ITER: Iterative Neural Repair for Multi-Location Patches

He Ye
hey@cs.cmu.edu
Carnegie Mellon University
United States

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

ABSTRACT

Automated program repair (APR) has achieved promising results, especially using neural networks. Yet, the overwhelming majority of patches produced by APR tools are confined to one single location. When looking at the patches produced with neural repair, most of them fail to compile, while a few uncompilable ones go in the right direction. In both cases, the fundamental problem is to ignore the potential of partial patches. In this paper, we propose an iterative program repair paradigm called ITER founded on the concept of improving partial patches until they become plausible and correct. First, ITER iteratively improves partial single-location patches by fixing compilation errors and further refining the previously generated code. Second, ITER iteratively improves partial patches to construct multi-location patches, with fault localization re-execution. ITER is implemented for Java based on battle-proven deep neural networks and code representation. ITER is evaluated on 476 bugs from 10 open-source projects in Defects4J 2.0. ITER succeeds in repairing 15.5% of them, including 9 uniquely repaired multi-location bugs.

ACM Reference Format:
He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE '24), April 14–20, 2024, Lisbon, Portugal.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3597503.3623337

1 INTRODUCTION

Automated program repair (APR) [12, 14, 33] is a promising idea to incorporate different concepts from automated software engineering together in order to repair software bugs automatically, eg. fault localization, code transformation and correctness verification. Test-suite-based program repair is one of the well-studied APR paradigms [22, 30, 58], in which test-suites are considered as program correctness specifications. Given a program and its test suite with at least one failing test, APR approaches generate patches to make the test suite pass, such patches are known as `plausible patches`. Finding plausible patches can be made with techniques involving search (e.g., GenProg [47], Elixir [38] and Hercules [39]), semantic analysis (e.g., SemFix [34], DirectFix [31] and Angelix [32]), and deep learning (e.g., SequenceR [8], SelfAPR [59] and Modit [6]).

ICSE '24, April 14–20, 2024, Lisbon, Portugal

ACM ISBN 979-8-4007-0217-4/24/04.
https://doi.org/10.1145/3597503.3623337

```
+ if  ( r != null ) {
4494  Collection c = r.getAnnotations();
4495  Iterator i = c.iterator();
Compilation Error:  '}' is expected
```

Figure 1: A patch for Chart-4 in Defects4J generated by AlphaRepair, Recoder, SelfAPR, RewardRepair, Cure, and CoCoNut. This compilation error patch could be further improved into a correct patch in a subsequent iteration.

We note that all the aforementioned APR approaches repair bugs with a single repair attempt, a conceptual framework called in this paper the *"one-step repair paradigm"*. In this paper, a "step" is defined as a complete repair attempt consisting of fault localization, patch generation, and patch validation, ultimately resulting in a decision to keep or discard such a patch. The term "one-step repair paradigm" signifies that the life cycle of the generated patch involves only one opportunity to be evaluated for keeping or discarding. If there exists a patch to make all test pass, then that plausible patch is kept, otherwise, all intermediate patches are discarded. Different from one-step repair, multi-step repair that we propose in this paper consists of refining previous repair results.

Problem 1: One-step repair is inefficient for single-location bugs. The one-step repair paradigm ignores the potential of partial patches that could lead to correct patches with only a small adjustment. To illustrate this point, Figure 1 shows a patch that is commonly generated by six different repair techniques: AlphaRepair [51], Recoder [62], SelfAPR [59], RewardRepair [60], Cure [18] and CoCoNut [28]. This patch applies a correct fix pattern on a correct buggy location. Unfortunately, all these techniques miss adding a '}' to get a valid AST structure and complete the patch. This patch is partial: it goes in the right direction but it is not finished. Yet, it is a good working solution that can be further improved into a correct patch. However, all existing APR approaches simply discard partial patches.

Problem 2: One-step repair is fundamentally limited for repairing multi-location bugs. A multi-location patch is a patch that requires edits at multiple, non-contiguous locations in a program. It is known that generating multi-location patches is especially challenging [49]. There are only a few works target to repair multi-location bugs, with mixed success [24, 39, 49]. Repairing multi-location bugs needs to address two key challenges. The first challenge is judging partial correctness in order to decide whether a work-in-progress patch should be discarded or further improved. The second challenge is to correctly locate all buggy locations: most APR is founded on spectrum-based fault localization [1] which only outputs single lines in isolation, and not tuples of lines that should be changed together.

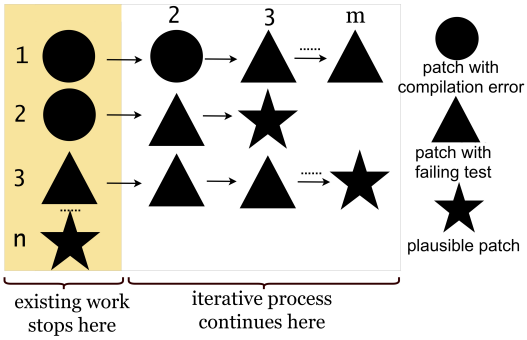


Figure 2: Sketch of iterative repair: ITER chains compilation error repair and functional error repair with a single repair model.

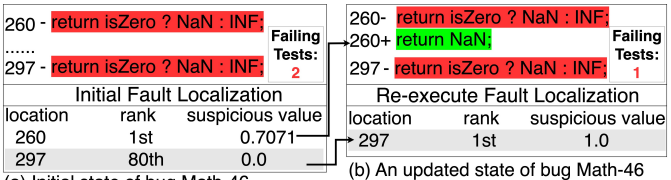


Figure 3: Math-46: iterative execution of fault localization enables to focus on the next location to be repaired.

To address the above two problems, we devise a novel program repair approach ITER, whose primary aim is to improve partial patches until they become correct. The novelty of ITER is to conduct an iterative patch refinement process, coupled with iterative fault localization based on the partial patches obtained in previous repair steps. This patch refinement process leads to an updated test execution result, consequently improving the accuracy of fault localization. To our knowledge, the iterative repair process has been demonstrated by only a few works such as GenProg [47], DeepFix [15] and an early work from Arcuri and Yao [3]. ITER builds upon this iteration idea by introducing two unique components: 1) iterative patch refinement to keep and improve the intermediate patches while previous works choose to discard them and 2) iterative fault localization with the partial patches to obtain an updated list of suspicious locations.

Solution 1: Iteration over partial patches Figure 2 shows a sketch of iterative repair. Existing work generates all candidate patches independently and without interaction, as highlighted in the yellow bar area. One-step repair would completely discard all non-plausible patches. Departing away from one-step repair, ITER considers partial patches as the basis for further improvement. For example, in the second row of Figure 2 , ITER takes the first patch compilation error to further improve it, to turn it into a compilable patch with a test failure and then into a plausible patch that passes all tests. ITER chains compilation error repair and functional error repair in arbitrary order, as shown in the different rows of Figure 2. The iteration over partial patches serves in both training and inference stages. The iterative training process not only enables the data augmentation with diverse training samples, but also enables the neural network to be trained with its own output, which subsequently becomes the new input in the next iteration.

It is worth noting that prior work by Bhatia et al. [4] also explore the concept of chaining compilation error repair and functional error repair. However, their approach utilizes two separate models, whereas ITER accomplishes this chaining using a single repair model. Additionally, ITER allows the flexibility of arbitrary orders for conducting compilation error repair and functional error repair, whereas the prior work strictly follows a sequential order of first addressing compilation errors and then fixing functional errors.

Solution 2: Iteration over locations to be fixed ITER defines a way to iterate over fault localization results in a novel manner. The core idea is to re-execute fault localization if a partial patch is found that reduces the number of failing tests. This results in an improved ranked list of suspicious buggy locations, updated per the behavioral changes from the partial patch. This is different from all existing repair approaches that only execute fault localization once. Our experiments give clear evidence that this is key to pushing the state-of-the-art of repairing multi-location bugs.

Let us look at Figure 3 to give a better intuition of this iteration idea. Figure 3 shows a bug from `Math-46` from Defects4J, which requires edits at two locations `line 260` and `line 297`. A partial patch can be generated at `line 260` reducing the failing tests. Re-executing fault localization largely changes the suspicious lines and re-ranks the next buggy location `line 297` from the previous 80th place to the 1st place. This is explained as many suspicious locations are eliminated thanks to the partial patch being found, and the new ranked list only contains the most relevant suspicious locations.

We implement ITER on top of state-of-the-art neural program repair. This is very natural for two reasons. First, our insight is that the same neural network can be good at fixing both compilation errors and functional errors, hence enabling the chained patch improvements of Figure 1. Second, one can use self-supervised training sample generation to train the network on a sequence of patches being improved from one to another and leverage the power of advanced input representations with error messages [59].

