC	P	F	#V	C	T	T2V	T2C	#V	C	T	T2V	T2C
chart	1	32	37	1	291	221	28.5	7.5	21.6	536	84	54.1	5.6	19.9
chart	9	38	1	1	17	-	14.4	3.3	-	52	43	72.2	3.6	20.8
chart	11	32	15	1	1	1	19.4	17.6	17.6	0	-	-	-	-
chart	24	6	0	1	2	1	26.7	25.0	25.0	2	1	16.8	15.0	15.0
chart	26	108	23	22	213	3	32.7	11.5	12.2	82	1	53.6	15.2	15.2
closure	5	98	56	1	4	1	247.3	186.3	186.3	2	-	975.9	493.5	-
closure	11	18	2261	2	434	20	846.8	167.5	201.5	0	-	-	-	-
closure	14	97	3005	3	1	1	355.0	123.5	123.5	0	-	672.2	-	-
closure	18	122	3929	1	1	1	561.4	101.5	101.5	5	1	1367.1	518.0	518.0
closure	31	122	3835	1	12	1	570.6	118.4	118.4	9	8	1440.1	1068.2	1181.5
closure	33	27	259	1	171	141	290.8	19.2	266.7	2720	1	258	6.9	6.9
closure	40	46	305	2	5	1	25.9	6.1	6.1	4	1	119.5	27.4	27.4
closure	46	11	10	3	161	116	24.1	4.2	21.3	0	-	-	-	-
closure	62	45	45	2	122	90	37.5	10.3	30.4	87	31	126.7	8.1	31.9
closure	63	45	45	2	122	49	34.8	8.8	20.3	87	31	127.1	8.1	31.7
closure	70	19	2337	5	1	1	127.9	105.3	105.3	5	1	70.4	31.9	31.9
closure	73	70	482	1	1	1	49.2	39.4	39.4	1	1	473.4	413.5	413.5
closure	86	39	52	7	1	1	8.9	6.1	6.1	0	-	-	-	-
closure	113	39	26	1	1	1	48.7	32.5	32.5	0	-	26.8	-	-
closure	115	69	151	5	761	1	853.4	4.3	4.3	0	-	-	-	-
closure	118	23	19	2	4	3	33.0	24.6	29.7	0	-	12.3	-	-
closure	119	124	764	1	2	2	113.5	94.9	113.4	0	-	-	-	-
closure	125	15	538	1	103	103	154.1	13.1	151.0	98	-	131.3	9.7	-
closure	126	95	71	2	39	1	103.6	7.8	7.8	425	1	601.4	8.4	8.4
closure	128	9	61	1	14	1	37.8	9.3	9.3	0	-	-	-	-
closure	130	36	301	1	15	4	239.1	216.9	221.4	0	-	-	-	-
lang	6	24	35	1	51	5	142.3	6.6	19.7	0	-	-	-	-
lang	33	11	0	1	3	1	21.7	11.6	11.6	7	1	11.0	5.5	5.5
lang	38	6	33	1	69	18	6.7	1.5	4.0	28	4	10.7	1.1	1.2
lang	45	37	0	1	40	-	35.6	6.5	-	68	34	105.1	9.6	58.5
lang	51	51	0	1	37	1	8.1	4.2	4.2	424	46	188.4	5.4	15
lang	55	6	4	1	29	10	12.5	1.1	3.0	15	3	3.6	0.4	0.9
lang	59	17	2	1	12	7	31.7	5.0	11.8	0	-	-	-	-
math	5	22	5	1	225	1	43.1	3.2	3.2	61	1	11.3	0.6	0.6
math	32	52	6	1	2	1	10.2	9.2	9.2	5	4	37.5	18.9	32.2
math	33	40	21	1	2	2	114.9	74.0	74.1	0	-	251.6	-	-
math	50	125	3	1	812	94	489.2	98.5	137.6	1101	28	1502.6	54.3	93.5
math	53	5	19	1	10	9	60.0	25.2	51.3	10	6	19	11.1	13.3
math	59	2	0	1	2	1	3.4	2.4	2.4	0	-	0.9	-	-
math	80	15	16	1	1450	936	86.9	13.2	65.2	3877	1366	156.7	2.8	58.0
math	82	15	13	1	44	22	63.9	3.6	25.5	13	9	33.1	3.4	22.7
math	85	43	12	1	235	5	16.7	3.9	3.9	709	4	68.3	1.5	1.5
time	19	31	721	1	38	30	15.5	10.4	14.8	0	-	-	-	-
TOTAL		1887	19518	88	5560	-	6047.1	1645.0	425.9	10433	-	8998.7	2747.7	2625.0

For each fault in DEFECTS4J (identified by its PROJECT name and ID) that RESTORE or JAID can correctly fix: the size LOC of the faulty method being repaired (in lines of code), and the number of Passing and Failing tests exercising the method; for each tool RESTORE and JAID: the number #V of valid fixes; the position C of the first correct fix in the output; the wall-clock running time T to completion; the wall-clock running time until the first valid fix (T2V) and the first correct fix (T2C) are found. All times are in minutes.

4.2.3 General Application of Retrospective Fault Localization

To support our claim that retrospective fault localization is applicable to program repair tools other than JAID, we implemented it atop the SimFix [9] automated program repair system.[6] We picked SimFix because it is a state-of-the-art repair technique for Java (as shown in Table 2, it correctly fixes the largest number of DEFECTS4J bugs when only one fix per bug is considered) and because its source code and replication package are publicly available.

The key mechanism of retrospective fault localization is the feedback loop that uses the information gathered during partial validation of candidate fixes to tune fault localization; this mechanism is general—and hence it is present both in RESTORE and SimFix+. On the other hand, *how* the feedback loop collects and processes information, and precisely *when* it does so depends on the details of the technique to which retrospective fault localization is applied. Let's see what peculiarities of SimFix affected our implementation of retrospective fault localization in SimFix+.

6. We used the latest revision `c2a5319` from SimFix's repository https://github.com/xgdsmileboy/SimFix.

TABLE 4
Summary Statistics of the Experiments

MEASURE	$\frac{\sum \text{Restore}}{\sum \text{Jaid}}$	mean(Jaid − Restore)	*slope* b: 95% b_l	$\widehat{b}$	b_h	*crossing* χ $\widehat{\chi}$	χ_h
#v	0.44	181	0.2	0.3	0.4	0.02	0.04
c	0.98	1	0.6	0.7	0.8	0.05	0.13
t	0.32	214	0.2	0.2	0.3	0.02	0.04
t2v	0.29	83	0.1	0.1	0.2	0.02	0.04
t2c	0.42	64	-0.0	0.1	0.2	0.03	0.07
c2v	0.43	1498	0.2	0.3	0.4	0.03	0.07
c2c	0.64	602	-0.2	0.1	0.3	0.11	0.26

For each MEASURE*: the* relative cost $\frac{\sum Restore}{\sum Jaid}$ *of* Restore *over* Jaid*; the* mean cost difference mean($Jaid - Restore$) *between* Jaid *and* Restore*; the estimate* $\widehat{b}$ *of* slope b *expressing* Restore*'s cost as a linear function of* Jaid*, with 95 percent probability interval* (b_l, b_h)*; the estimate* $\widehat{\chi}$ *and upper bound* χ_h *on the* crossing ratio χ.

