Enhancing the Efficiency of Automated Program Repair via Greybox Analysis

YoungJae Kim
kyj1411@unist.ac.kr
UNIST
South Korea

Yechan Park
pych05@unist.ac.kr
UNIST
South Korea

Seungheon Han
shhan@unist.ac.kr
UNIST
South Korea

Jooyong Yi
jooyong@unist.ac.kr
UNIST
South Korea

ABSTRACT

In this paper, we pay attention to the efficiency of automated program repair (APR). Recently, an efficient patch scheduling algorithm, Casino, has been proposed to improve APR efficiency. Inspired by fuzzing, Casino adaptively chooses the next patch candidate to evaluate based on the results of previous evaluations. However, we observe that Casino utilizes only the test results, treating the patched program as a black box. Inspired by greybox fuzzing, we propose a novel patch-scheduling algorithm, Gresino, which leverages the internal state of the program to further enhance APR efficiency. Specifically, Gresino monitors the hit counts of branches observed during the execution of the program and uses them to guide the search for a valid patch. Our experimental evaluation on the Defects4J benchmark and eight APR tools demonstrates the efficacy of our approach.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; **Automatic programming**.

KEYWORDS

Automated Program Repair, Patch Scheduling, Greybox Analysis

ACM Reference Format:
YoungJae Kim, Yechan Park, Seungheon Han, and Jooyong Yi. 2024. Enhancing the Efficiency of Automated Program Repair via Greybox Analysis. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27–November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Automated Program Repair (APR) is a technique that automatically generates patches for buggy programs. Since the introduction of the seminal APR tool, GenProg, in 2009 [52], APR has been actively researched for the last 15 years, introducing various approaches such as template-based [31], learning-based [50], and semantics-based [41] approaches. In particular, the repairability of APR tools has been significantly improved over the years. For example, jGenProg [36], introduced in 2017, could correctly fix only 2% of the 224 bugs in the Defects4J benchmark [20], while the latest tool, SRepair [57], can correctly fix 45% of the 695 bugs in the extended benchmark. This is a remarkable achievement in the field of APR.

While research on repairability should continue, there is another important aspect of APR that has relatively received less attention: *APR efficiency*. Recent APR tools, including the aforementioned SRepair, are often evaluated under the so-called "perfect fault localization" assumption, wherein the correct fix location is provided to the APR tool. Under this assumption, the APR tool generates hundreds of patch candidates only for the given location and then runs those candidates against the test suite to find a valid patch. This assumption is useful for evaluating the repairability of APR tools. However, it hinders the evaluation of APR efficiency. In this work, we remove the perfect-fault-localization assumption and turn our attention to APR efficiency, asking the following question: *How can we explore the patch space efficiently?*

Existing Techniques for Patch-Space Exploration. When GenProg was introduced, APR efficiency was one of its main concerns. For efficient exploration, GenProg associates each patch candidate with a fitness score and drives the search toward high-fitness candidates, using an online search algorithm, specifically genetic programming [25].

However, more recent APR tools using template-based or learning-based approaches employ a simplistic method for patch-space exploration. They simply enumerate the patch candidates in a *predefined order* based on the suspiciousness scores of the program locations to which the patch candidates are applied. Such an approach is suboptimal because it does not consider the runtime information obtained during the repair process. Indeed, Benton et al. [3] showed that APR efficiency improves when the original patch scheduling algorithms of APR tools are replaced with their online algorithm, SeAPR, which adaptively recomputes the suspiciousness scores of patch candidates based on the runtime information.

More recently, Kim et al. [21] proposed another adaptive patch-scheduling algorithm, Casino, that has been shown to be more efficient than SeAPR and GenProg's patch-scheduling algorithm.

ASE '24, October 27–November 1, 2024, Sacramento, CA, USA
© 2024 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM...$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

Casino formulates the patch space as a tree structure where each path of the tree represents a distinct patch candidate, as illustrated in Figure 2. Similar to grammar-based fuzzing [16], Casino traverses the patch-space tree by randomly selecting an edge at each node, prioritizing edges that are more likely to lead to a valid (a.k.a., plausible) patch that passes all available tests. To adjust the likelihood of each edge at runtime, Casino observes test results. When a previously failing test passes after applying a patch candidate σ, Casino increases the likelihood of the edges leading to σ.

Our Approach. We observe that Casino uses runtime information in a blackbox manner, observing only the test results. *Inspired by greybox fuzzing [4], we propose a new patch-scheduling algorithm, Gresino, that combines blackbox and greybox information to explore the patch space more efficiently.* Similar to coverage-guided greybox fuzzing such as AFL [67], Gresino observes branch hit counts during the repair process. Gresino uses this information to identify "critical" branches—those whose changes in hit counts lead to a patch σ that passes a previously failing test. Gresino then prioritizes patch candidates that are likely to affect the hit counts of the critical branches in a way similar to σ.

To evaluate Gresino, we compare its efficiency against Casino with the eight APR tools described in § 4.5. When assessed with the 395 buggy versions of the Defects4J [20] benchmark, Gresino substantially outperforms Casino in six tools, while performing similarly in two tools. The high search efficiency of Gresino also leads to higher recall than Casino, enabling it to fix more bugs successfully than Casino when given the same time budget.

In summary, we make the following contributions in this paper:

(1) **Novel Technique.** We introduce a novel patch-scheduling algorithm, Gresino, that outperforms the state-of-the-art patch-scheduling algorithm, Casino, in terms of APR efficiency. The key contributor to the efficiency of Gresino lies in its design that exploits "greybox" information—specifically, branch hit counts—during the repair process.

(2) **Extensive Evaluation.** We conduct an extensive evaluation of Gresino with eight APR tools. For each APR tool and bug, we schedule up to 3,000 patch candidates using Gresino. Considering the stochastic nature of Gresino, we repeat the evaluation 10 times for each APR tool and bug pair. For comparison, we also evaluate Casino 10 times for the same pairs.

(3) **Replication Package.** We provide a replication package that includes the implementation of Gresino and the experimental scripts to reproduce the results presented in this paper. The replication package is available at:

> *https://github.com/UNIST-LOFT/GreyboxAPR*

2 BACKGROUND

2.1 Automated Program Repair (APR)

Figure 1 shows the typical workflow of automated program repair (APR) tools. As illustrated in the figure, the majority of APR tools are test-based, meaning that they generate patches that pass all available tests. However, those patches, known as *plausible patches*, may not necessarily be correct due to the incompleteness of tests. This is a well-known *overfitting problem* of APR. Many early APR tools stop searching for patches once they find a plausible one. However, the first-found patch may be incorrect due to the overfitting problem. To avoid this problem, recent APR tools generate *multiple plausible patches*, increasing the chance of including the correct patch among them [6, 7, 12, 13, 26, 42, 54, 55, 66].