Our vision of iterative repair is multifaceted. At training time, ITER proposes an iterative loop by collecting the output of the patch generator to augment the training samples. This step largely increases the number of training samples: 2 726 077 new training samples, 5X more than the initial training dataset. More importantly, ITER learns from its own mistakes, the failed repaired attempts in the previous optimization round.

At inference time, ITER does what we call 'iterative inference", chaining fault localization, patch generation, and patch validation into one single repair loop. Patch validation executes tentative patches and provides the essential signals of failing tests and error diagnostics. Fault localization provides a ranked list of suspicious locations for the patch generator, based on the current partial patch. The patch generator produces candidate patches to repair either the initial bug or the current partial patch. This happens in a loop until a plausible patch is found or a maximum budget is reached.

We evaluate ITER on 476 real-world bugs from 10 open-source projects collected in Defects4J [19] benchmark and compare ITER with the recent relevant techniques. ITER correctly repairs 74 of 476 bugs in the evaluation dataset. Our experimental result shows ITER is able to improve partial patches into correct patches. As a result, ITER repairs 9 unique multi-location bugs compared to

related work Hercules [39], VarFix [49] and DEAR [24], thanks to the effectiveness of iterative inference.

To sum up, we make the following contributions:

- We devise "iterative repair", a novel program repair paradigm for partial patch improvement. To our knowledge, this is the first repair architecture that chains repairs of compilation errors and test errors with one single repair model, refining partial patches until they become plausible.
- We implement ITER, realizing the vision of iterative repair for Java, based on state-of-the-art deep learning models and input/output representations.
- We perform an original series of experiments and show that ITER repairs 74 bugs from 10 open-source projects in Defects4J. Per our goal, we demonstrate that ITER is able to repair multi-location bugs, better than the state-of-the-art, with 9 patches that have never been synthesized.
- We make all our code and data publicly available at https://anonymous.4open.science/r/IteRepair-3A26 and we have set up an online demonstration at https://www.iterativerepair.tech .

2 APPROACH

Figure 4 gives an overview of ITER. ITER has two main novel concepts: iterative training and iterative inference. The upper part illustrates the iterative training and the lower part shows the iterative inference. As highlighted in the arrows in blue, both the training and inference consider the output from the patch generator as the basis of the new bugs and further improve them to seek an iteration loop.

At training time, the patch generator at a given iteration 1, .., n is used to generate self-augment samples in an iterative loop. The number of iterations in the loop is configurable by end users and this iteration ends when the max iteration configuration is achieved. There are two insights behind iterative training. First, this self-augmentation increases the number of training samples beyond the initial training dataset. Second, such an iterative loop enables the patch generator to learn to repair its own mistakes (i.e., its output).

At inference time, the iterative vision of ITER is as follows: all candidate patches generated by the patch generator that are not yet plausible are further improved until they compile and pass the test specification. ITER conducts the following three steps: 1) fault localization, 2) patch generation, and 3) patch validation. For each iteration, ITER keeps track of the state of the generated patch with three pieces of information: 1) the new diagnostic (compiler error or failed test error message). 2) the new buggy locations that are identified for uncompilable patches (from a compiler). 3) the new suspicious locations that are identified with fault localization (from fault localization).

ITER is novel in the following: First, ITER conducts the iteration process in both training and inference processes. ITER considers improving two types of patches with a) compilation errors or b) function errors until they compile and pass test cases. This is in contrast to all the existing step repairs in the literature [17, 49–51, 62] that only conduct a one-step repair. Second, ITER conducts the iteration loop by integrating three repair components into one iterative process: fault localization, patch generation, and patch validation. This is different from existing learning-based repair techniques that consider fault localization and patch validation as separate pre-process and post-process tasks from patch generation. This is different from all the prior work that only executes fault localization once.

2.1 Iterative Training

ITER is built on the concept of self-supervised learning which consists of automatically producing labeled data and bug diagnostics for unlabeled programs. In the context of repair, it has been proven useful for grounding training on the execution of the buggy and patched programs [2, 56, 59].

In ITER, self-supervised learning is used to provide two kinds of training samples: compilation bugs and functional bugs as well as the corresponding fixes. This is the key to iteratively improving the different patches until they compile and pass all tests, potentially alternating between compilation errors and functional errors. Second, self-supervised learning with execution enables ITER to include diagnostics of buggy programs to be repaired, which is important guiding information to drive the iterative repair process [59].

Input and Execution For training, ITER takes an input of correct programs and perturbs them into buggy programs using a perturbation model [2, 56, 59]. To determine the bugginess of the perturbation-based samples, per the prior work [59], ITER executes perturbation-based samples with the compiler and the test suite, and ITER only keeps those samples that cause compilable errors or test failures. All buggy samples are fed to a neural patch generator whose goal is to learn to transform a buggy program (b) into the fix program (f).

Iteration At a given point in time, ITER's neural network outputs different variants of the bug $\mathcal{V}$ (i.e., candidate patches) for a given buggy sample based on a configurable beam search width k. The usage of those training outputs is two-fold. First, they are used to compute a loss value and to optimize the neural network with back-propagation accordingly. Second, they are used as new training samples (as shown in the blue arrow), as a data augmentation technique. These self-augment training samples generated from the neural networks are able to increase the number of training samples beyond the perturbation model that is used. Importantly, the neural network is trained to repair its own mistakes and this is intuitively useful for ITER to repair real-world bugs at inference time.

Iterative Training Sample Selection Algorithm 1 shows the generation and selection of new training samples. ITER takes as input the initial training samples $\mathcal{I}$ consisting of pairs of buggy and ground truth code (b_i, f_i) and a neural patch generator $\mathcal{G}$. ITER is configured be a beam size width as k and a max iteration MAX. As shown in line 7, for each buggy sample b_i, the patch generator outputs a set of variants $\mathcal{V}$ of v_i based on beam size width k. Not all the v_i are considered valid augmented training samples to be added to $\mathcal{S}$. We discard those that identical to the buggy program b_i and those that are identical to the ground truth fix f_i (in line 10 and 11). All the iterative training samples $\mathcal{S}$ are combined with initial training input $\mathcal{I}$ and they form the new training dataset for the next iteration.

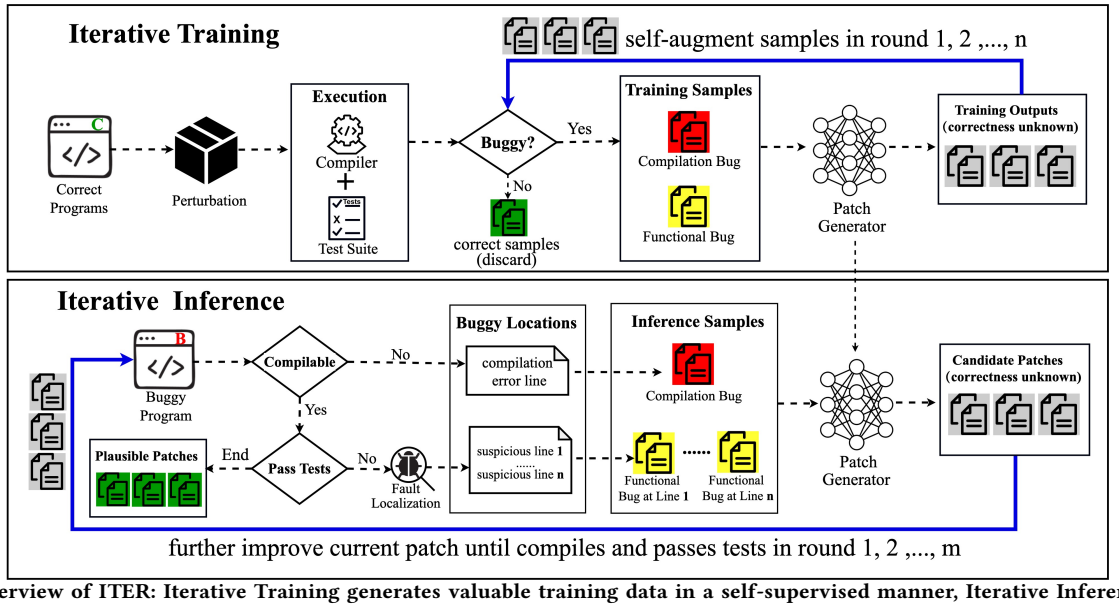


Figure 4: Overview of ITER: Iterative Training generates valuable training data in a self-supervised manner, Iterative Inference chains compilation error repair and functional error repair to improve partial patches until they become plausible.