A key difference between Jaid (and hence Restore) and SimFix is that the latter's fault localization process, like most automated repair techniques', targets *statements* as possible fault locations—rather than snapshots. Precisely, SimFix applies the Ochiai [11] spectrum-based fault-localization technique to rank statements according to their suspiciousness. For each statement above a certain suspiciousness rank, SimFix searches for "donor code" (code snippets in the same project that are similar to those close to the suspicious statement), extracts modification patterns from the donors, and builds candidate fixes by matching these patterns to the suspicious statement. To winnow the many candidate fixes that are generated by this process, it tries to match them against a "catalog" of fixes—which is generated by mining programmer-written repairs during a preliminary phase done once before running SimFix on all bugs. As soon this process determines one fix that is valid (i.e., passes all available tests), SimFix stops.

We call SimFix+ the modified version of SimFix we built by adding retrospective fault localization. Just like Restore, SimFix+ undergoes a feedback loop: after a few candidate fixes are generated, their partial validation results inform a more accurate iteration of fault localization. In SimFix+, each iteration of the feedback loop uses M_P percent more code snippets for each suspicious statement to generate a few candidates fixes to "seed" retrospective fault localization. M_P is set to 20 percent for the initial iterations and 10 percent for the others, which is usually sufficient to generate enough candidates to drive the process; if this is not the case (namely, it generates less than 20 candidates), SimFix+ repeatedly increases M_P, by 10 percent each time, until at least 20 candidates are produced or all code snippets are used.

Like in Restore, partial validation in SimFix+ runs only the *failing* tests for the current bug. As soon it finds a candidate fix that passes at least one failing test ("the mutant is killed"), the candidate's fixing location increases its suspiciousness score, and hence SimFix+ immediately begins a new iteration that generates all fixes at that location and validates them. This behavior is different from Restore's—where a new iteration only begins after all candidates have undergone partial validation—but is consistent with SimFix's standard behavior of stopping as soon as it finds one valid fix.

In Section 4.3.5, we experimentally compare SimFix and SimFix+ by running both on Defects4J faults. Each fixing experiment used exclusively one virtual machine instance running Ubuntu 16.04 on two cores of an Intel Xeon Processor E5-2630 and 8 GB of RAM. Using the same setting as in the original experiments [9], each SimFix (and SimFix+) run is forcefully terminated after a 300-minute timeout if it is still running.

4.3 Experimental Results

In this section, we report the experiment results as answers to the research questions.

4.3.1 RQ1: Effectiveness

RQ1 assesses the *effectiveness* of Restore in terms of the *valid* and *correct* fixes it can generate.

Since most automated program repair tools for Java have been evaluated on the same Defects4J bugs as Restore, we can compare *precision* and *recall* of the various tools in Table 2.[7] Restore and Jaid can output multiple, ranked valid fixes for the same bugs; in contrast, other tools often stop after producing one valid fix. We keep this discrepancy into account in Table 2 by reporting different values of precision and recall according to whether we consider all valid fixes, only those in the top-10 positions, or only those produced in the top position (the first produced).

Valid fixes. Restore produced at least one valid fix for 97 faults in Defects4J. As shown in Table 2, that is more than any other automated repair tools for Java.

On the 36 faults that Jaid can also handle, Restore often produces *fewer valid fixes* than Jaid: overall, Restore produces 56 percent ($1 - 0.44$) fewer valid fixes than Jaid; and produces more valid fixes for only 13 faults. As we'll see later, Restore also produces *more* correct fixes than Jaid; thus, fewer valid fixes per bug can be read as an advantage in these circumstances.

Correct fixes. Restore produced at least one correct fix for 41 faults in Defects4J—when considering all fixes for the same bug. As shown in Table 2, that is more than any of the other automated repair tools for Java, and constitutes a 21 percent increase (7 faults) over the runners-up SimFix and SketchFix according to this metric. Restore correctly fixed 8 faults that *no other tool* can currently fix, in addition to the 6 faults that only Restore and Jaid can fix. This indicates that Restore's fix space is somewhat *complementary* to other repair tools for Java.

The output list of valid fixes should ideally rank correct fixes *as high as possible*—so that a user combing through the list would only have to peruse a limited number of fix suggestions. For the 23 faults that both Restore and Jaid correctly fix, the two tools behave similarly on the majority of bugs: Restore ranks the first correct fix 1 position higher than Jaid on average; and ranks it lower in 11 faults. Even thought this difference between the two tools is limited, Restore still fixes 18 more bugs than Jaid, and ranks first 8 of them. In addition, Fig. 4b suggests that Restore's advantage over Jaid emerges with "harder" faults with many valid fixes—where a reliable ranking is more important for practical usability.

7. Since these experimental all refer to the same set of bugs (without cross-validation), precision and recall have a narrower scope as effectiveness metrics here than they have in the context of information retrieval.