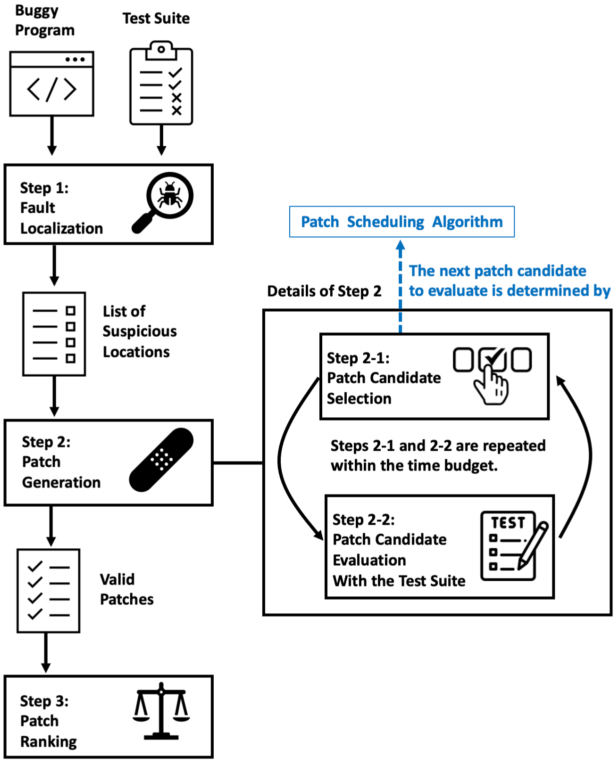

Buggy Program
Test Suite
Step 1: Fault Localization
List of Suspicious Locations
Step 2: Patch Generation
Valid Patches
Step 3: Patch Ranking
Patch Scheduling Algorithm
The next patch candidate to evaluate is determined by
Details of Step 2
Step 2-1: Patch Candidate Selection
Steps 2-1 and 2-2 are repeated within the time budget.
Step 2-2: Patch Candidate Evaluation With the Test Suite
TEST

Figure 1: Workflow of APR. In this work, we propose a new patch scheduling algorithm, Gresino, which can be plugged in at Step 2-1, as a replacement of the original patch scheduling algorithm.

2.2 Search Efficiency in APR

In this work, we focus on the *search efficiency* for finding plausible patches. Our goal is to identify as many plausible patches as possible within a given time budget, ideally all plausible patches in the patch space of a given APR tool. Please note that there exist separate techniques to find the correct patch among the plausible ones [5, 14, 17, 49, 51, 59, 61], which is orthogonal to this work.

The problem of search efficiency boils down to *how to schedule the order of evaluating patch candidates*, which corresponds to Step 2-1 in Figure 1. As more plausible patches are scheduled earlier in the patch evaluation process, the search efficiency becomes higher. In most APR approaches, the scheduling order is determined by the following two factors: (1) how suspicious a program location l is, and (2) how promising a patch candidate p is at l. The first factor is usually determined by a fault localization technique [1], and the second factor is determined by a patch prioritization technique (see § 7). A typical APR process involves visiting each suspicious location l in order of their suspiciousness and then evaluating each

Algorithm 1 CASINO Patch Scheduling Algorithm

```
Input: $P$: Program to be repaired
Input: Initial patch-space tree $\mathcal{T}$ (see the step 0 of § 2.3)
Input: $\mathbb{TS}$: Test-suite
Output: $\mathbb{D}_{\text{patches}}$
 1: $\mathbb{D}_{\text{patches}} \leftarrow \emptyset$ // $\mathbb{D}_{\text{patches}}$: Set of detected patches
 2: while $t_{\text{elapsed}} < t_{\text{limit}} \wedge$ CONTINUE() do
 3:     // Phase I: Vertical navigation via Thompson sampling
 4:     $n := \text{ROOT}(\mathcal{T})$ // Start patch-tree traversal from the root node
 5:     while $n$ has an enabled outgoing edge do
 6:         for each enabled edge $e$ of $n$ do
 7:             $Beta(\alpha_k, \beta_k) \leftarrow \text{LABEL}(e)$
 8:             Draw $\hat{\theta}_k$ according to $Beta(\alpha_k, \beta_k)$
 9:         end for
10:         $e := \arg\max_k \hat{\theta}_k$ // Select the $k$-th edge having the maximum $\hat{\theta}_k$
11:         $n := \text{TRAVERSE}(n, e)$ // Traverse edge $n \xrightarrow{e} n'$ and $n := n'$
12:     end while
13:
14:     // Phase II: Horizontal navigation via $\epsilon$-greedy algorithm
15:     $s \leftarrow \text{MOSTSUSPICIOUSSCORE}(\mathcal{T}(n))$
16:     $\mathbb{L} \leftarrow \text{CANDIDATES}(s)$
17:     Toss a coin with success probability $\epsilon$
18:     if success then
19:         $\sigma_{next} \leftarrow \text{CHOOSEATRANDOM}(\mathbb{L})$ // choose at random
20:     else
21:         $\sigma_{next} \leftarrow \text{CHOOSEINPREDETERMINEDORDER}(\mathbb{L})$
22:     end if
23:
24:     patch, execinfos $\leftarrow \text{PATCHEVAL}(P, \sigma_{next}, \mathbb{TS})$
25:     if ISVALID(patch, execinfos) then
26:         // A valid (a.k.a., plausible) patch is found
27:         $\mathbb{D}_{\text{patches}} \leftarrow \mathbb{D}_{\text{patches}} \cup$ {patch}
28:     end if
29:
30:     // Update edge labels
31:     UPDATE(execinfo, $\mathcal{T}, \mathbb{L}_{MSU}, \sigma_{next}$)
32: end while
```

patch candidate p available at l in order of their likelihood of being the correct patch. To limit the search space, most APR tools modify only a single program location and we consider this setting in this work.