Algorithm 1 Iterative Training

1: **Input** initial training samples $\mathcal{I}$: (b_1,f_1), (b_2,f_2),...,(b_n,f_n), patch generator: $\mathcal{G}$
2: **Config:** beam search width: k, max iteration: MAX
3: **Output:** iterative training samples $\mathcal{S} \leftarrow \emptyset$
4: $\mathcal{S} \leftarrow \mathcal{I}$
5: count $\leftarrow$ 0
6: **while** count < MAX **do**
7: **for** (b_i, f_i) in $\mathcal{S}$ **do**
8: $\mathcal{V}: v_0 \ldots v_k \leftarrow \mathcal{G}(b_i, k)$
9: **for** v_i in $\mathcal{V}$ **do**
10: **if** v_i ! = b_i **and** v_i ! = f_i **then**
11: $\mathcal{S} \leftarrow \mathcal{S} \cup (v_i, f_i)$
12: **end if**
13: **end for**
14: **end for**
15: count++
16: **end while**

2.2 Iterative Inference

The core idea of ITER is to iteratively improve the current patches until they compile and pass tests. ITER considers a partial patch as the basis for the next improvement. ITER integrates fault localization, patch generation and patch validation in each iteration loop rather than treating them as three separate tasks as related work [18, 28, 50, 51, 59, 60, 62]. Such integration brings two benefits. First, patch validation selects partial patches that are able to improve the effectiveness of fault localization (as demonstrated in Figure 3). Second and accordingly, a better fault localization further improves the effectiveness of ITER in generating correct patches.

Input and Execution At inference time, ITER takes as input a buggy program and executes it against the accompanying compiler and test suite to determine whether it is a compilation bug or a functional bug, and to collect the diagnostic. If the bug contains compilation errors, ITER takes the explicit compilation error line and diagnostic based on the compiler. If the bug contains functional errors, ITER executes fault localization to produce a ranked list of suspicious locations, each of which is considered a separate bug to be repaired.

Iteration ITER outputs different candidate patches for a given bug based on the configurable beam size k. All the candidate patches are tested against the compiler and test suite. If those patches are buggy (uncompilable or failing tests), they are taken again as input in order to be further improved, this is the core idea of iterative inference. After one or more iterations, if one patch is plausible, i.e., it passes all test cases, it goes to the plausible patch pool and it will not be further improved.

The iteration is signaled by the number of failing test cases. There are two options to be considered during iteration. Option 1: when a partial patch is found to reduce the number of failing tests, then this patch is kept and the fault localization is re-executed based on this partial patch. Option 2: when no partial patch reduces the number of failing tests, ITER continuously improves the buggy location in its last iteration.

Iterative Algorithm Algorithm 2 explains the iterative inference process as pseudo-code. The key recursive repair function is called in `line 29` by providing the newly generated patch `p` based on the previously patched program `b'`. The patch `b'` is validated against compiler C and test suite $\mathcal{T}$ to obtain the current fail tests $\mathcal{N}_{b'}$, note that $\mathcal{N}_{b'}$ is only obtained when `b'` is compilable (line 7). ITER takes the failing test number as the signal to identify partial patches to be kept and re-execute the fault localization $\mathcal{F}$. As shown in line `line 22` and `line 23`, fault localization $\mathcal{F}$ is re-executed when $\mathcal{N}_{b'}$ is fewer than initial failing tests $\mathcal{N}_b$. ITER is ended under two conditions. First, ITER is ended when a plausible patch `p` is found (`line 17`), which forms the pair of data $< p, \mathcal{L} >$. Second, ITER is ended when iteration i achieves a maximum configuration (`line 20`).

In theory, the maximum number of total candidate patches P on one location $\mathcal{L}$ depends on the beam search width k (i.e. the number of candidate patches generated per bug) and the number of iteration i. We compute the candidate patch number P in Equation 1.

$$P = k^i \tag{1}$$

Algorithm 2 Iterative Inference

Input: initial buggy program b, fault localization $\mathcal{F}$, patch generator $\mathcal{G}$, compiler C, test suite $\mathcal{T}$, initial failing tests number $\mathcal{N}_b$
Config: beam search width k, max iteration MAX
Output: plausible patches $plausible$
Init: $plausible \leftarrow \emptyset$,

MAIN()
  $p \leftarrow \emptyset$ //initial patch
  $i \leftarrow 0$ //iteration
  $\mathcal{L} \leftarrow \emptyset$ //buggy location
  ITER (b, p, k, i, $\mathcal{F}$, $\mathcal{G}$, C, $\mathcal{T}$, $\mathcal{L}$, $\mathcal{N}_b$)

ITER (b, p, k, i, $\mathcal{F}$, $\mathcal{G}$, C, $\mathcal{T}$, $\mathcal{L}$, $\mathcal{N}_b$)
  b' $\leftarrow$ apply(b, p) //apply patch
  $\mathcal{N}_{b'} \leftarrow$ validate(b', C, $\mathcal{T}$)
  if $\mathcal{N}_{b'}$ == 0 **then**
    $plausible \leftarrow$ <p,$\mathcal{L}$>
    break //break repair iteration when plausible patch is found
  end if
  if i > MAX **then**
    break //end repair iteration
  end if
  if $\mathcal{L} \in \emptyset$ or $\mathcal{N}_{b'} < \mathcal{N}_b$ **then**
    $\mathcal{L} \leftarrow \mathcal{F}$ (b', C, $\mathcal{T}$) // fault localization iteration
    $\mathcal{N}_b \leftarrow \mathcal{N}_{b'}$
  end if
  for loc in $\mathcal{L}$ **do**
    $\mathcal{P} \leftarrow \mathcal{G}$ (b', loc, k)
    for p in $\mathcal{P}$ **do**
      ITER (b', p, k, i++, $\mathcal{F}$, $\mathcal{G}$, C, $\mathcal{T}$, loc, $\mathcal{N}_b$) // repair iteration
    end for
  end for

Table 1: Iterative Training Dataset. In each iteration, we execute ITER to generate two patches per bug and keep the valid ones for the next iteration.

Training Iteration	Self-augment Samples	Total
Initial	-	651 787
Iteration_1	+541 192	1 192 979
Iteration_2	+879 012	2 071 991
Iteration_3	+1 305 873	3 377 864
Total Training Samples for ITER	+2 726 077	3 377 864

3 EXPERIMENTAL METHODOLOGY

3.1 Research Questions

- **RQ1 (Effectiveness)**: To what extent does ITER perform compared to related work?
- **RQ2 (Multi-location Bugs)**: To what extent does ITER repair multi-location bugs?
- **RQ3 (Ablation Study)**: To what extent does each component in ITER contribute to the final effectiveness?
- **RQ4 (Time Cost)**: What's the time cost of ITER in generating plausible patches?

3.2 Implementation

Training Dataset We implement ITER on the data from Ye et al. [59] comprising 651 787 patches, including 281 762 compilation error samples and 370 025 functional error sample. Recall that it is key for us to have both kinds of errors to enable iterative repair. ITER follows the same code representation as [59] by considering the context code surrounded by buggy code, additional meta-information (e.g., variables grouped by type), and diagnostics of the bug. ITER follows the prior with to split compilation error diagnostic and functional error diagnostic with special tokens $[CE]$ and $[FE]$. We configure ITER to take a maximum of 384 input tokens as context code and generate a patch with a maximum of 76 tokens with a batch size of 32 and a vocabulary size of 32 128.

ITER is configured to be trained with three iterations, i.e., $MAX = 3$, and the beam search width $k = 2$. This means that ITER generates two candidate patches for each single given buggy location and this continues for three iterations. Table 1 gives the self-augment training dataset generated by ITER during an iterative loop. In total, ITER is trained on 3 377 864 training samples including 2 726 077 self-augment samples generated with our novel iterative loop. This means that ITER's iterative training results in 5X more training samples than simple self-supervised data generation per [59] (3 377 864 vs. 651 787). We observe that the number of self-augment training samples is not exactly twice than that of the training samples in the last iteration. This is because ITER discards those generated patches that are identical to the buggy and ground truth fix (see Algorithm 1).

Inference ITER integrates GZoltar [37] as fault localization to provide suspicious buggy locations to be repaired. There are two reasons for choosing GZoltar as the fault localization tool. First, GZoltar is widely used in program repair related works, such as DEAR [24], Recoder [62], SelfAPR [59], SimFix [16], and TBar [25]. Considering the same implementation and same suspiciousness formula Ochiai (default formula used in GZoltar) enables us to conduct a fair comparison with the related works. Second, the effectiveness of GZoltar in detecting real-world bugs has been demonstrated in previous comprehensive evaluations with different fault localization techniques [36]. ITER is implemented with GZoltar 1.7.2 [1]. Specifically, we configure the fault localization with `ochiai` formula and `entropy` metric. The detailed setting can be found in our online repository.