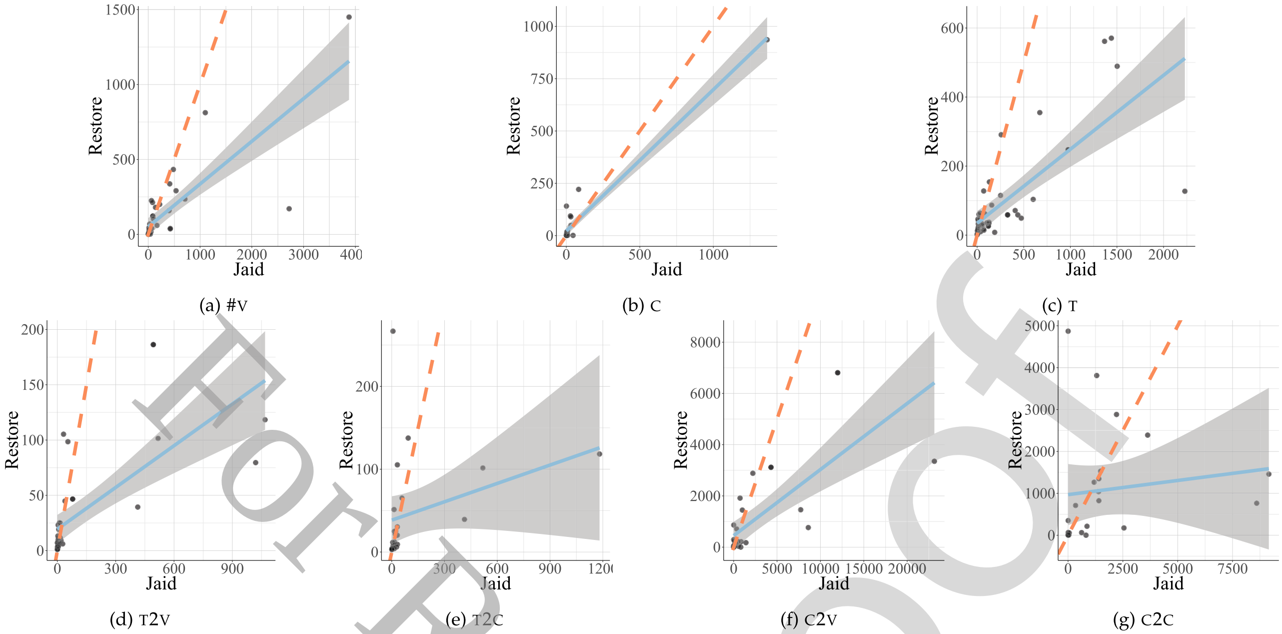


Fig. 4. Comparison of JAID and RESTORE on various measures. For each measure m, a point with coordinates $x = J_{m,k}, y = R_{m,k}$ indicates that JAID costed $J_{m,k}$ of m on fault k while RESTORE costed $R_{m,k}$ of m on fault k. The dashed line is $y = x$; the solid line is the linear regression with y dependent on x.

Precision. While it can correctly fix more bugs, RESTORE has a *precision* that is lower than other repair tools. In designing RESTORE we primarily aimed at extending the fix space that can be explored effectively by leveraging retrospective fault localization; since there is a trade off between explorable fix space and precision, the latter is not as high as in other tools that targeted it as a primary goal.

Extended fix space. RESTORE explores a larger fix space than JAID, since it can also use expressions outside method fixme in the same class to build fixes (Section 3.3.5). In all experiments when RESTORE could produce valid fixes, 68,344 candidate fixes produced during final fix generation belong to the extended fix space (and hence cannot be produced by JAID). Among them, 2,049 candidates are valid (corresponding to 52 faults); and 9 are correct (one for each of 9 faults). In all, the extended fix space enabled RESTORE to generate valid fixes for 17 more bugs than JAID, correct fixes for 9 more bugs than JAID; and correct fixes for 5 of the 8 bugs that only RESTORE can correctly fix among all tools (Table 2).

Multi-line fixes. Four of the bugs correctly fixed by RESTORE (*Closure40*, *Closure46*, *Closure115*, and *Closure128*) have programmer-written fixes in DEFECTS4J that change *multiple lines*. For example, project developers fixed the buggy method of bug *Closure128*:

```
1 static boolean isSimpleNumber(String s) {
2   int len = s.length();
3   for (int index = 0; index < len; index++) {
4     char c = s.charAt(index);
5     if (c < '0' || c > '9') return false;
6   }
7   return len > 0 && s.charAt(0) != '0';
8 }
```

by adding `if (len == 0) return false;` before line 3 *and* changing line 7 to `return len == 1 || s.charAt(0) != '0';`. RESTORE, instead, just changed line 7 to

```
if (len == 1) return true;
   else return len > 0 && s.charAt(0) != '0';
```

RESTORE's conditional return is equivalent to the programmer-written fix even though it only modifies one location. Such complex fixes demonstrate how RESTORE manages to combine bug-fixing effectiveness and competitive performance: this fix was the first valid fix in the output, generated in less than 10 minutes.

> RESTORE can correctly fix 41 faults in DEFECTS4J when allowing multiple fixes for the same bug; 19 of these faults are fixed by the first fix output by RESTORE.
>
> RESTORE trades off a lower precision for a larger fix space, which includes correct fixes for 8 faults that no other tools can fix.

4.3.2 RQ2: Performance

RQ2 assesses the *performance* of RESTORE in terms of its running time.

Total Time. RESTORE's wall-clock total running time per fault ranged between 1.5 minutes and 21 hours, with a median of 53 minutes. This means that RESTORE achieves a speedup of 3.1 (1/0.32) over JAID;Fig. 4c indicates that the major difference in favor of RESTORE is particularly marked for the *harder* faults—which generally require long running times.

Comparing with other tools in terms of running time would require to replicate their evaluations using uniform experimental settings—something we did not do in this experimental evaluation. Nevertheless, it is plausible other tools have an overall significant running time too: HDA, ACS, ssFix, Elixir, CapGen, and SimFix are all based on mining external code to learn common features of correct fixes; this process is likely time consuming—even though it would be amortized over a consequent long run of the

TABLE 5
How Retrospective Fault Localization Achieves Progress

	#	LOCALIZED	CANDIDATES	SHARPENING	PLAUSIBLE
CORRECT	41	41	23,529	2,582	511
VALID	98	75	84,989	7,348	2,762
ALL	357	107	495,359	9,854	3,377
SINGLE	74	57	61,530	5,307	2,108

Each row focuses on faults in one category: those that RESTORE can repair with a CORRECT fix; with a VALID fix; ALL faults in DEFECTS4J; and those with a SINGLE failing test. In each category, the table reports how many faults are in total (#); for how many RESTORE's fault localization can find a location suitable to build a correct fix (LOCALIZED, either because RESTORE actually built a correct fix or because the DEFECTS4J reference fix modifies that location); the number of CANDIDATES used as mutants in retrospective fault localization; how many of these candidates are SHARPENING and PLAUSIBLE.

TABLE 6
How Many Times Retrospective Fault Localization Iterates

	ITERATIONS									
	1	2	3	4	5	6	7	8	9	10
VALID	86	3	0	0	3	1	2	0	1	2
CORRECT	35	2	0	0	1	1	1	0	1	0

Among all faults in DEFECTS4J that RESTORE could repair with a VALID or a CORRECT fix, how many ITERATIONS RESTORE's feedback loop went through to sharpen fault localization.

tools—but is not present in RESTORE (or JAID). This indicates that RESTORE's performance is likely to remain competitive overall, and that retrospective fault localization can bring a performance boon. Performing more fine-grained experimental comparisons belongs to future work.

Time to Valid/Correct. Especially important for a repair tool's practical usability is the *time elapsing until* a fix appears in the *output*. All else being equal, shorter times mean that users can start inspecting fix suggestions earlier—possibly supporting a more interactive usage—so that the whole repair process can be sped up. On average, RESTORE outputs the first *valid* fix 83 minutes before JAID—a 3.4 speedup (1/0.29) according to the linear regression line; and the first *correct* fix 64 minutes before JAID—a 2.3 speedup (1/0.43). While Figs. 4d and 4e suggest that these averages summarize a behavior that varies significantly with some faults, it is clear that RESTORE's is *substantially faster* in many cases—especially with the "harder" faults that require long absolute running times. Cutting the running times in less than half on average in these cases results in speedups that often span one order of magnitude, and sometimes even two orders of magnitudes.

RESTORE's performance is the combined result of exploring a larger fix space than JAID (which takes more time) and using retrospective fault localization (which speeds up fault localization). That RESTORE finds many more correct fixes while simultaneously often drastically decreasing the running times indicates that its fault localization techniques bring a decidedly positive impact with no major downsides.