In most APR approaches, the order of validating patch candidates is predetermined and remains unchanged throughout the repair process; that is, the patch scheduling is *offline*. In contrast, recent work [3, 21] has shown that the search efficiency of APR tools can be improved by replacing the offline patch scheduling algorithm with an *online* one; that is, the order of validating patch candidates changes dynamically based on the runtime information obtained during the repair process.

2.3 Online Patch Scheduling

In this subsection, we describe the current state-of-the-art online patch scheduling technique, CASINO [21], which forms the basis of our approach. CASINO represents the patch space as a tree structure where each path of the tree represents a distinct patch candidate. For example, a patch σ applicable at a program location l within a method m in a file f is represented as a path $r \rightarrow f \rightarrow m \rightarrow l \rightarrow \sigma$ where r is the root node of the tree.

The CASINO scheduling algorithm selects a patch candidate for validation by traversing the tree top-down from the root node to a leaf node. At each intermediate node, it chooses the next edge to traverse at random with higher probabilities for edges deemed more promising. Edges are considered more promising if taking them during the current repair session has led to the discovery of a greater number of "interesting" patches. The definition of an "interesting patch" is given below:

Definition 1 (Interesting Patch)**.** A patch σ is considered interesting when the program p patched with σ passes a negative (i.e., previously failing) test.

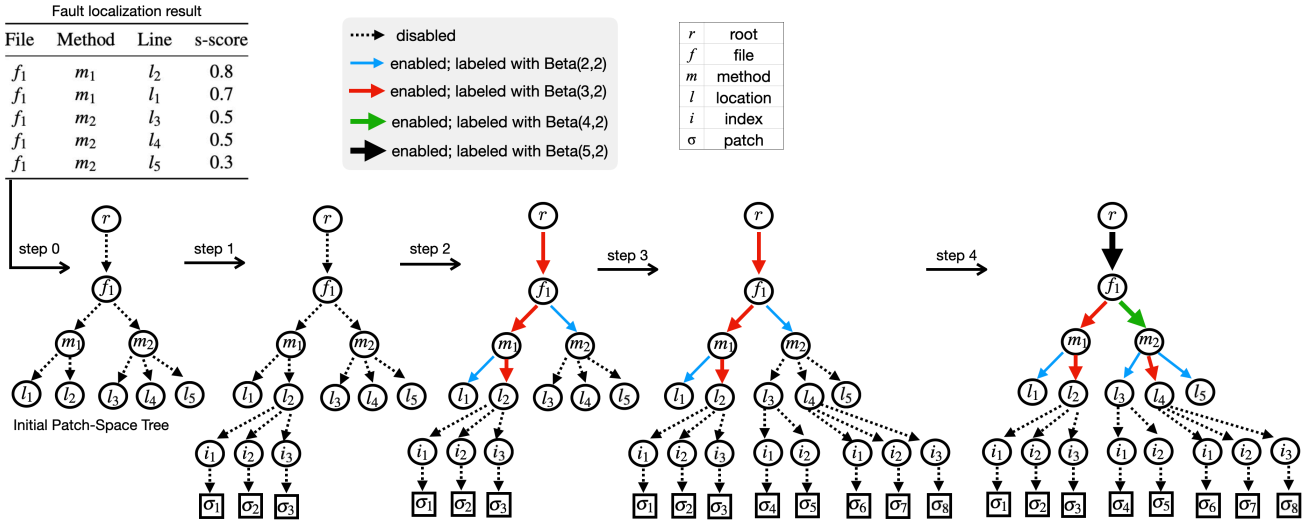
Algorithm 1 describes the CASINO patch scheduling algorithm and Figure 2 illustrates how it works step by step. It is assumed that fault localization is performed before the patch scheduling begins, as is the case in most APR tools. The top-left corner of the figure shows an example table summarizing the fault localization result, showing the suspiciousness scores (abbreviated as s-scores) of each program location l within method m in file f.

Step 0) Before running the CASINO algorithm, an initial patch-space tree is constructed. The leaves of this tree represent suspicious program locations l obtained from fault localization. At a later step when l is selected for repair, node l is expanded with the patch candidates that the APR tool generates at l. The initial patch-space tree $\mathcal{T}$ obtained at this step is passed as input to Algorithm 1. In our running example, we have 5 suspicious program locations, $l_1, l_2, \ldots, l_5$, which constitute the leaves of the patch-space tree. Initially, all edges of the patch-space tree are disabled (denoted with dotted lines in the figure), reflecting the fact that no runtime information is available to guide the traversal of the tree.

Step 1) CASINO selects a patch candidate for validation by navigating the patch space. The patch-space navigation is performed in two phases. In the first phase (lines 3–12 of Algorithm 1), CASINO traverses the patch-space tree top-down from the root node along enabled edges. Since all edges are initially disabled, this phase instantly ends without traversing any edge.

In the second phase (lines 14–22), CASINO looks for the most suspicious score s of the program locations within $\mathcal{T}(n)$—i.e., the patch-space *subtree* rooted at the current node n (line 15). Subsequently, it extracts a list $\mathbb{L}$ of patch candidates available at the locations with score s (line 16). Since the first phase has ended without traversing any edge, n refers to the root node of the patch-space tree and $\mathcal{T}(n)$ is identical to the entire patch-space tree. In our example, the program's most suspicious location is l_2 with a suspiciousness score of 0.8, and three patch candidates—σ_1, σ_2, and σ_3—are available in $\mathbb{L}$.

Most APR tools validate each patch candidate in a predetermined order. For example, learning-based tools validate patch candidates in the order of patch likelihood returned from a trained model [55, 68]. However, there is no guarantee that the predetermined order is optimal for finding interesting patches. To resolve this issue, CASINO allows for the random selection of patch candidates from $\mathbb{L}$ (line 19) as well as selection based on the predetermined order (line 21).



Fault localization result

File	Method	Line	s-score
f_1	m_1	l_2	0.8
f_1	m_1	l_1	0.7
f_1	m_2	l_3	0.5
f_1	m_2	l_4	0.5
f_1	m_2	l_5	0.3

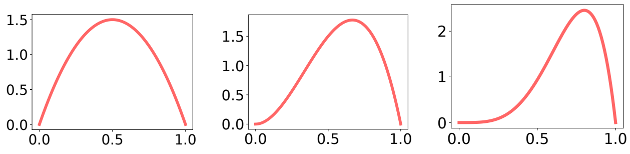
Figure 2: Overview of how the Casino patch scheduling algorithm works.