At inference time, ITER at most processes the top-50 suspicious locations identified by GZoltar. This is a smaller number compared to related work, such as Hercules [39], which processes top-200 suspicious locations. However, considering the iterative nature of the repair process, ITER indeed generates more than 200 tentative patches at the end. ITER configures the repair max iteration $MAX = 3$ and the beam search width $k = 10$. Consequently, ITER generates 10 patches in each iteration, and this results in at most $10^3 = 1000$ patches according to Equation 1. Performing three iterations ($MAX = 3$) is a reasonable number based on our experimental results (we demonstrate it in RQ1). Beyond this iteration threshold, there will be a significantly increased computational overhead. While a common choice for the beam search width ranges from 50 to 100 (SequenceR [8] and Recoder [62] consider a beam width of 50 and 100, respectively), ITER deliberately chooses a narrower beam search width of $k = 10$ to accommodate the iterative process. This sacrifice allows ITER to focus on the iteration process and enables early stopping when a partial or plausible patch is found, optimizing efficiency and reducing unnecessary computational overhead.

3.3 Patch Assessment

The patch Assessment follows the existing related work [8, 18, 28, 51, 62]. All the plausible patches are manually analyzed based on the ground truth developer's patch. To sum up, a patch is deemed correct if 1) it is identical to the developer-provided patch, 2) it is identical to correct patches generated by existing techniques

[1] https://github.com/GZoltar/gzoltar/releases/tag/v1.7.2. Released on May 9, 2019

Table 2: Comparison with State-of-the-art. In the cells, x/y : x denotes the number of correct patches, and y denotes the number of plausible patches that pass all human-written test cases. A '-' indicates that the number has not been reported in the corresponding article.

Projects	Bugs	SimFix	TBar	Recoder	AlphaRepair	SelfAPR	ITER		
							1	2	3
Chart	26	4/8	9/14	8/14	6/-	6/10	6/10	8/11	9/13
Cli	39	0/4	1/7	3/3	5/5	2/7	3/7	5/11	6/13
Closure	133	6/8	8/12	13/27	12/	12/20	11/18	13/21	15/22
Codec	18	0/2	2/6	2/2	6/7	3/6	1/3	1/4	2/6
Compress	47	0/6	1/2	1/3	1/3	1/2	0/2	1/4	2/6
Csv	16	0/2	1/5	4/4	1/2	0/1	1/2	2/5	2/5
JacksonCore	26	0/0	0/6	0/4	3/3	1/3	1/3	2/3	2/3
Lang	65	9/13	5/14	9/15	11/-	6/12	5/9	9/10	9/11
Math	105	14/26	18/36	15/30	16/-	13/24	12/23	21/32	25/36
Time	26	1/1	1/3	2/2	3/-	1/3	1/2	2/4	2/4
Total	476	34/70	46/105	57/104	64/-	44/91	41/79	64/105	**74/119**

where the patches have been publicly reviewed by the community in an open-source repository, 3) it is semantically equivalent to the developer-provided patch.

3.4 Methodology for RQ1

In RQ1, we compare ITER against the state-of-the-art program repair baselines on 10 open-source projects from widely used benchmark Defects4J 2.0 [19]. Since ITER is based on the core concept of iterative fault localization, we compare against techniques that also rely on fault localization, spanning over both learning-based and template-based APR. For learning-based APR, we choose three recently published approaches evaluated on Java bug datasets with fault localization: Recoder [62], SelfAPR [59], and AlphaRepair [51]. For template-based APR, we choose two best-performing techniques with fault localization according to Zhu et al. [62] evaluation: TBar [25] and SimFix [16]. Note that all papers [8, 17, 18, 28, 50, 60] assuming perfect fault localization are naturally excluded from this comparison. We report the quantitative results from the corresponding papers and repositories [26, 51, 62]. We measure the number of plausible patches and correct patches according to the criteria discussed in Section 3.3.

3.5 Methodology for RQ2

One of the main contributions of ITER is to integrate fault localization into an iteration loop and to re-execute fault location based on partial patches. This is the essential element to further improve partial patches with a better ranked list of unrepaired locations. Therefore, we evaluate ITER in repairing multi-location bugs. For RQ2, we take a subset from Defects4J 2.0 [19] by considering the following criteria: 1) multi-class bug: the ground truth patch by the developers spread more than one buggy class. 2) multi-edit bug: the ground truth patch contains more than one non-contiguous edit location in one buggy class. 3) the ground truth patch contains less than five edit locations in total. This gives us 35 multi-location bugs for evaluation.

We compare ITER against the state-of-the-art of multi-location repair approaches: Hercules [39], VarFix [49] and DEAR [24]. Hercules repairs multi-location bugs by searching for siblings (i.e., similar-looking code). VarFix repairs multi-location bugs by merging all potential edits into a meta-program, where they are guarded by if-conditions that are controlled at runtime. To compare with DEAR [24], we re-implement it because of the unavailability of the repaired bugs and patches in their open repository. Following the description provided in the paper, we re-implement DEAR's fault localization technique which consists of two steps: pair-wise hunk detection and multiple-statement expansion. In the first step, a hunk detection model is trained to learn whether two statements from different hunks should be fixed together. In the second step, DEAR trains a multiple-statement expansion model to determine whether the candidate locations determined in the first step should be expanded to their surrounding locations. In our re-implementation, both the hunk detection and statement expansion models are trained based on DistilBERT [40] [2]. The training data for hunk detection learning consists of training pairs that combined from the top-50 ranked statements from GZoltar based on 430 Defects4J bugs. On the other hand, the training data for statement expansion model consists of training points directly from those top-50 ranked statements. In both training dataset setups, the human developer patch locations provided by Defects4J are considered as the ground truth of buggy. The patch generator is ITER's one, which enables us to well isolate the effect of fault localization. The detail of re-implementation setup is given in our open-source repository.

3.6 Methodology for RQ3

In RQ3, we conduct an ablation study to evaluate the contribution of self-supervised training samples that are added at each training iteration. Specifically, we measure performance with: ITER with no additional sample (denoted as $ITER_0$), augmented with training samples in iteration 1 ($ITER_1$), augmented with training samples in iteration 2 ($ITER_2$). We evaluate performance on Defects4J (v1.2) per the number of bugs that are generated with plausible patches.

3.7 Methodology for RQ4

In RQ4, we analyze the time cost spent in generating plausible patches at inference time. This time costs sum up the costs of calling the neural network, running the compiler and tests to identify partial patches, and re-running fault-localization. Specifically, we analyze the time cost depending on the number of inference iterations and report the minimum, maximum, and median time cost of plausible patches generated by ITER.

[2] provided by Hugging Face: https://huggingface.co/distilbert-base-uncased

4 EXPERIMENTAL RESULTS

4.1 Answers to RQ1: Effectiveness

In RQ1, we compare the effectiveness of ITER with five state-of-the-art program repair techniques over 476 bugs from 10 open-source projects from the Defects4J v2.0 benchmark. Table 2 give the evaluation results. The first column gives the project name and the second column gives the number of bugs in this project for evaluation. From the third to the last columns, we report the evaluation results of correct patches and plausible patches separated by a slash (correct/plausible). Specifically for ITER, we evaluate the repair results of the ITER over all three iterations of iterative inference, which is indicated by the iteration number.

ITER successfully generates correct patches for 74 bugs and plausible patches for 119 bugs, which outperforms all previous techniques over both template-based and learning-based APR techniques. ITER improves the next best baseline AlphaRepair [51] by 15.6% (74 vs. 64) in terms of correct patch generation. ITER improves the next best baseline TBar [25] by 13.3% (119 vs. 105) in terms of plausible patch generation.

Improvement over iteration. The last three columns clearly show that ITER's effectiveness comes from our backbone concept of iterative repair. ITER boosts the correct patch number from 41 in `iteration 1` to 74 in `iteration 3`, which is an 85.4% improvement compared to only using the patch generator once. We now give two examples to illustrate how ITER generates correct patches based on partial patches that contain either a compiler error or a functional error.

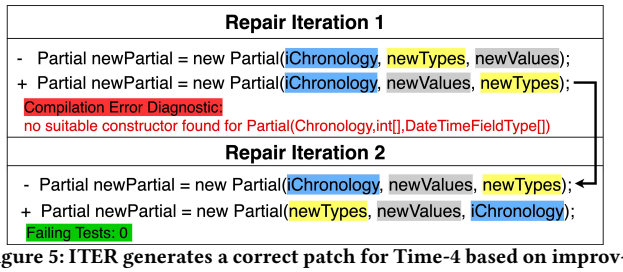


Figure 5: ITER generates a correct patch for Time-4 based on improving a partial patch with a compiler error.