> RESTORE *is usually much faster than* JAID *even though it explores a larger fix space: 3.1 speedup in total running time; 3.4 speedup in time to the first valid fix; 2.3 speedup in time to the first correct fix.*

4.3.3 RQ3: Fault Localization

Retrospective fault localization is RESTORE's key contribution: a novel fault localization technique that naturally integrates into generate-and-validate program repair algorithms. RQ1 and RQ2 ascertained that retrospective fault localization indirectly improves program repair by supporting searching a larger fix space while simultaneously improving performance. In RQ3 we look into how retrospective fault localization is *directly* more efficient.

Checked to Valid/Correct. To this end, we follow [26]'s survey of fault localization in automated program repair and compare the number of fixes that are *checked* (generated and validated) until the first *valid* (C2V, called NFC in [26]) and the first *correct* (C2C) fix is generated. The smaller these measures the more efficiently fault localization drives the search for a valid or correct fix.

RESTORE needs to check 57 percent fewer $(1 - 0.43)$ fixes than JAID until it finds the first valid fix. RESTORE significantly improves measure C2C too: it needs to check 36 percent $(1 - 0.64)$ fewer fixes than JAID until it finds the first correct fix. Even though JAID is more efficient on some faults, Figs. 4f and 4g show that RESTORE prevails in the clear majority of cases, as well as in the harder cases that require to check many more candidate fixes (exploring a larger search space); the difference is clearly statistically significant (slope under 0.4 with 95 percent confidence, and the overlap of regression line and "no effect" line is only for small absolute values of C2V and C2C, as also reflected by the crossing ratio). These results are direct evidence of retrospective fault localization's greater precision in searching for fault causes.

Candidate fixes as mutations. Retrospective fault localization treats candidate fixes as mutants. As described in Section 3.3.3, a candidate that passes at least one previously failing test (during partial validation) increases the suspiciousness ranking of all snapshots associated with the candidate's location. Such candidate fixes sharpen fault localization, and hence we call them *sharpening* candidates. If a sharpening candidate is furthermore associated with a location where a correct fix can be built (according to the correct fixes actually produced in the experiments or in DEFECTS4J) we call it *plausible*.

Table 5 measures sharpening and plausible candidates in different categories. Only 2 percent of all candidates are sharpening; however, the percentage grows to 9 percent for faults RESTORE can build a valid fix for; and to 12 percent for faults RESTORE can build a correct fix for. These cases are those where retrospective fault localization achieved progress; in some cases (*plausible* candidates) it even led to finding program locations where a correct fix can be built. Table 5 also shows that sharpening and plausible candidates are 9 percent for faults with a single failing test case in DEFECTS4J. These can be considered "hard" faults because of the limited information about faulty behavior; retrospective fault localization can perform well even in these conditions.

Table 6 looks at RESTORE's fault localization feedback loop, which is repeated until retrospective fault localization has successfully refined the suspiciousness ranking. While some faults require as many as ten iterations, in most cases

TABLE 7
Comparison Between Restore's and Restore-full's Effectiveness and Performance

	VALID	CORRECT	TIME
Restore	98	41	122.4
Restore-full	87	27	160.6

The number of Defects4J faults with valid fixes, with correct fixes, and the average running time (in minutes) per fault in Restore compared to those in Restore-full (Restore with only full validation).

only one iteration is needed to achieve progress. This suggests that candidate fixes are often "good mutants" to perform fault localization—and they provide information that is complementary to that available with simpler spectrum-based techniques.

> *Restore's retrospective fault localization improves the efficiency of the search for correct fixes: on average, 57 percent fewer fixes need to be generated and checked until a valid one is found. The candidate fixes generated by* Restore *are effective as mutants to perform fault localization.*

4.3.4 RQ4: Robustness

RQ4 investigates whether Restore's overall effectiveness and running time are affected by changes in features and parameters of its algorithms.

Partial validation. Table 7 summarizes some key performance measures about Restore, and compares them to the same measures for Restore-full—a variant of Restore that only uses full validation as discussed in Section 4.2.2.

Restore-full is clearly less effective than Restore, as the former *misses* valid fixes for 11 faults and correct fixes for 14 faults that the latter can find. It is also slower than Restore; in fact, much slower than what suggested by the 40-minute difference per fault reported in Table 7. Remember that Restore-full is forcefully terminated after it runs for twice as long as Restore on each fault. With this cap, Restore-full could not complete its analysis for 17 of the 98 faults where Restore produces valid fixes, and it could not even finish the first round of mutation-based fault localization for 13 of them. (Restore could produce a correct fix for 11 out of these 13 faults.) Therefore, partial validation is an important ingredient to make retrospective fault localization scale up, and hence be effective.

Parameters. Table 8 shows how some key performance measures about Restore change as we individually change the value of each of four parameters N_S, N_P, N_I, and N_L.

The more snapshots N_S are used for fixing, the more valid and correct fixes Restore can generate. A closer look indicates a *monotonic* behavior: if Restore can fix a fault using s snapshots, it can also fix it using $t > s$ snapshots. Unsurprisingly, increasing N_S also increases the running time. Since the number of correctly fixed faults increases only by a few units, whereas the running time increases substantially, it seems a case of diminishing returns.

In contrast, the effects of changing the percentage N_P of snapshots used in each iteration of retrospective fault localization are very modest—both on the running time and on the number of valid and correct fixes. Increasing N_I—that is, iterating retrospective fault localization even after it has contributed to refining the ranking of suspicious locations—also has a modest effect on effectiveness but noticeably increases the running time. Overall, Restore's behavior is not much affected by how snapshots are sampled, but repeating retrospective fault localization beyond what is needed tends to decrease Restore's efficiency without any clear advantage.

TABLE 8
How Changing Parameters Affects Restore's Behavior

PARAMETER	VALUE	VALID	CORRECT	TIME
N_S	800	90	39	101.5
	*1500	98	41	127.0
	3000	103	42	180.4
N_P	5%	98	39	126.6
	*10%	98	39	127.0
	20%	99	40	133.5
N_I	*0	98	41	127.0
	2	100	41	140.4
	4	100	40	169.1
	6	100	41	181.6
N_L	2	91	33	96.8
	*5	98	41	124.5
	10	98	41	149.9

For each parameter that control Restore's algorithms, the table reports the number of Defects4J faults with valid fixes, with correct fixes, and the average running time per fault of Restore with different values of the parameter. Values marked with an asterisk (*) are defaults; in the experiments where a parameter has a non-default value, all other parameters are set to their defaults.

The default value of parameter N_L—the number of most suspicious locations used for final fix generation (Section 3.3.5)—seems to strike a good balance between effectiveness and efficiency: increasing N_L does not lead to fixing more faults, but visibly increases the running time; decreasing it reduces the running time, but also fixes fewer faults.