Step 2) Suppose that in the previous step (i.e., step 1), σ_2 was randomly chosen and after applying it to the program, a negative test passes, making σ_2 an interesting patch. When this occurs, Casino updates the relevant edges of the patch-space tree (line 31). The update is twofold. First, the edges along the path leading to σ_2, namely $r \rightarrow f_1 \rightarrow m_1 \rightarrow l_2$ are enabled. These edges are shown in red in the figure. Second, the neighboring edges of the red edges are also enabled. These edges are shown in blue in the figure. More specifically, given an enabled edge $n_1 \longrightarrow n_2$, all edges $n_1 \longrightarrow n_3$ where $n_3 \neq n_2$ are considered in the neighborhood and are enabled.

Each enabled edge is labeled with a probability distribution that represents the likelihood of finding an interesting patch along that edge. Casino uses the Beta distribution for this purpose. The Beta distribution, $Beta(\alpha, \beta)$, is a continuous probability distribution defined on the interval $[0, 1]$ and its shape is determined by two parameters, α and β. Figure 3 shows examples of Beta distributions with different α and β values.

When enabling initially disabled edges, Casino assigns $Beta(3, 2)$ to the edges leading to an interesting patch (the red edges in the figure), and $Beta(2, 2)$ to the neighboring edges (the blue edges in the figure). $Beta(3, 2)$ and $Beta(2, 2)$ are shown in Figures 3(b) and 3(a), respectively. Notice that as α becomes larger than β, the Beta distribution becomes more left-skewed (the mode shifts to the right). This implies that the red edges labeled with $Beta(3, 2)$—those that directly contribute to an interesting patch—are deemed more promising than the blue edges labeled with $Beta(2, 2)$—those neighboring the red edges.

Step 3) At the next iteration, Casino repeats the process of selecting a patch candidate. As before, the process starts with traversing the patch-space tree from the root node r. In our example, r has only one enabled edge, $r \longrightarrow f_1$, and the algorithm traverses to f_1. The f_1 node has two enabled edges, $f_1 \longrightarrow m_1$ and $f_1 \longrightarrow m_2$. To choose one of them, Casino samples a value in $[0, 1]$ from each Beta distribution associated with the enabled edges and selects the edge with the highest sample value (lines 6–11).



(a) $Beta(\alpha = 2, \beta = 2)$ **(b)** $Beta(\alpha = 3, \beta = 2)$ **(c)** $Beta(\alpha = 5, \beta = 2)$

Figure 3: Examples of Beta distributions, $Beta(\alpha, \beta)$

In our example, the edges $f_1 \longrightarrow m_1$ and $f_1 \longrightarrow m_2$ are associated with $Beta(3, 2)$ and $Beta(2, 2)$, respectively. Casino samples values $\hat{\theta}_1$ and $\hat{\theta}_2$ from $Beta(3, 2)$ and $Beta(2, 2)$, respectively. It is more likely that $\hat{\theta}_1 > \hat{\theta}_2$ than the other way around because $Beta(3, 2)$ is more left-skewed than $Beta(2, 2)$. See Figure 3. Thus, the edge $f_1 \longrightarrow m_1$ is more likely to be chosen than $f_1 \longrightarrow m_2$. However, $f_1 \longrightarrow m_2$ can still be chosen due to the stochastic nature of the selection process.

Suppose $f_1 \longrightarrow m_2$ is chosen and the algorithm traverses to m_2. Since m_2 has no outgoing enabled edge, phase I ends. Subsequently, in phase II, Casino looks for the most suspicious locations within $\mathcal{T}(m_2)$ and extracts a list of patch candidates available at those locations. $\mathcal{T}(m_2)$—the subtree rooted at m_2—contains l_3, l_4 and l_5, and l_3 and l_4 have the highest suspiciousness scores of 0.5 among them. Thus, Casino extracts a list of patch candidates available at l_3 and l_4 (i.e., $\sigma_4, \sigma_5, \ldots, \sigma_8$) and chooses one of them either randomly or based on the predetermined order, as explained in step 1.

Step 4) Suppose that in the previous step (i.e., step 3), σ_6 was randomly chosen and after applying it to the program, a negative test passes. Then the edges leading to σ_6, i.e., $r \longrightarrow f_1 \longrightarrow m_2 \longrightarrow l_4$, are enabled. In addition, a neighboring edge, $m_2 \longrightarrow l_3$, is also enabled.

Notice that $r \longrightarrow f_1$ and $f_1 \longrightarrow m_2$ were already enabled at a previous step when the first interesting patch, σ_2, was found. Now that an additional interesting patch (i.e., σ_6) has been found along these edges, Casino updates the Beta distributions associated with these edges. When a new interesting patch is found along an already enabled edge e, Casino increases the α value of the Beta distribution of e by 2^n, where n is the number of additional interesting patches found along e.

Relation to Fuzzing. The online scheduling policy of Casino is akin to that of fuzzing, as noted by its authors [21]. Similar to fuzzing, the repair process is guided by runtime information, specifically test results. The following summarizes the patch scheduling policy used in Casino:

Guidance Policy 1 (Blackbox Guidance Policy)**.** While traversing the patch-space tree, this policy gives higher priority to edges that are more likely to lead to the discovery of interesting patches. Note that the only runtime information used in this policy is whether a test passes or fails after applying a patch.

3 PROPOSED TECHNIQUE

The recent success of fuzzing is largely due to the use of greybox approaches. Unlike blackbox fuzzing which uses only the input-output behavior of the program, greybox fuzzing leverages the program's internal state such as branch coverage to guide the search for bug-revealing inputs. However, current online patch scheduling techniques such as Casino [21] and SeAPR [3] use a blackbox approach; they only use test results to guide the search for valid patches. In this work, we introduce a greybox patch scheduling technique to further enhance the efficiency of APR.

3.1 What to Observe: Critical Branches

Similar to coverage-guided greybox fuzzing, we leverage coverage information observed during program execution. However, fuzzing and APR have different goals and we utilize coverage information in a different way from fuzzing.

Our key insight is that the differences in the program's behavior between the original and validly patched programs can provide useful hints for discovering other valid patches. We capture the program's behavior using program spectra [15], specifically, branch-count spectra, which count the number of times each branch is taken during the execution of the program. Greybox fuzzers such as AFL similarly capture coverage information.