A correct patch derived from a temporary uncompilable. Figure 5 gives an example where ITER generates a correct patch for bug `Time-4` with two iterations, where the patch is identical to the ground truth fix provided by the developer. This patch requires a swap among three parameters in a method call. In the first iteration, ITER swaps the positions of only two parameters `newTypes` and `newValues`, and this leads to a compiler error: type checking fails because of the absence of the appropriate constructor. In the second iteration, ITER produces a patch based on the previous compiler error by swapping another set of parameters `iChronology` and `newTypes`. The second repair iteration leads to a correct patch. This patch is only found in the second iteration and consists of an iterative chain of compilation error repair and functional error repair.

Analysis of patch evolution over iterations. We also analyze the evolutionary path of plausible patches generated by ITER. Recall that for each bug, there are multiple candidate patches that could be generated by compounding beam and iterative improvement. ITER generates a total 1432 different plausible patches for 119 bugs, approximately 12 plausible patches per bug. Table 3 visualize the evolutionary path for 1432 plausible patches. In the table, the `CE` and `FE` respectively indicate compilation error and functional error, i.e. a test failure.

Table 3: Evolution analysis of 1432 plausible patches. CE and FE respectively indicate compilation error and functional error.

No. Iteration	Patch Evolution	Patches	Percentage
1 (231)	P1: plausible	231	16.2%
2 (390)	P2: CE->plausible	103	7.2%
	P3: FE->plausible	287	20.0%
3 (811)	P4: CE->CE->plausible	62	4.3%
	P5: CE->FE->plausible	19	1.3%
	P6: FE->CE->plausible	335	23.4%
	P7: FE->FE->plausible	398	**27.8%**
Total	-	1432	100%

The first column gives the number of iterations and the total number of plausible patches found at this iteration. The second column gives the evolution path in each iteration. For example, `CE->plausible` means that the first partial patch is with a compilation error and the improved patch in the second iteration is plausible.

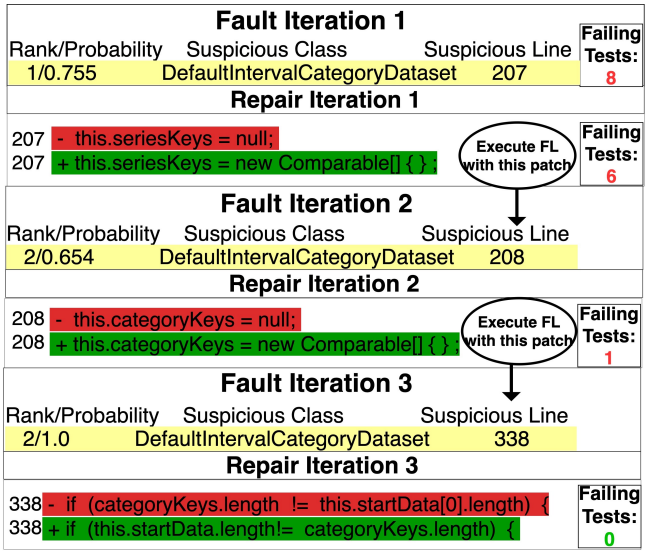
Overall, we make the following observations. First, each iteration indeed produces more plausible patches: 231 < 390 < 811. Because of the compounding effect of beam and fault localization, the trend is clearly superlinear. Second, both temporary compilation errors (CE) and functional errors (FE) contribute to the final effectiveness of ITER. There are four patch evolutions (P2, P4, P5 and P6) that involve partial patches with compilation errors and yield 36.2% (7.2% + 4.3% + 1.3% + 23.4%) of the final plausible patches.

> Answer to RQ1: ITER correctly fixes 74 bugs for 10 open-source projects from Defects4J 2.0, which outperforms our strong baselines from previous research with fault localization. This state-of-the-art performance is made possible by the novel concept of iterative inference, providing the unique capability to iteratively improve partial patches towards final correct patches.

4.2 Answers to RQ2: Multi-location Bugs

In RQ2, we focus on multi-location bugs, which is arguably one of the frontiers of program repair research. Table 4 gives the evaluation of ITER in repairing 35 multi-location bugs from Defects4J, with a quantitative comparison against VarFix [49], Hercules [39] and DEAR [24]. The first fourth columns show the bug information including their bug id, the number of classes that required edits, the number of edit locations in the ground truth patches, and the number of failing tests. The repair results are shown in the last four columns, where a ✓ indicates the bug is successfully repaired and a ☑ indicates the bugs only repaired by ITER and not by any technique from previous research. In the last row, we summarize the repair results by showing the total number of repaired bugs.

Effectiveness Over the 35 evaluated multi-location bugs, ITER repairs 17 of them with 9 unique bugs that have never been repaired in the literature before. This result outperforms Hercules by 41.6% (17 vs. 12), VarFix by 240% (17 vs. 5) and DEAR by 750% (17 vs. 2). We explain this major improvement of ITER as follows. First, none of



(a) Iterative repair process of ITER.

```
207 -    this.seriesKeys = null;
207 +    this.seriesKeys = new Comparable[] { } ;
208 -  this.categoryKeys = null;
208 +  this.categoryKeys = new Comparable[] { } ;
                ... ...
338 -  if (categoryKeys.length  !=  this.startData[0].length)  {
338 +  if (this.startData.length!=  categoryKeys.length)  {
```

(b) Final patch generated by ITER.

Figure 6: A multi-location patch uniquely found by ITER for Chart-16.

the three related work conducts any kind of patch improvement or iterations, ITER is the first approach to iteratively repair bugs with neural networks. This concept of iterative inference is essential to stack edit locations one after the other and synthesize a final multi-location patch. Second, ITER is the only approach that re-executes fault localization after a partial patch is found that reduces the number of failing tests. This key technical feature eliminates the suspicious locations that have been repaired, and enables the other buggy locations involved in the multi-location bug to be identified and better ranked, as more suspicious.

Case Study. Figure 6 gives a unique patch for bug `Chart-16`, that is only found by ITER. Bug `Chart-16` is a multi-location bug because it requires three fixes. Figure 6 (a) shows the iterative process of ITER in generating this patch and (b) gives a final patch generated by ITER after several iterations. ITER starts the repair iteration based on the suspicious `line 207` identified in suspicious class `DefaultIntervalCategoryDataset`, ranked in the 1st place with a suspicious probability score of `0.755`. After one repair iteration, ITER obtains a partial patch that reduces the original 8 failing tests to 6 failing tests. According to the design of ITER (see line 23 in Algorithm 2), ITER re-executes fault localization in the next repair iteration and identifies new suspicious locations. In the second repair iteration, ITER obtains another partial patch that reduces the previous 6 failing tests to 1 single failing test, which results in re-executing the fault localization again. Finally, ITER generates a patch that passes all test cases. This enables ITER to succeed in generating the patch for `Chart-16`.

This bug cannot be repaired by VarFix due to the lack of appropriate suspicious locations as they only execute fault localization once. Hercules by its design cannot repair this bug as Hercules only searches for similar-looking code to apply the same fix in different locations, while here the patch is not composed of similar code.

Table 4: Multi-location Bug Repair. ITER improves over the related work that specifically targets multi-location repair.

Bugs				VarFix	Hercules	DEAR	ITER
ID	Classes	Edits	Failing Tests				
Chart-14	2	4	4		✓		
Chart-16	1	3	8				✓
Cli-34	1	2	2				✓
Closure-4	1	2	2		✓		✓
Closure-78	1	2	1				
Closure-79	2	2	5				✓
Closure-106	2	2	4				
Closure-109	1	2	2		✓		
Closure-115	1	2	7				✓
Closure-131	1	2	2				✓
Codec-1	3	3	5				✓
Lang-7	1	2	1				
Lang-27	1	2	1				
Lang-34	1	2	27				✓
Lang-35	1	2	1				
Lang-41	1	4	2				
Lang-47	1	2	2		✓		✓
Lang-60	1	2	1		✓		
Math-1	2	2	2				✓
Math-4	2	2	2		✓		✓
Math-22	2	2	2	✓		✓	✓
Math-24	1	2	1		✓		
Math-35	1	2	4	✓	✓		✓
Math-43	1	3	6		✓		✓
Math-46	1	2	2	✓	✓		✓
Math-62	1	3	1	✓			
Math-65	1	2	1				
Math-71	2	2	2				
Math-77	2	2	2				✓
Math-79	1	2	1				
Math-88	1	3	1	✓			
Math-90	1	2	1				
Math-93	1	4	1				
Math-98	2	2	2		✓	✓	✓
Math-99	1	2	2				
Total 35 bugs				5	12	2	17

Let us now discuss the performance of DEAR, which aims to find out multiple related suspicious locations and repair them at once with deep learning models. Table 5 gives a comprehensive analysis of fault localization for the 17 repaired bugs. The analysis compares the fault localization results obtained from DEAR, ITER without iterative fault localization (FL), and ITER with iterative FL. In the table, the ✓and ✗ respectively indicates the true location is selected or not selected for repair edits, while the numbers indicate the ranked position for the corresponding edit location. The ↑, ↓, and → symbols signify whether the ranked position improves, decreases, or remains unchanged for ITER with iterative FL. The ✓ in the last column indicates the bugs uniquely repaired with iterative FL. In the final row, we summarize the number of repaired bugs based on the three considered analyses. ITER successfully includes all correct edit locations for 17 bugs, whereas DEAR only selects for two bugs (Math-22 and Math-98), and from this, we draw the following observations.