> *Partial validation is crucial for the efficiency of retrospective fault localization.* Restore*'s effectiveness is usually only weakly dependent on the values of internal parameters.*

4.3.5 RQ5: Generalizability

By comparing SimFix to SimFix+ (our variant of SimFix that implements retrospective fault localization) RQ5 analyzes the applicability of retrospective fault localization to tools other than Restore.

Both SimFix and SimFix+ can build *valid* fixes for the same 64 faults in Defects4J. SimFix can generate valid fixes for another 4 faults that SimFix+ cannot, and hence can fix 68 faults in total; conversely, SimFix+ can generate valid fixes for another 7 faults that SimFix cannot, and hence can fix 71 in total. In the case of the 4 faults that only SimFix can repair, SimFix's simple spectrum-based fault localization was sufficiently precise to guide the process to success (by ranking high locations that lead to suitable donor code). In contrast, the donor code leading to candidates that are useful for mutation-based fault localization (see Section 4.2.3) was ranked low; thus, SimFix+'s retrospective fault localization took multiple iterations and a long time to go through

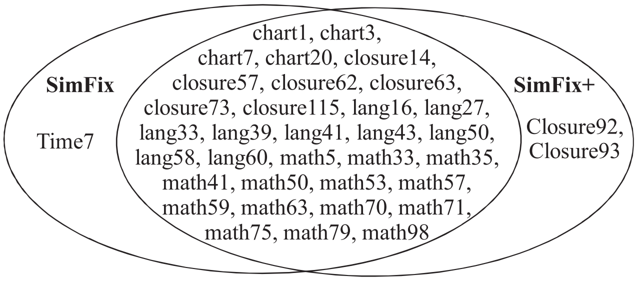
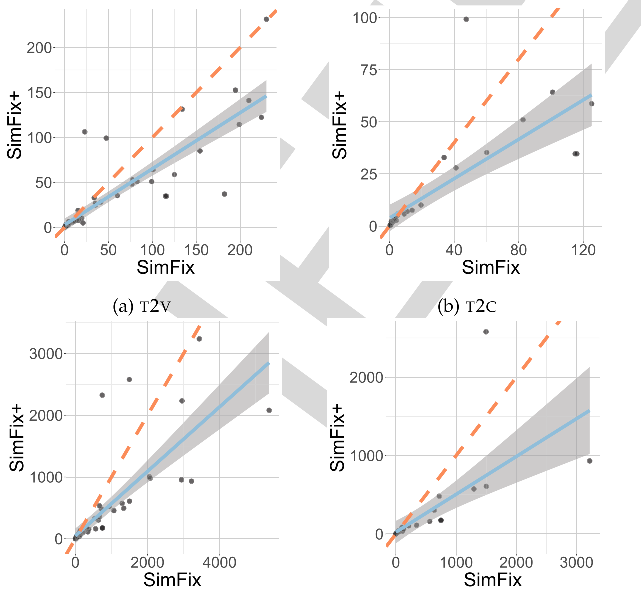


Fig. 5. Faults in DEFECTS4J bugs for which SimFix and SimFix+ can build correct fixes.

the many candidates, and ended up hitting the tool's 300-minute timeout. The cases of the 7 faults that only SimFix+ can repair are opposite: spectrum-based fault localization was imprecise, hampering the performance of SimFix, whereas mutation-based fault localization could successfully complete its analysis and sharpen the suspiciousness ranking as required by these 7 faults.

As shown in Fig. 5, both SimFix and SimFix+ can build *correct* fixes for the same 33 faults in DEFECTS4J. SimFix can generate correct fixes for 1 other fault that SimFix+ cannot, and hence can correctly fix 34 faults in total; conversely, SimFix+ can generate correct fixes for another 2 faults that SimFix cannot, and hence can correctly fix 35 in total. As in the case of the valid fixes, the differences are due to higher ranks of locations that lead to suitable donor code against lower ranks of donor code that is useful for mutation-based fault localization (or vice versa) in certain conditions.

How does SimFix+ compares to SimFix on the *large majority* of DEFECTS4J faults where both tools are successful? For the 64 DEFECTS4J faults that both can repair with at least a *valid* fix, Fig. 6a and Fig. 6c visually compare total running time (T2V)[8] and number of candidates checked (C2V) until a



(a) T2V (b) T2C

(c) C2V (d) C2C

Fig. 6. Comparison of SimFix and SimFix+ on various measures. For each measure m, a point with coordinates $x = u, y = v$ indicates that SimFix costed u on a certain fault while SimFix+ costed v on the same fault. As in Fig. 4, the dashed line is $y = x$; the solid line is the linear regression with y dependent on x.

TABLE 9
Summary Statistics of the Experiments on SimFix and SimFix+

MEASURE	$\frac{\sum \text{SimFix+}}{\sum \text{SimFix}}$	mean(SimFix − SimFix+)	slope b: 95% b_l	$\widehat{b}$	b_h	crossing χ $\widehat{\chi}$	χ_h
T2V	0.69	14	0.5	0.6	0.7	0.03	0.15
T2C	0.63	9	0.3	0.5	0.6	0.06	0.20
C2V	0.60	238	0.4	0.5	0.6	0.02	0.08
C2C	0.55	166	0.3	0.5	0.7	0.01	0.16

For each MEASURE*: the* relative cost $\frac{\sum \text{SimFix+}}{\sum \text{SimFix}}$ *of SimFix+ over SimFix; the* mean cost difference mean(SimFix − SimFix+) *between SimFix and SimFix+; the estimate* $\widehat{b}$ *of* slope b *expressing* RESTORE*'s cost as a linear function of SimFix, with 95 percent probability interval* (b_l, b_h)*; the estimate* $\widehat{\chi}$ *and upper bound* χ_h *on the* crossing ratio χ.

valid fix is found. When both SimFix and SimFix+ are successful, the latter is decidedly more *efficient*: the summary statistics of Table 9 confirm that it takes 69 percent of the running time, and needs to check 60 percent as many candidates. For the 33 DEFECTS4J faults that both tools can repair with a *correct* fix, the advantage of SimFix+ over SimFix in terms of total running time (T2C) and number of candidates checked (C2C) until a correct fix is found is also evident, as shown in Figs. 6a, 6c, and Table 9.

Unlike RESTORE—which "uses" some of the efficiency brought by retrospective fault localization to explore a larger fix space than JAID—SimFix+ has exactly the same fix space as SimFix. What we found in this section's experiments is consistent with this design choice: SimFix+ has an effectiveness that is very similar to that of SimFix (precisely, slightly better precision and recall); retrospective fault localization brings clear improvements but mostly in terms of efficiency. Trading off some of this greater efficiency to explore a larger fix space belongs to future work.

> *Retrospective fault localization implemented atop SimFix cuts down the running time of the tool by 30 percent or more, without negatively affecting bug-fixing effectiveness.*

4.4 Threats to Validity

Construct Validity. Threats to construct validity are concerned with whether the measurements taken in the evaluation realistically capture the phenomena under investigation.