Suppose that while performing APR, an "interesting" patch that makes a previously failing test pass is found.[1] The behavioral differences between the original and patched programs can be represented as the differences in the branch counts. Those branches taken a different number of times between the two programs—which we call *critical branches* (see Definition 2)—contribute to causing the test to pass. In the remaining repair process, we monitor critical branches to guide the search for other valid patches.

Definition 2 (Critical Branch)**.** Given the original buggy program P_0 and its patched version $P_\star$ that passes a negative test t, a critical branch $b_\star$ with regard to t is an element of the following set:

$$\{b_\star \mid count[\![(P_\star, t)]\!](b_\star) \neq count[\![(P_0, t)]\!](b_\star)\}$$

where $count[\![(P, t)]\!](b_\star)$ denotes the number of times the branch $b_\star$ is taken during the execution of the program P for test t. Given multiple interesting patches, we combine the critical branches for each interesting patch by taking their union.

We distinguish between two types of critical branches—positive and negative critical branches—that are defined as follows.

Definition 3 (Positive Critical Branch)**.** Given the original buggy program P_0 and its patched version $P_\star$ that passes a negative test t, a positive critical branch $b^+_\star$ with regard to t is an element of the following set: $\{b^+_\star \mid count[\![(P_\star, t)]\!](b^+_\star) > count[\![(P_0, t)]\!](b^+_\star)\}$

Definition 4 (Negative Critical Branch)**.** Given the original buggy program P_0 and its patched version $P_\star$ that passes a negative test t, a negative critical branch $b^-_\star$ with regard to t is an element of the following set: $\{b^-_\star \mid count[\![(P_\star, t)]\!](b^-_\star) < count[\![(P_0, t)]\!](b^-_\star)\}$

3.2 How to Guide: Greybox Guidance Policy

In this subsection, we introduce our new patch scheduling algorithm, Gresino, which builds upon the Casino algorithm. As in Casino, Gresino performs online patch scheduling by traversing the patch-space tree. However, Gresino incorporates a new guidance policy for navigating the patch-space tree as described below:

Guidance Policy 2 (Greybox Guidance Policy)**.** While traversing the patch-space tree, this policy gives higher priority to edges that are more likely to lead to a patch candidate that *behaves similarly to the interesting patches* found earlier during the repair process.

We approximate the patch behavior based on critical branches as follows:

Definition 5 (Count-based Similarity of Patch Behavior)**.** Given a positive critical branch $b^+_\star$ extracted from an interesting patch $\sigma_\star$ and test t, a patch σ is considered similar to $\sigma_\star$ with regard to $b^+_\star$, if the following condition holds:

$$count[\![(P_\sigma, t)]\!](b^+_\star) > count[\![(P_0, t)]\!](b^+_\star)$$

where P_σ denotes the program obtained by applying the patch σ to the original buggy program P_0. This definition can be extended to negative critical branches $b^-_\star$ in a similar manner.

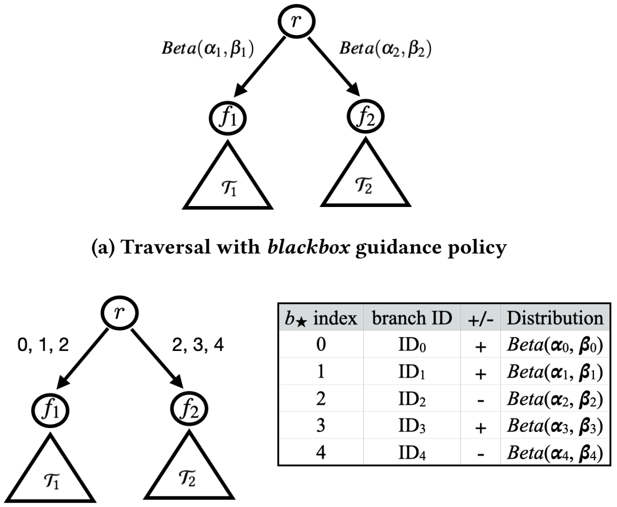
We now refine Guidance Policy 2 (Greybox Guidance Policy) based on Definition 5 as follows:

Guidance Policy 3 (Greybox Guidance Policy (Refined))**.** While traversing the patch-space tree, this policy gives higher priority to edges that are more likely to lead to a patch candidate that *shows count-based similarity to the interesting patches* found earlier during the repair process.

3.2.1 Blackbox vs. Greybox Guidance Policy. Figure 4 illustrates the difference between the blackbox and greybox guidance policies. While traversing the patch-space tree, the blackbox guidance policy (Figure 4(a)) consults the Beta distributions of the enabled edges to decide which edge to traverse next, as described in § 2.3. Notice that each edge is associated with a single Beta distribution which represents the likelihood of the edge leading to an interesting patch.

In contrast, when traversing the patch-space tree with the greybox guidance policy (Figure 4(b)), each edge is associated with a set of Beta distributions, one for each critical branch. For example, in Figure 4(b), the left edge is associated with the Beta distributions of the three critical branches indexed as 0, 1, and 2. Each Beta distribution represents the likelihood of the edge leading to a

[1] See Definition 1 for the definition of an interesting patch.



(a) Traversal with *blackbox* guidance policy

$b_\star$ index	branch ID	+/-	Distribution
0	ID0	+	$Beta(\boldsymbol{\alpha}_0, \boldsymbol{\beta}_0)$
1	ID1	+	$Beta(\boldsymbol{\alpha}_1, \boldsymbol{\beta}_1)$
2	ID2	-	$Beta(\boldsymbol{\alpha}_2, \boldsymbol{\beta}_2)$
3	ID3	+	$Beta(\boldsymbol{\alpha}_3, \boldsymbol{\beta}_3)$
4	ID4	-	$Beta(\boldsymbol{\alpha}_4, \boldsymbol{\beta}_4)$

(b) Traversal with *greybox* guidance policy. Each edge is labeled with the indices of the critical branches relevant to the edge. As shown in the table on the right, each critical branch is associated with a unique branch ID of the program under repair (the 2nd column), whether the critical branch is positive or negative (the 3rd column), and the Beta distribution of the critical branch (the 4th column).

Figure 4: Comparing blackbox and greybox guidance policies. While the figures are presented for the root level only, other levels are treated similarly.

patch candidate that shows count-based similarity to the interesting patches found earlier.