Table 5: Detailed Analysis of Fault Localization. The ✓and ✗ respectively indicates the true location is selected or not selected for repair edits, while the '-' symbol denotes that the edit location is not applicable.

Bugs	DEAR FL			ITER w/o iterative FL			ITER Iterative FL			
	Edit1	Edit2	Edit3	Edit1	Edit2	Edit3	Edit1	Edit2	Edit3	Uniquely Repaired
Chart-16	✗	✗	✓	1st	2nd	19th	1st	2nd	2nd (↑)	
Cli-34	✗	✗	-	42nd	✗	-	42nd	3rd (↑)	-	✓
Closure-4	✗	✗	-	35th	✗	-	35th	39th (↑)	-	✓
Closure-79	✗	✗	-	6th	✗	-	6th	4th (↑)	-	✓
Closure-115	✗	✗	-	1st	10th	-	1st	5th(↑)	-	
Closure-131	✗	✗	-	16th	✗	-	16th	4th (↑)	-	✓
Codec-1	✗	✗	✗	1st	3rd	✗	1st	1st (↑)	1st (↑)	✓
Lang-34	✗	✗	-	13th	15th	-	13th	28th(↓)	-	
Lang-47	✗	✗	-	5th	9th	-	5th	4th(↑)		
Math-1	✗	✗	-	1st	8th	-	1st	3rd (↑)	-	
Math-4	✗	✗	-	4th	✗	-	4th	6th (↑)	-	✓
Math-22	✓	✓	-	1st	2nd	-	1st	1st (↑)	-	
Math-35	✗	✗	-	3rd	7th	-	3rd	7th(→)	-	
Math-43	✗	✗	✗	35th	36th	37th	35th	36th (→)	37th (→)	
Math-46	✗	✗	-	1st	✗	-	1st	1st (↑)	-	✓
Math-77	✗	✗	-	1st	17th	-	1st	11th (↑)	-	
Math-98	✓	✓	-	4th	14th	-	4th	4th(↑)	-	
Number of Repaired Bugs	2			10			17			

Our experiments demonstrate that ITER, both without and with iterative fault localization, significantly outperforms DEAR thanks to the concept of partial patches. DEAR heavily relies on identifying the correct combination of pair-wise edit locations and aims to fix them together in a simultaneous manner. In contrast, ITER does not rely on pair-wise locations and instead leverages the partial patches to make decisions to keep or discard an edit location. Consequently, DEAR redundantly makes multiple repair attempts for a single edit location, depending on how many times it is combined with other different edit locations. On the other hand, ITER repairs each location independently. If ITER succeeds in producing a partial patch for one edit location, the partial patch is retained and that location is not revisited again. This enables ITER to mitigate the pair-wise exponential explosion.

In theory, the likelihood for DEAR to identify correct pair-wise locations from a large combination of pairs is indeed low. For instance, when considering the top-50 suspicious edit locations suggested by SBFL, the probability of correctly identifying the pair is only $\frac{1}{^{50}C_2}$ (0.08%). This is confirmed in practice, resulting in the identification of only two successful pairs of edit locations.

ITER with iterative fault localization demonstrates a substantial improvement by repairing 7 more bugs compared to ITER without iterative FL. This is because of the failure of simple SBFL to include the edit locations in the ranked list of considered suspicious locations. However, upon re-executing FL with partial patches, these previously neglected edit locations emerge in the top list. Notably, over 17 bugs, iterative FL enhances the ranking of edit locations for 16 bugs by re-ranking the edit locations in earlier positions or emerge (↑). This refinement in FL enables ITER to faster locate the correct patches by providing more accurate edit locations.

Analysis of Unrepaired Bugs Over 35 evaluated bugs, ITER fails to repair 17 of them and we summarize the following limitations. First, there are 12 bugs ITER that cannot be repaired due to the single failing test. ITER requires the signal of failing test cases to keep the partial patch and locate different locations. Second, there are 6 bugs ITER that cannot be repaired due to the complexity of partial patches. ITER only conducts three iterations and fails to generate the correct fixes at all different locations. This suggests increasing the iteration number would be possible to repair more multi-location bugs.

Table 6: Ablation Study of ITER w.r.t Training and Inference Iterations. The numbers are the plausible patches.

Training Iteration	Training Samples	Inference Iteration		
		1	2	3
$ITER_0$	651 787	68	93	104
$ITER_1$	1 192 979	71	101	113
$ITER_2$	2 071 991	72	104	117
$ITER$	3 377 864	79	105	119

Answer to RQ2: ITER outperforms all multi-location repair approaches. This demonstrates the power of iterative repair. Re-execution of fault localization is the key to correctly locating suspicious locations to be repaired in a meaningful sequence. Nine (9) multi-location bugs repaired by ITER were never reported as fixed in previous literature.

4.3 Answers to RQ3: Ablation Study

In this RQ, we do an ablation study regarding the importance of the number of training iterations. We evaluate ITER with a different number of training iterations. Table 6 gives the evaluation results on 476 bugs from 10 open-source projects in Defects4J 2.0. It is the matrix where the columns represent the number of training iterations while the rows capture the number of inference iterations. In addition, the second column gives the number of generated self-augment training samples. Each cell indicates the number of bugs that have been repaired with plausible patches.

As expected, the training iterations do increase the number of generated training samples, approx. 2x for every training iteration. Regarding performance, we make the following observations. First, more training iterations with more training samples indeed improve the effectiveness of ITER (performance monotonously improves column-wise). Second, looking at the first row (no training iteration, one single pass of training sample generation), we see that

the inference iteration alone improves the effectiveness of ITER. This suggests all neural repair approaches would potentially benefit from iterative inference. Third, the number of inference iterations contributes more than the number of training iterations. For example, by using three inference iterations, ITER boosts the plausibility number from 79 to 119 (last row), which is a 25.2% improvement. Lower gains are observed across columns, with for example a 14.4% improvement in the last column (104 to 119).

> Answer to RQ3: ITER introduces two novel concepts for program repair, iterative training and iterative inference. Our experiments show that both contribute to the final effectiveness of ITER. Yet, iterative inference makes a bigger contribution than iterative training. This suggests that most related work in program repair can achieve better performance by using this potent concept at inference time. This remark applies to the recent promising results on performing program repair with large language models [10, 17, 50, 51].

4.4 Answers to RQ4: Time Cost

Figure 7 shows the time cost distribution of 1432 plausible patches discussed in RQ1. For a single patch, the overall time cost stacks neural net usage, compiler and test execution and fault localization. The x-axis indicates the number of inference iterations and the y-axis indicates the hours spent in generating the plausible patches.

The majority of plausible patches are produced in the range of 1 to 10 hours. Per the concept of iterative inference, the time cost increase with the number of inference iterations, both in terms of median and outliers. The full quantitative analysis of the time cost is given in Table 7, where the first three columns show the minimum, maximum and median time that ITER costs. The range is from `0.0053` hours to at most `84.5` hours. We can see half of the plausible patches are generated within `4.57` hours (median in the third column). In addition, we compute that 18.7% and 33.2% of plausible patches are generated within respectively one and two hours. This is significant per the study by Noller et al. [35] showing that patches generated within one hour are appreciated by developers in practice.

We give those end-to-end results because they are the ones that matter to practitioners. We note that these numbers cannot be directly compared with data from related papers because they tend to exclude the fault localization time and patch validation time. One reason for that is many experiments assume perfect fault localization (e.g., Xia et al. [50] reported that the LLM takes 2.5 hours to generate 600 candidate patches for Defects4J with perfect fault localization). Therefore, we argue that our measurements are more meaningful both from a scientific and industrial perspective.

> Answer to RQ4: Half of the plausible patches from ITER are generated within 4 hours and a half. 18.72% of plausible patches can be generated within one hour.

5 THREATS TO VALIDITY

An internal threat relates to 1) our fault localization tool GZoltar may cause failures for some evaluated bugs due to dependency and version issues, which prevents ITER from generating more plausible patches. 2) manual patch correctness assessment. To mitigate these threats, we make our tool and generated patches publicly available.