An important measure is the number of *correct* fixes—fixes that are semantically equivalent to programmer-written fixes for the same fault. Since correctness is manually assessed, different programmers may disagree with the authors' classifications in some cases. To mitigate the threat, we follow the common approach [7], [23] of being conservative: fixes that do not clearly have the same behavior as the programmer-written ones are regarded as *incorrect*.

Several measures could be used to assess the performance of automated program repair tools. In our evaluation, we focus on measures that have a clear impact on *practical usability*—especially number of valid and correct fixes, and running time.

When, in Section 4.3.3, we zoom in to analyze the behavior of different aspects of RESTORE's fault localization technique,

8. Since SimFix and SimFix+ stop after one valid fix is built, total running time T and running time T2V until a valid fix is found coincide.

we use the number of fixes generated and validated until the first valid fix is found. This measure has been used by other evaluations of fault localization in program repair [26] because it assesses the overall effectiveness of fault localization in guiding the search for valid fixes—instead of measures, such as the rank of program locations, narrowly focused on the standard output of fault localization without context [27].

Our summary statistics in Table 4 follow recommended practices [17]; in particular, we used statistics that are easy to interpret, and based statistical significance on whether “an estimate is at least two standard errors away from some [...] value that would indicate no effect present” [28].

Internal Validity. Threats to internal validity are mainly concerned with factors that may affect the evaluation results but were not properly controlled for.

One obvious threat to internal validity are possible bugs in the implementation of Restore, or in the scripts we used to run our experiments. To address this threat, we reviewed our code and our experimental infrastructure between authors, to slash chances that major errors affected the soundness of our results.

Another possible threat comes from comparing Restore to tools other than Jaid based on the data of their published experimental evaluations—without *repeating* the experiments on the same system used to run Restore. This threat has only limited impact: we do not compare Restore to tools other than Jaid on measures of performance—which require a uniform runtime environment—but only on measures of effectiveness such as precision and recall—which record each tool’s bug-fixing capabilities on the same Defects4J benchmark.

External Validity. Threats to external validity are mainly concerned with whether our findings generalize—supporting broader conclusions.

Defects4J has become accepted as an effective benchmark to evaluate dynamic analysis and repair tools for Java, because of the variety and size of its curated collection of faults. At the same time, as with every benchmark, there is the lingering risk that new techniques become narrowly optimized for Defects4J without ascertaining that they do not overfit the benchmark. As future work, we plan to carry out evaluations on faults from different sources, to strengthen our claims of external validity.

Both the implementation and the evaluation of Restore are based on the Jaid repair system, and hence the fine-grained evaluation of Restore focused on how it improves over Jaid. To demonstrate that most of the ideas behind retrospective fault localization (Section 3) are applicable to other generate-and-validate automated program repair techniques, we also implemented retrospective fault localization on top of SimFix [9]—another state-of-the-art program repair technique for Java. Generalizing retrospective fault localization to work with repair techniques that are even more different—for example, based on synthesis—belongs to future work.

5 Related Work

Research in automated program repair has gained significant traction in the decade since the publication of the first works in this area [29], [30]—often taking advantage of advances in fault localization. In this section, we focus on reviewing the approaches that have more directly influenced the design of Restore. Other publications provide comprehensive summaries of fault localization [31] and automated program repair [32], [33] techniques.

5.1 Fault Localization

The goal of fault localization is finding positions in the source code of a faulty program that are responsible for the fault. The concrete output of a fault localization technique is a list of statements, branches, or program states ranked according to their likelihood of being implicated with a fault. By focusing their attention on specific parts of a faulty program, such lists should help programmers debugging and patching. While this information may not be enough for human programmers [27], it is a fundamental ingredient of *automated* program repair. Thus, research in fault localization has seen a resurgence as part of an effort to improve automated repair.

Spectrum-based fault localization techniques [34], [35] are among the most extensively studied. The basic idea of spectrum-based fault localization is to use coverage information from tests to infer suspiciousness values of program entities (statements, branches, or states): for example, a statement executed mostly by failing tests is more suspicious than one executed mostly by passing tests.

Several automated program repair techniques use spectrum-based fault localization algorithms [7], [30], [36], [37], [38], [39]. Generating a correct fix, however, typically requires more information than the suspiciousness ranking provided by spectrum-based techniques: an empirical evaluation of 15 popular spectrum-based fault localization techniques [26] found that the typical evaluation criteria used in fault-localization research (namely, the suspiciousness ranking) are not good predictors of whether a technique will perform well in automated program repair. This observation buttresses our suggestion that fault localization should be *co-designed* with automated program repair to perform better—as we did with retrospective fault localization.

Fault localization needs sources of additional information to be more accurate. One effective idea—pioneered by delta debugging [40]—is to *modify* a program and observe how small local modifications affect its behavior in passing vs. failing runs. More recently, ideas from mutation testing [41] and delta-debugging have been combined to perform *mutation-based* fault localization: randomly mutate a faulty program, and assess whether the mutation changes the behavior on passing or failing tests.

Metallaxis [6] and MUSE [5], [42] are two representative mutation-based fault localization techniques. Experiments with these tools indicate that mutation-based fault localization often outperforms spectrum-based fault localization in different conditions [5], [6]. In our work, we used a variant of the Metallaxis algorithm, because it tends to perform better than MUSE with tasks similar to those we need for automated program repair. The main downside of mutation-based fault localization is that it can be a performance hog, because it requires to rerun tests on a large amount of mutants. Thus, a key idea of our retrospective fault localization is to reuse, as much as possible, validation results (which have to be performed anyway for program repair) to perform mutation-based analysis.

In retrospective fault localization, a simple fault-localization process bootstraps a feedback loop that implements a more accurate mutation-based fault localization. Restore currently uses a spectrum-based technique for the bootstrap phase (see Section 3.2.2); however, other fault localization techniques—such as those based on statistical analysis [43], [44], machine learning [45], [46], or deep learning [47]—could be used instead. Even techniques that are not designed specifically for fault localization may be used, as long as they produce a ranked list of suspicious program entities. For example, MintHint [48] performs a correlation analysis to identify expressions that should be changed to fix faults. The expressions, or more generally their program locations, could thus be treated as suspicious entities for the purpose of initiating fault localization.

5.2 Automated Program Repair

Generate-and-Validate (G&V) remains the most widespread approach to automated program repair: given a faulty program and a group of passing and failing tests, generate fix candidates by heuristically searching a program space; then, check the validity of candidates by rerunning all available tests. GenProg [30], [49] pioneered G&V repair by using genetic programming to mutate a faulty program and generate fix candidates. RERepair [50] works similarly to GenProg but uses random search instead of genetic programming. AE [51] enumerates variants systematically, and uses simple semantic checks to reduce the number of equivalent fix candidates that have to be validated. Par [38] uses patterns modeled after existing programmer-written fixes to guide the search toward generating fixes that are easier for programmers to understand.