The blackbox and greybox guidance policies differ in performing the following operations on the patch-space tree:

- **Tree Traversal**: Unlike the blackbox guidance policy, multiple Beta distributions can be available for an edge under consideration. We randomly select one of them in the current implementation for simplicity. For example, in Figure 4(b), if index 0 and 4 are randomly selected for the left and right edges, respectively, then $Beta(\alpha_0, \beta_0)$ and $Beta(\alpha_4, \beta_4)$ are used to decide which edge to traverse. Alterantively, it could also be possible to consider all Beta distributions associated with the critical branches – for instance, considering indices 0, 1, and 2 for the left edge and 2, 3, and 4 for the right edge. However, to keep the runtime overhead low, we opt for a simple random selection.
- **Updating Beta Distributions**: In both policies, the Beta distributions are updated when a new interesting patch is found. However, the two policies update different Beta distributions. In the greybox guidance policy, we first identify the critical branches relevant to the interesting patch and then update the Beta distributions associated with these critical branches. For example, in Figure 4(b), suppose a new interesting patch identifies two critical branches b_0^+ and b_2^-, where b_i represents the critical branch at index i and the superscript + (or −) indicates that the critical branch is positive (or negative). For b_0^+, its associated Beta distribution, $Beta(\alpha_0, \beta_0)$, is updated into $Beta(\alpha_0 + 1, \beta_0)$. Similarly, $Beta(\alpha_2, \beta_2)$ is updated into $Beta(\alpha_2 + 1, \beta_2)$ for b_2^-.
- **Initializing Beta Distributions**: Similar to the update operation, the two policies initialize different Beta distributions. Suppose a fresh critical branch b_4^+ is obtained. Since b_4^+ is a fresh critical branch (previously, only b_4^- was identified), a fresh Beta distribution, $Beta(1, 1)$, is initialized for b_4^+.

3.2.2 *Potential Benefits of Greybox Guidance Policy.*

Our greybox guidance policy offers the following potential benefits over the blackbox guidance policy:

Potential Benefit 1 (More Informative Guidance): In Figure 4(a), suppose that N interesting patches have been found both in the left subtree (i.e., $\mathcal{T}_1$) and the right subtree (i.e., $\mathcal{T}_2$). In the blackbox guidance policy, both edges of node r would have equal chances of being selected. However, the execution patterns of the interesting patches may differ between the two subtrees. For example, consider the following scenario: The N interesting patches found in $\mathcal{T}_1$ observe an *increase* in the count of a sole critical branch $b_\star$ compared to that in the original program. Meanwhile, the N interesting patches found in $\mathcal{T}_2$ observe an *increase* only $N/2$ times, and for the remaining $N/2$ times, the count of $b_\star$ *decreases*. In the greybox guidance policy, the edge leading to the left subtree (i.e., $\mathcal{T}_1$) would be more likely to be selected, as it is more likely to lead to a patch candidate that behaves similarly to the interesting patches found earlier. As illustrated in this example, *the greybox policy can guide the search efficiently by exploiting richer information than is available in the blackbox policy.*

Potential Benefit 2 (Program Dependence Awareness): Consider an interesting patch $\sigma_\star$ that modifies a method m_1. Suppose applying $\sigma_\star$ changes the number of times a critical branch $b_\star \in m_2$ is taken. Note that modifying m_1 may affect the execution of another method m_2. In the blackbox guidance policy, the dependence between m_1 and m_2 is not considered. The discovery of $\sigma_\star$ only affects the Beta distribution of the edge associated with m_1. In contrast, the greybox guidance policy can accommodate the dependence between m_1 and m_2 to the search process. This is because the Beta distributions of the critical branches are shared across multiple edges. For example, in Figure 4(b), a critical branch indexed as 2 is shared between the left and right edges. In summary, *the greybox policy can guide the search efficiently by considering the program dependence information observed via critical branches.*

3.2.3 *Adjustive Guide between Blackbox and Greybox Policies.*

Despite the potential benefits of the greybox guidance policy, it does not necessarily mean that the greybox guidance policy is always superior to the blackbox guidance policy. Therefore, we propose a hybrid approach that combines the blackbox and greybox guidance policies. In Gresino, the traversal of the patch-space tree is started with the blackbox guidance policy of Casino. While traversing the tree, Gresino allows switching to the greybox guidance policy at a random node of the patch-space tree. This switch is allowed to occur only if a set of critical branches $\mathbb{CB}$ is not empty. $\mathbb{CB}$ is initially empty and is updated whenever an interesting patch $\sigma_\star$ is found. Specifically, it is extended with the critical branches relevant to $\sigma_\star$. If $\mathbb{CB}$ is not empty, the switch to the greybox guidance policy at node n is made with a probability λ, which we increase as