Per the standards of the field, our approach has been tested in one language (Java) and the evaluation is carried out on well-established benchmarks. A threat to external validity relates to whether the novel concepts of ITER, iterative training, and iterative inference generalize to arbitrary programming languages. We do believe that our approach can be applied to other programming languages and datasets and that performance would follow accordingly.

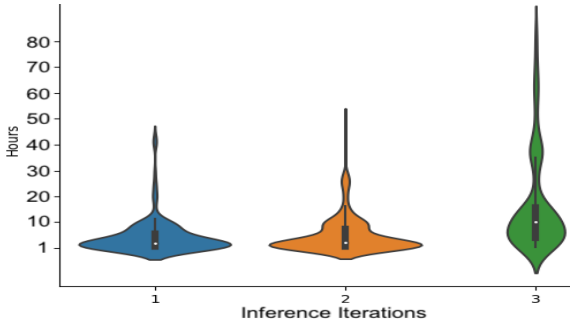


Figure 7: Time cost of ITER in generating plausible patches.

6 RELATED WORK

6.1 Program Repair

Over the past decade, extensive research efforts have yielded a substantial body of work on automatic program repair, which can be broadly categorized into three categories [14]: search-based repairs, semantic-based repairs and learning-based repairs. Search-based repair approaches, such as GenProg [22], Relifix [43], ssFix [53], SoFix [27], and others [7, 13, 29, 52, 61] typically define a search space with different kinds of edit patterns, such as inserting null checks [9], copying statement [48], mutating operators [44], or modifying expressions [21, 25]. Search-based repair approaches first generate candidate patches and then they use a search algorithm to locate plausible patches from patch search space. Contrary to search-based repair generation approaches, semantic-based program repair techniques, such as Nopol [55], SemFix [34] and others [11, 20, 41, 42, 54] first construct constraints that should be satisfied to fix a bug, and then they use program synthesis to synthesize patches that satisfy the repair constraints.

Learning-based repair uses neural networks, and has been mostly with supervised learning on commits. Supervised learning-based repair approaches, such as Codit [5], Modit [6], CURE [18], DLFix [23] and Tufano's work [45, 46] learn to generate fixes from past commits submitted by developers: given a buggy code and surrounding context, the neural network is optimized to translate the buggy code to the correct code. In essence, supervised learning-based repair learns code transformations from a vast number of past commits, e.g., open-source projects in GitHub [8, 18, 28, 45]. Contrary to the above supervised learning approaches, another line of work [2, 56, 57, 59] has proposed to use self-supervised learning to completely get rid of commit collection in the learning process.

There are two key differences between ITER and the aforementioned works. First, none of them conducts iterative refinement with partial patches, by chaining repairs of compilation errors and

Table 7: Time cost analysis of ITER in generating 1432 plausible patches

Time			Proportion	
min	max	median	< 1 Hour	< 2 Hours
0.0054 H	84.50 H	4.57 H	18.72 %	33.20%

test failure errors. Second, no existing work re-executes fault localization with partial patch and demonstrates major empirical results on multi-location bugs.

6.2 Patch Evolution

Prior work considered patch evolution into the repair process, which is related to the idea of iterative repair presented in this paper. For example, Gupta et al. [15] propose DeepFix to iteratively repair compilation errors. ITER is different from DeepFix as follows. First, DeepFix's iterative process only proceeds when all previous repairs are accepted by the compiler (i.e., compilable). If any patch is rejected and deemed non-compilable, DeepFix's iterative process is terminated. In contrast, ITER conducts novel patch refinement through multiple attempts to even improve low-quality patches. Second, DeepFix does not incorporate a fault localization process since it focuses on resolving compilation errors, while one of the key components in ITER is the iterative fault localization process with partial patches.

Arcuri and Yao [3] present a co-evolutionary approach to automatically fix software bugs. Their approach utilizes genetic programming to evolve programs that pass the existing set of unit tests, while also employing test generation techniques yield new tests for finding bugs in the evolutionary programs. ITER shares a similar concept of patch evolution. However, there are some notable differences. Arcuri and Yao's work assumes a known formal specification, allowing for the generation of potentially infinite new test cases. On the other hand, ITER considers a fixed number of test cases written by developers from the real-world, without assuming any high-level specification. In addition, ITER incorporates an iterative fault localization process which is not considered in their work.

It is worth mentioning that GenProg [22, 47], over a decade ago, considered an iterative process of stacking edits by a genetic programming algorithm. Yet, we revisit this idea with major novelty. First, GenProg does not consider partial patches with compilation errors which are simply discarded. ITER transforms uncompilable patches into plausible and correct ones. Second, GenProg does not re-execute fault localization for partial patches. ITER demonstrates that this is essential for multi-location bugs. Third, GenProg is based on AST transformation, while ITER builds up on deep learning for repair that did not exist at GenProg time and has been shown powerful in recent years.

6.3 Multi-location Program Repair

In practice, most APR approaches do not aim to multi-location repair, and there are only a few works that target multi-location repair based on symbolic variables and heuristics for patch location combination. Semantic program repair techniques, such as DirectFix [31] and Angelix [32], facilitate multiple-location fixes by substituting multiple suspicious expressions with multiple symbolic variables. The synthesis of a multi-location patch involves replacing multiple symbolic variables with concrete values through the process of symbolic execution and constraint solving. Wong et al. [49]'s VarFix use variational execution to merge all possible edits into a meta-program, where all edits are guarded by if-conditions that control whether to include the edit at runtime. Saha et al.'s Hercules [39] repair multi-location bugs by searching for similar code in commit history to find so-called evolutionary siblings. In essence, Hercules applies a substantially similar transformation at a number of different locations. Li et al. [24]'s DEAR is a deep learning-based approach, which selects and combines multiple buggy locations provided by SBFL. DEAR's objective is to generate patches for all identified locations simultaneously to repair multi-location bugs. ITER is fundamentally different. First, none of approaches mentioned above considers an iterative repair process to refine the generated patches and they do not re-execute fault localization to obtain updated suspicious locations. This is what we do, and we provide strong empirical evidence that this works. Second, ITER is not limited to repairing multiple locations with similar edits as Saha et al. [39]. Our experimental replication package contains multi-location patches where the edits are completely different one from another, yet combined in a meaningful multi-location patch. Third, unlike Hercules and DEAR, ITER does not necessitate the combination of multiple buggy locations. Instead, ITER utilizes partial patches to determine whether to retain or discard each individual buggy location.

7 CONCLUSION

We have presented a novel neural program repair paradigm called iterative repair. Our approach, called ITER, integrates fault localization, patch generation, and patch validation in an original iterative loop. This results in two breakthroughs: First, ITER is able to improve partial patches that have been generated in prior repair attempts in order to converge to a final correct patch. Second, ITER is able to push the state-of-the-art on repairing multi-location bugs with 9 unique multi-location bugs that were never repaired before. The original concepts ITER has wide applicability, incl. on maximizing repair performance with large language models.

Our future work focuses on improving the performance of ITER. To this end, we aim to investigate patch minimization to filter out less useful patches and not keep them for the next iteration. Having a good patch minimization technique will greatly improve the performance of ITER and benefit finding plausible patches at an early time.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for the insightful feedback. This work was partially supported by The Wallenberg Foundation and WASP Postdoctoral Scholarship Program - KAW 2022.0368. We thank Xuechun Xu from KTH Royal Institute of Technology for the support of computing resources. Some experiments were performed on resources provided by the Swedish National Infrastructure for Computing.

REFERENCES

[1] Rui Abreu, Peter Zoeteweij, and Arjan J.C. van Gemund. 2007. On the Accuracy of Spectrum-based Fault Localization. In *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*. 89–98.

[2] Miltiadis Allamanis, Henry Jackson-Flux, and Marc Brockschmidt. 2021. Self-Supervised Bug Detection and Repair. In *Advances in Neural Information Processing Systems*.

[3] Andrea Arcuri and Xin Yao. 2008. A novel co-evolutionary approach to automatic software bug fixing. In *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*. 162–168. https://doi.org/10.1109/CEC.2008.4630793

[4] Sahil Bhatia, Pushmeet Kohli, and Rishabh Singh. 2018. Neuro-Symbolic Program Corrector for Introductory Programming Assignments. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 60–70. https://doi.org/10.1145/3180155.3180219

[5] S. Chakraborty, Y. Ding, M. Allamanis, and B. Ray. 2020. CODIT: Code Editing with Tree-Based Neural Models. *IEEE Transactions on Software Engineering* (2020). https://doi.org/10.1109/TSE.2020.3020502

[6] Saikat Chakraborty and Baishakhi Ray. 2021. On Multi-Modal Learning of Editing Source Code. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[7] L. Chen, Y. Pei, and C. A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[8] Z. Chen, S. J. Kommrusch, M. Tufano, L. Pouchet, D. Poshyvanyk, and M. Monperrus. 2019. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2019).