This first generation of G&V tools is capable of working on real-world bugs, but has the tendency to *overfit* the input tests [3]—thus generating many fixes that pass validation but are not actually correct [2]. A newer generation of tools addressed this shortcoming by supplying G&V program repair with *additional information*, often coming from mining human-written fixes. AutoFix [39] uses contracts (assertions such as pre- and postconditions) to improve the accuracy of fault localization. SPR [52] generates candidate fixes according to a set of predefined transformation functions; Prophet [53] implements a probabilistic model, learned by mining human-written patches, on top of SPR to direct the search towards fixes with a higher chance of being correct. HDA [22] performs a stochastic search similar to genetic programming, and uses heuristics mined from fix histories available in public bug repositories to guide the search toward generating correct fixes. ACS [19] builds precise changes of conditional predicates, based on a combination of dependency analysis and mining API documentations. Genesis [54] learns templates for code transformations from human patches, and instantiates the templates to generate new fixes. ssFix [25] matches contextual information at the fixing location to a database of human-written fixes, and uses this to drive fix generation. Jaid [7] uses rich state abstractions in fault localization to generate correct repairs for a variety of bugs. Elixir [21] specializes in repairing buggy method invocations, using machine-learned models to prioritize the most effective repairs. SimFix [9] 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 [20] improves the effectiveness of expression-level fix generation by leveraging fault context information so that fixes more likely to be correct are generated first. SketchFix [24] expresses program repair as a sketching problem [55] with "holes" in suspicious statements, and uses synthesis to fill in the holes with plausible replacements. Restore and SketchFix both work to better integrate phases that are normally separate in automated repair—fault localization and fix validation in Restore, and fix generation and fix validation in SketchFix.

Most of these tools are quite effective at generating correct fixes for real bugs; several of them do so by mining *additional information*. Further improvements in G&V repair hinge on the capability of improving the precision of fault localization. A promising option is using mutation-based fault localization, which was recently investigated [56] on data from the BugZoo[9] repair benchmarks. [56] found no significant improvement on the overall repair performance—supposedly because the single-edit mutations used in the study may be too simple to reveal substantial differences between programs variants.

In our retrospective fault localization, we combine mutation testing with a G&V technique that can generate complex "higher-order" program mutants, and tightly integrate fault localization and fix generation. This way, Restore benefits from the additional accuracy of mutation-based fault localization without incurring the major overhead typical of mutation testing.

Test Selection and Prioritization has been studied in the context of G&V automated program repair to improve the efficiency of fix evaluation. For example, techniques based on genetic programming—such as GenProg [30] and PAR [38]—can become very computationally expensive if they evaluate all program mutations on all available tests. To improve this situation, one could use all the failing tests but only a small sample of the passing tests—selected randomly [57] or using an adaptive test suite reduction strategy [58]. Another approach is the FRTP technique [50], [59], which gives higher priority to a test the more fixes it has invalidated in previous iterations. Restore currently uses a very simple test selection strategy for partial validation (Section 3.3.2) consisting in just running the originally failing tests. This was quite economical, yet effective, in the experiments with Defects4J, but cannot replace a full validation step. To achieve further improvements we will consider more sophisticated test selection strategies in future work.

Correct-by-Construction program repair techniques [37], [60], [61], [62], [63] express the repair problem as a constraint satisfaction problem, and then use constraint solver to build fixes that satisfy those constraints. Relying on static instead of dynamic analysis makes correct-by-construction techniques generally *faster* than G&V ones, and is particularly effective when looking for fixes with a restricted, simple form.

6 Conclusions

We presented *retrospective fault localization*: a novel fault localization technique that integrates into the standard generate-and-validate process followed by numerous automated

9. https://github.com/squaresLab/BugZoo

program repair techniques. By executing a form of mutation-based testing using byproducts of automated repair, retrospective fault localization delivers accurate fault localization information while curtailing the otherwise demanding costs of running mutation-based testing.

Our experiments compared Restore—implementing retrospective fault localization—with 13 other state-of-the-art Java program repair tools—including Jaid, upon which Restore's implementation is built. They showed that Restore is a state-of-the-art program repair tool that can search a large fix space—correctly fixing 41 faults from the Defects4J benchmark, 8 that no other tool can fix—with drastically improved performance (speedup over 3, and candidates that have to be checked cut in half).

Retrospective fault localization is a sufficiently general technique that it could be integrated, possibly with some changes, into other generate-and-validate program repair systems. To support this claim, we implemented it atop SimFix [9]—another recent automated program repair tool for Java—and showed it brings similar benefits in terms of improved efficiency. As part of future work, we plan to combine retrospective fault localization with other recent advances in fault localization—thus furthering the exciting progress of automated program repair research.

Acknowledgments

Tongtong Xu, Tian Zhang, and Minxue Pan were supported by the National Natural Science Foundation of China (No. 61690204 and 61972193) and the Fundamental Research Funds for the Central Universities, China (No. 14380020 and 14380022). Liushan Chen and Yu Pei were supported by the Hong Kong RGC General Research Fund (GRF) under grant PolyU 152703/16E and PolyU 152002/18E and by the Hong Kong Polytechnic University under internal fund 1-ZVJ1 and G-YBXU. Carlo A. Furia was partially supported by the Swiss National Science Foundation (SNF) under grant Hi-Fi 200021-182060.

References

[1] M. Zhivich and R. K. Cunningham, "The real cost of software errors," *IEEE Security Privacy*, vol. 7, no. 2, pp. 87–90, Mar./Apr. 2009.

[2] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.

[3] E. K. Smith, E. T. Barr, C. L. Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 532–543.

[4] 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 *Proc. 36th. Softw. Eng.*, 2014, pp. 234–242.

[5] S. Moon, Y. Kim, M. Kim, and S. Yoo, "Ask the mutants: Mutating faulty programs for fault localization," in *Proc. IEEE Int. Conf. Softw. Testing, Verification Valid.: IEEE Computer Society*, 2014, pp. 153–162.

[6] M. Papadakis and Y. Le Traon, "Metallaxis-FL: Mutation-based fault localization," *Softw. Testing Verification Rel.*, vol. 25, no. 5–7, pp. 605–628, Aug. 2015.

[7] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts," in *Proc. 32nd IEEE/. Autom. Softw. Eng.*, 2017, pp. 637–647.

[8] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440, [Online]. Available: http://defects4j.org

[9] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.

[10] T. Durieux, B. Danglot, Z. Yu, M. Martinez, S. Urli, and M. Monperrus, "The Patches of the Nopol Automatic Repair System on the Bugs of Defects4J version 1.1.0," Université Lille 1 - Sciences et Technologies, Research Report hal-01480084, 2017.

[11] R. Abreu, P. Zoeteweij, and A. J. C. V. Gemund, "An evaluation of similarity coefficients for software fault localization," in *Proc. 12th Pacific Rim Int. Symp. Dependable Comput.*, 2006, pp. 39–46.