Algorithm 2 Gresino Patch Scheduling Algorithm. Changes to Casino are highlighted in blue.

```
Input: $P$: Program to be repaired
Input: Initial patch-space tree $\mathcal{T}$
Input: $\mathbb{TS}$: Test-suite consisting of a negative test and positive tests
Output: $\mathbb{D}_{\text{patches}}$
1: $\mathbb{D}_{\text{patches}} \leftarrow \emptyset$ // $\mathbb{D}_{\text{patches}}$: Set of detected patches
2: $\mathbb{CB} \leftarrow \emptyset$ // $\mathbb{CB}$: Set of critical branches
3: while $t_{\text{elapsed}} < t_{\text{limit}} \wedge \textsc{Continue}()$ do
4:     // Phase I: vertical navigation with blackbox guidance policy
5:     $n := \textsc{Root}(\mathcal{T})$ // Start patch-tree traversal from the root node
6:     while $n$ has an enabled outgoing edge do
7:         if $\mathbb{CB} \neq \emptyset$ then
8:             Toss a coin with success probability $\lambda$
9:             if success then break // jump to line 20
10:            end if
11:        end if
12:        for each enabled edge $e$ of $n$ do
13:            $Beta(\alpha_k, \beta_k) \leftarrow \textsc{Label}(e)$
14:            Draw $\hat{\theta}_k$ according to $Beta(\alpha_k, \beta_k)$
15:        end for
16:        $e := \arg\max_k \hat{\theta}_k$ // Select the $k$-th edge having the maximum $\hat{\theta}_k$
17:        $n := \textsc{Traverse}(n, e)$ // Traverse edge $n \xrightarrow{e} n'$ and $n := n'$
18:    end while
19:    // Phase II
20:    $s \leftarrow \textsc{MostSuspiciousScore}(\mathcal{T}(n))$
21:    Toss a coin with success probability $\epsilon$
22:    if success then
23:        if $\mathbb{CB} \neq \emptyset$ then
24:            // vertical navigation with greybox guidance policy
25:            while $n$ has an enabled outgoing edge do
26:                $b_\star \leftarrow \textsc{ChooseAtRandom}(\mathbb{CB})$
27:                for each outgoing edge $e$ of $n$ do
28:                    if $s \in \textsc{Score}(e)$ then
29:                        $Beta(\alpha_k, \beta_k) \leftarrow \textsc{Label}(e)[b_\star]$
30:                        Draw $\hat{\theta}_k$ according to $Beta(\alpha_k, \beta_k)$
31:                    end if
32:                end for
33:                $e := \arg\max_k \hat{\theta}_k$
34:                $n := \textsc{Traverse}(n, e)$
35:            end while
36:            $\sigma_{next} \leftarrow n$
37:        else
38:            $\sigma_{next} \leftarrow \textsc{ChooseAtRandom}(\textsc{Candidates}(s))$
39:        end if
40:    else
41:        $\sigma_{next} \leftarrow \textsc{ChooseInPredeterminedOrder}(\textsc{Candidates}(s))$
42:    end if
43:
44:    $\texttt{patch}, \texttt{execinfos} \leftarrow \textsc{PatchEval}(P, \sigma_{next}, \mathbb{TS})$
45:    if $\textsc{IsValid}(\texttt{patch}, \texttt{execinfos})$ then $\mathbb{D}_{\text{patches}} \leftarrow \mathbb{D}_{\text{patches}} \cup \{\texttt{patch}\}$
46:    end if
47:    if $\textsc{IsInteresting}(\texttt{patch}, \texttt{execinfos})$ then
48:        $\mathbb{CB} \leftarrow \mathbb{CB} \cup \textsc{ExtractCriticalBranches}(\texttt{execinfos})$
49:    end if
50:
51:    $\textsc{Update}(\texttt{execinfo}, \mathcal{T}, \mathbb{L}_{MSU}, \sigma_{next}, \mathbb{CB})$
52: end while
```

the use of the blackbox guidance policy at node n fails to find an interesting patch more number of times. Once the switch is made, the remaining traversal of the tree is performed with the greybox guidance policy.

3.2.4 Putting All Together. Algorithm 2 shows the Gresino algorithm, where the differences between Casino and Gresino are highlighted in blue. The input-output interface of Gresino is identical to that of Casino. The algorithm takes as input a program P to be repaired, the initial patch-space tree $\mathcal{T}$ and a test-suite $\mathbb{TS}$, and produces as output a set of detected patches.[2] In Algorithm 2, we assume that $\mathbb{TS}$ has a single negative test for simplicity, but the algorithm can be easily extended to handle multiple negative tests.

At the start, Gresino runs in the same manner as Casino, as initially there are no critical branches to consider. The algorithm maintains a set of critical branches $\mathbb{CB}$ (initially $\emptyset$) and extends it with the critical branches satisfying Definition 2, whenever an interesting patch is found (lines 47–49).

Gresino starts to work differently from Casino once $\mathbb{CB}$ becomes non-empty. Given a non-empty $\mathbb{CB}$, tree traversal using the blackbox guidance policy is allowed to switch to the greybox guidance policy with a probability λ (line 7–11). Then, tree traversal with the greybox guidance policy is performed in lines 24–36. The traversal is performed all the way down to the leaf node, which corresponds to a patch candidate to be evaluated (line 36). While traversing the tree with the greybox guidance policy, we do not distinguish between enabled and disabled edges. Thus, the traversal is guaranteed to reach a leaf node. However, at each intermediate node of the tree, Gresino considers only the edges associated with the most suspicious score s (lines 28–31).[3] For example, consider the rightmost patch-space tree shown in Figure 2. Suppose that the switch to the greybox guidance policy is made at node m_2. In this case, the most suspicious score in $\mathcal{T}(m_2)$—the subtree rooted at m_2—is 0.5 associated with l_3 and l_4. Thus, only the edges leading to l_3 and l_4 are considered for traversal, while the other edges such as $m_2 \longrightarrow l_5$ are ignored.

4 EXPERIMENTAL DESIGN AND SETUP

4.1 Research Questions

To evaluate our approach Gresino, we ask the following four research questions:

- **RQ1 (Search Efficiency)**: How efficiently does Gresino find *valid* patches? We call a patch valid when it passes all available tests. A valid patch is also referred to as a plausible patch in APR literature.
- **RQ2 (Recall)**: In how many versions does Gresino find *acceptable* patches? We call a patch acceptable when it is valid and is considered correct. Refer to § 4.2 for the definition of acceptable patches.
- **RQ3 (Ablation Study)**: Gresino uses both the greybox and blackbox guidance policies. How does the performance of Gresino change when one of the guidance policies is removed?
- **RQ4 (Generalizability)**: Does the APR efficiency of Gresino generalize to a fresh benchmark?

[2] For a description of how the initial patch-space tree is constructed, see step 0 of § 2.3.

[3] $\textsc{Score}(e)$ returns a set of the suspicious scores of the locations reachable through e.

4.2 Patch Correctness

In APR literature, patch correctness has been determined either manually or by using an additional test suite, typically generated with an automated test generation tool. The former is prone to subjectivity and difficult to use when there is a large number of patches to review. On the other hand, the latter is more scalable but it may fail to identify some incorrect patches.

In our experiments, we obtain tens of thousands of valid patches (31,759 patches) and resort to an automatic approach. To mitigate the risk of failing to identify incorrect patches, we employ a differential testing tool, DiffTGen [58], which is specifically designed to detect incorrect patches. If DiffTGen finds no semantic difference between an obtained patch and the corresponding correct version provided in the benchmark, we classify the patch as *acceptable*. The same approach was used to evaluate Casino [21].

4.3 Implementation of Gresino

Gresino is implemented on top of SimAPR [21, 22], which provides a framework to evaluate patch scheduling algorithms. SimAPR contains the implementation of Casino, which we have extended to develop Gresino. However, unlike Casino, Gresino's greybox approach requires instrumenting each branch b of the program under repair to count the number of times b is executed. Our implementation instruments Java bytecode using ASM [9]. When a patch is applied, only the modified bytecode is re-instrumented. Our instrumentation is similar to that of AFL [4, 67], which also measures branch hit counts with low overhead.