[9] Thomas Durieux, Benoit Cornu, Lionel Seinturier, and Martin Monperrus. 2017. Dynamic patch generation for null pointer exceptions using metaprogramming. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 349–358. https://doi.org/10.1109/SANER.2017.7884635

[10] Zhiyu Fan, Xiang Gao, Martin Mirchev, Abhik Roychoudhury, and Shin Hwei Tan. 2023. Automated Repair of Programs from Large Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*.

[11] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-Avoiding Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 8–18. https://doi.org/10.1145/3293882.3330558

[12] Luca Gazzola, Daniela Micucci, and Leonardo Mariani. 2017. Automatic Software Repair: A Survey. *IEEE Transactions on Software Engineering* (2017).

[13] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical Program Repair via Bytecode Mutation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 19–30. https://doi.org/10.1145/3293882.3330559

[14] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[15] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. DeepFix: Fixing Common C Language Errors by Deep Learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence* (San Francisco, California, USA) *(AAAI'17)*. AAAI Press, 1345–1351.

[16] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping Program Repair Space with Existing Patches and Similar Code *(ISSTA)*.

[17] Nan Jiang, Thibaud Lutellier, Yiling Lou, Lin Tan, Dan Goldwasser, and Xiangyu Zhang. 2023. KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery.

[18] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *Proceedings of the ACM/IEEE 43rd International Conference on Software Engineering*.

[19] Rene Just, Darioush Jalali, and Michael D Ernst. 2014. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ACM, 437–440.

[20] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. JFIX: Semantics-Based Repair of Java Programs via Symbolic PathFinder. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Santa Barbara, CA, USA) *(ISSTA 2017)*. Association for Computing Machinery, New York, NY, USA, 376–379. https://doi.org/10.1145/3092703.3098225

[21] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: Syntax- and Semantic-guided Repair Synthesis via Programming by Examples. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2017)*.

[22] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on* 38, 1 (2012), 54–72. https://doi.org/10.1109/TSE.2011.104

[23] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2020. DLFix: Context-Based Code Transformation Learning for Automated Program Repair. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 602–614. https://doi.org/10.1145/3377811.3380345

[24] Yi Li, Shaohua Wang, and Tien N. Nguyen. 2022. DEAR: A Novel Deep Learning-Based Approach for Automated Program Repair. In *Proceedings of the 44th International Conference on Software Engineering* (Pittsburgh, Pennsylvania) *(ICSE '22)*. Association for Computing Machinery, New York, NY, USA, 511–523. https://doi.org/10.1145/3510003.3510177

[25] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: Revisiting Template-based Automated Program Repair. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 31–42. https://doi.org/10.1145/3293882.3330577

[26] Kui Liu, Shangwen Wang, Anil Koyuncu, Kisub Kim, Tegawendé F. Bissyandé, Dongsun Kim, Peng Wu, Jacques Klein, Xiaoguang Mao, and Yves Le Traon. 2020. On the Efficiency of Test Suite Based Program Repair: A Systematic Assessment of 16 Automated Repair Systems for Java Programs. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 615–627. https://doi.org/10.1145/3377811.3380338

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

[28] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. 2020. CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair *(ISSTA 2020)*.

[29] Matias Martinez and Martin Monperrus. 2016. ASTOR: A Program Repair Library for Java. In *Proceedings of ISSTA*.

[30] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Trans. Softw. Eng. Methodol.* 27, 4, Article 15 (oct 2018), 37 pages. https://doi.org/10.1145/3241980

[31] S. Mechtaev, J. Yi, and A. Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. 448–458. https://doi.org/10.1109/ICSE.2015.63

[32] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*.

[33] Martin Monperrus. 2017. Automatic Software Repair: a Bibliography. *ACM Computing Surveys* 51 (2017), 1–24. https://doi.org/10.1145/3105906

[34] Hoang Duong Thien Nguyen, Dawei Qi, Abhik Roychoudhury, and Satish Chandra. 2013. SemFix: Program repair via semantic analysis. In *2013 35th International Conference on Software Engineering (ICSE)*. 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[35] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust Enhancement Issues in Program Repair. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[36] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D. Ernst, Deric Pang, and Benjamin Keller. 2017. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 609–620. https://doi.org/10.1109/ICSE.2017.62

[37] André Riboira and Rui Abreu. 2010. The GZoltar Project: A Graphical Debugger Interface *(TAIC PART'10)*. Springer-Verlag, Berlin, Heidelberg.

[38] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. ELIXIR: Effective Object Oriented Program Repair. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 648–659.

[39] Seemanta Saha, Ripon K. Saha, and Mukul R. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[40] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. 2020. DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter. arXiv:1910.01108 [cs.CL]

[41] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic Program Repair. In *42nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[42] Ridwan Shariffdeen, Shin Hwei Tan, Mingyuan Gao, and Abhik Roychoudhury. 2021. Automated Patch Transplantation. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. 1–36.

[43] Shin Hwei Tan and Abhik Roychoudhury. 2015. relifix: Automated Repair of Software Regressions. In *2015 IEEE/ACM 37th IEEE International Conference on*

Software Engineering, Vol. 1. 471–482. https://doi.org/10.1109/ICSE.2015.65

[44] Christopher Steven Timperley, Susan Stepney, and Claire Le Goues. 2017. An Investigation into the Use of Mutation Analysis for Automated Program Repair. In *SSBSE*.

[45] Michele Tufano, Jevgenija Pantiuchina, Cody Watson, Gabriele Bavota, and Denys Poshyvanyk. 2019. On Learning Meaningful Code Changes via Neural Machine Translation. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 25–36. https://doi.org/10.1109/ICSE.2019.00021

[46] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. An Empirical Study on Learning Bug-Fixing Patches in the Wild via Neural Machine Translation. *ACM Trans. Softw. Eng. Methodol.* 28, 4, Article 19 (Sept. 2019), 29 pages. https://doi.org/10.1145/3340544

[47] Westley Weimer, ThanhVu Nguyen, Claire Le Goues, and Stephanie Forrest. 2009. Automatically finding patches using genetic programming. In *2009 IEEE 31st International Conference on Software Engineering*. 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[48] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-Aware Patch Generation for Better Automated Program Repair. In *Proceedings of the 40th International Conference on Software Engineering (ICSE '18)*.

[49] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: Balancing Edit Expressiveness and Search Effectiveness in Automated Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. Association for Computing Machinery, New York, NY, USA, 354–366. https://doi.org/10.1145/3468264.3468600

[50] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated Program Repair in the Era of Large Pre-trained Language Models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023)*. Association for Computing Machinery.

[51] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair via Zero-Shot Learning. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 959–971. https://doi.org/10.1145/3540250.3549101

[52] Qi Xin and Steven Reiss. 2019. Better Code Search and Reuse for Better Program Repair. In *2019 IEEE/ACM International Workshop on Genetic Improvement (GI)*. 10–17. https://doi.org/10.1109/GI.2019.00012

[53] Q. Xin and S. P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*.

[54] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise Condition Synthesis for Program Repair. In *Proceedings of the 39th International Conference on Software Engineering* (Buenos Aires, Argentina) *(ICSE '17)*. IEEE Press, 416–426. https://doi.org/10.1109/ICSE.2017.45

[55] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clément, Sebastian Lamelas, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2016. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *IEEE Transactions on Software Engineering* (2016).

[56] Michihiro Yasunaga and Percy Liang. 2020. Graph-based, Self-Supervised Program Repair from Diagnostic Feedback. In *International Conference on Machine Learning (ICML)*.

[57] Michihiro Yasunaga and Percy Liang. 2021. Break-It-Fix-It: Unsupervised Learning for Program Repair. In *International Conference on Machine Learning (ICML)*.

[58] He Ye, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2021. A comprehensive study of automatic program repair on the QuixBugs benchmark. *Journal of Systems and Software* 171 (2021), 110825. https://doi.org/10.1016/j.jss.2020.110825

[59] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE22)*. Association for Computing Machinery, Article 92, 13 pages.

[60] He Ye, Matias Martinez, and Martin Monperrus. 2022. Neural Program Repair with Execution-based Backpropagation. In *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering*.

[61] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming. In *IEEE Transactions on Software Engineering*.

[62] Qihao Zhu, Zeyu Sun, Yuan-an Xiao, Wenjie Zhang, Kang Yuan, Yingfei Xiong, and Lu Zhang. 2021. A Syntax-Guided Edit Decoder for Neural Program Repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 341–353. https://doi.org/10.1145/3468264.3468544