[12] W. E. Wong, V. Debroy, and B. Choi, "A family of code coverage-based heuristics for effective fault localization," *J. Syst. Softw.*, vol. 83, no. 2, pp. 188–208, Feb. 2010.

[13] L. Chen, Y. Pei, and C. A. Furia, "Contract-based program repair without the contracts: An extended study," *IEEE Trans. Softw. Eng.* Jan. 2020, to be published, http://dx.doi.org/10.1109/TSE.2020.2970009

[14] J. A. Jones and M. J. Harrold, "Empirical evaluation of the Tarantula automatic fault-localization technique," in *Proc. 20th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2005, pp. 273–282.

[15] M. Renieris and S. P. Reiss, "Fault localization with nearest neighbor queries," in *Proc. 18th IEEE Int. Conf. Autom. Softw. Eng.*, 2003, pp. 30–39.

[16] D. Critchlow, *Metric Methods for Analyzing Partially Ranked Data*. ser. Lecture Notes in Statistics. Springer, New York, 2012.

[17] T. Hoefler and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proc. Int. Conf. High Perform. Comput. Netw.*, 2015, pp. 73:1–73:12.

[18] R. McElreath, *Statistical Rethinking*. London, UK: Chapman & Hall/CRC, 2015.

[19] Y. Xiong *et al.*, "Precise condition synthesis for program repair," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 416–426.

[20] M. Wen, J. Chen, R. Wu, D. Hao, and S. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1–11.

[21] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object oriented program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 648–659.

[22] X. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 213–224.

[23] 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," *Empir. Softw. Eng.*, vol. 22, no. 4, pp. 1936–1964, 2017.

[24] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 12–23.

[25] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 660–670.

[26] Y. Qi, X. Mao, Y. Lei, and C. Wang, "Using automated program repair for evaluating the effectiveness of fault localization techniques," in *Proc. Int. Symp. Softw. Testing Anal.*, 2013, pp. 191–201.

[27] C. Parnin and A. Orso, "Are automated debugging techniques actually helping programmers?" in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 199–209.

[28] A. Gelman and D. Weakliem, "Of beauty, sex and power," *Amer. Scientist*, vol. 97, pp. 310–316, 2009.

[29] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congr. Evol. Comput.*, 2008, pp. 162–168.

[30] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 364–374.

[31] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

[32] M. Monperrus, "Automatic software repair: A bibliography," *ACM Comput. Surv.*, vol. 51, pp. 1–24, 2017.

[33] L. Gazzola, D. Micucci, and L. Mariani, "Automatic software repair: A survey," *IEEE Trans. Softw. Eng.*, vol. 45, no. 1, pp. 34–67, Jan. 2019.

[34] L. Naish, H. J. Lee, and K. Ramamohanarao, "A model for spectra-based software diagnosis," *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 3, pp. 11:1–11:32, Aug. 2011.

[35] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Acad. Ind. Conf. Practice Res. Techn. MUTATION*, 2007, pp. 89–98.

[36] V. Debroy and W. E. Wong, "Using mutation to automatically suggest fixes for faulty programs," in *Proc. 3rd Int. Conf. Softw. Testing Verif. Valid.*, 2010, pp. 65–74.

[37] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. Int. Softw. Eng.*, 2013, pp. 772–781.

[38] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 802–811.

[39] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Trans. Softw. Eng.*, vol. 40, no. 5, pp. 427–449, May 2014.

[40] A. Zeller, *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2005.

[41] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[42] S. Hong *et al.*, "Mutation-based fault localization for real-world multilingual programs," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 464–475.

[43] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 15–26.

[44] Chao Liu, Long Fei, Xifeng Yan, Jiawei Han, and S. P. Midkiff, "Statistical debugging: A hypothesis testing-based approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 10, pp. 831–848, Oct. 2006.

[45] L. C. Briand, Y. Labiche, and X. Liu, "Using machine learning to support debugging with tarantula," in *Proc. 18th IEEE Int. Symp. Softw. Rel.*, 2007, pp. 137–146.

[46] W. E. Wong and Y. Qi, "Bp neural network-based effective fault localization," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 19, no. 4, pp. 573–597, 2009.

[47] R. Gupta, A. Kanade, and S. Shevade, "Deep learning for bug-localization in student programs," *CoRR*, abs/1905.12454, 2019.

[48] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "Minthint: Automated synthesis of repair hints," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 266–276.

[49] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proc. 34th Int. Conf. on Softw. Eng.*, 2012, pp. 3–13.

[50] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 254–265.

[51] W. Weimer, Z. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *Proc. IEEE/ACM 28th Int. Autom. Softw. Eng.*, 2013, pp. 356–366.

[52] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. 10th Joint Meeting Foundations Soft. Eng.*, 2015, pp. 166–178.

[53] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2016, pp. 298–312.

[54] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 727–739.

[55] A. Solar-Lezama, "Program sketching," *Softw. Tools Technol. Transfer*, vol. 15, no. 5–6, pp. 475–495, 2013.

[56] C. S. Timperley, S. Stepney, and C. Le Goues, "An investigation into the use of mutation analysis for automated program repair," in *Proc. Int. Symp. Search Based Softw. Eng.*, 2017, pp. 99–114.

[57] E. Fast, C. Le Goues, S. Forrest, and W. Weimer, "Designing better fitness functions for automated program repair," in *Proc. Genetic Evol. Comput. Conf.*, 2010, pp. 965–972.

[58] K. R. Walcott, M. L. Soffa, G. M. Kapfhammer, and R. S. Roos, "Timeaware test suite prioritization," in *Proc. Int. Symp. Softw. Testing Anal.*, 2006, pp. 1–12.

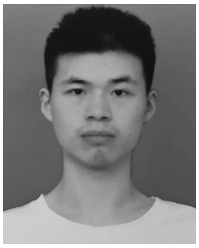
[59] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 180–189.

[60] S. Mechtaev, J. Yi, and A. Roychoudhury, "DirectFix: Looking for simple program repairs," in *Proc. 37th. Softw. Eng.*, 2015, pp. 448–458.

[61] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 691–701.

[62] J. Xuan *et al.*, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 34–55, Jan. 2017.

[63] X.-B. D. Le, D.-H. Chu, D. Lo, C. L. Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 593–604.

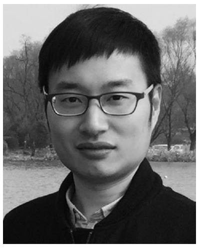


Tongtong Xu received the BA degree in physics from Nanjing University, China, he is currently working toward the PhD degree in the Department of Computer Science and Technology, Nanjing University, China. His main research interests lie in automatic program repair and software testing.

Liushan Chen received the BA degree in information management and information systems from the University of International Business and Economics, China, and the MSc degree in information technology from the HongKong Polytechnic University. She is currently working toward the PhD degree in the same University. Her main research interests lie in automatic program repair and software fault localization.

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.

Tian Zhang received the PhD degree in Nanjing University. He is an associate professor with the 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.



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

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

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.