4.4 Evaluation Methods

In this section, we describe how we evaluate the research questions.

RQ1 (Search Efficiency). We measure the search efficiency of Gresino by examining how the number of valid patches increases with the increase of (1) the number of evaluated patch candidates and (2) time. The former assesses the algorithmic efficiency, while the latter considers time overhead. For comparison, the efficiency of Casino is also measured in the same way.

Both Casino and Gresino, as patch-scheduling algorithms, can be applied to most template-based or learning-based APR tools. We apply both algorithms to the eight APR tools described in § 4.5. For this purpose, we utilize the SimAPR framework [21, 22]. This framework enables APR to be conducted in diverse settings with a user-chosen patch-scheduling algorithm and an APR tool. It also ensures that the same fault localization results of buggy versions are used across different settings, irrespective of the chosen patch-scheduling algorithm and APR tool.

Considering the stochastic nature of Gresino and Casino, we run each algorithm 10 times with unique random seeds. We also measure the efficiency of each APR tool without applying Gresino or Casino. This is to compare the efficiency of Gresino with that of the original scheduling algorithm of each APR tool. These original scheduling algorithms are deterministic and we run them only once. For each repair session, we evaluate up to 3,000 patch candidates in all three settings, i.e., Gresino, Casino, and the original scheduling algorithm of each APR tool.

RQ2 (Recall). To measure recall, we count the number of buggy versions for which Gresino finds acceptable patches. We consider a scenario where automatically generated valid patches are ranked using a patch-ranking technique [5, 14, 49, 51, 59, 61]. We investigate how often an acceptable patch is ranked within the top-N, among the valid patches collected during the 3,000 trial limit. For patch ranking, we utilize ODS [61], the state-of-the-art machine-learning-based patch classification tool.

RQ3 (Ablation Study). We compare the performance of Gresino, which combines greybox and blackbox guidance policies, against its two variations: one without the greybox guidance policy and the other without the blackbox guidance policy. Note that the former is equivalent to Casino. In the second variant that uses only the greybox guidance policy, tree traversal always immediately switches to the greybox guidance policy without using the blackbox policy, once a set of critical branches becomes non-empty. In Algorithm 2, this can be achieved by setting the value of λ to 1 in line 8.

RQ4 (Generalizability). We evaluate Gresino with a fresh benchmark not used in the evaluations for RQ1, RQ2, and RQ3, as detailed in § 4.5.

4.5 Evaluation Dataset and Studied APR Tools

To evaluate the first three research questions, we use Defects4J v1.2 [20], given its widespread adoption in APR studies [21, 23, 29–32]. Defects4J v1.2 comprises 395 bugs from six open-source software projects. To assess the generalizability of our approach in RQ4, we extract from Defects4J v2.0 a total of 440 bugs that are not included in Defects4J v1.2. Out of 835 bugs in Defects4J v2.0, the 395 bugs also present in Defects4J v1.2 are excluded from the second benchmark.

We consider all six APR tools provided by the SimAPR framework: (1) TBar [31], (2) Avatar [30], (3) FixMiner [23], (4) kPar [29], (5) Recoder [68], and (6) AlphaRepair [56]. Additionally, we include two state-of-the-art APR tools: SelfAPR [62], which uses a neural model trained for program repair, and SRepair [57], the latest LLM-based tool. Altogether, these eight APR tools cover the template-based approach (TBar, Avatar, FixMiner, and kPar) and the learning-based approach (Recoder, AlphaRepair, SelfAPR, and SRepair), which are the two most popular approaches in recent APR studies. The considered APR tools repair a single program location, as is common in APR studies. The consideration of less-studied multi-location APR tools [47, 63] is left for future work.

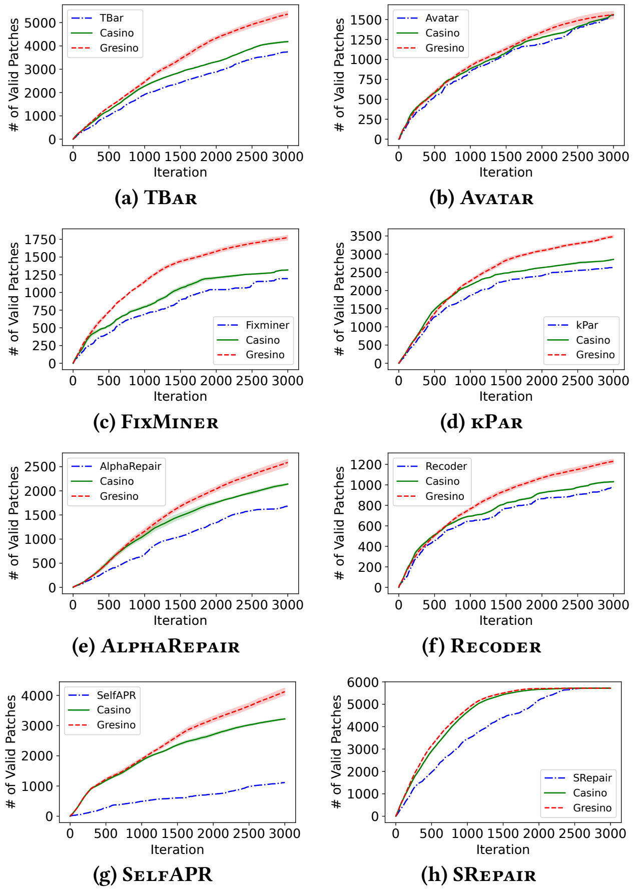
4.6 Experimental Environment

For each APR tool, we ran Gresino, Casino, and the original scheduling algorithm of the tool on the *same* machine. For all APR tools except for kPar, we used a machine equipped with AMD EPYC 2.6GHz CPUs (1024GB RAM). For kPar, we used a machine equipped with Intel Xeon Gold 3GHz CPUs (128GB RAM). Note that we compare the performance of each patch-scheduling algorithm only *within the same machine, without conducting any cross-machine comparisons.* We used Ubuntu 22.04 LTS and Java 1.8 across all experiments.

5 EXPERIMENTAL RESULTS

5.1 RQ1: Search Efficiency

Figure 5 shows the number of valid patches detected by each APR tool on the Y-axis over the number of iterations on the X-axis;



(a) TBar

(b) Avatar

(c) FixMiner

(d) kPar

(e) AlphaRepair

(f) Recoder

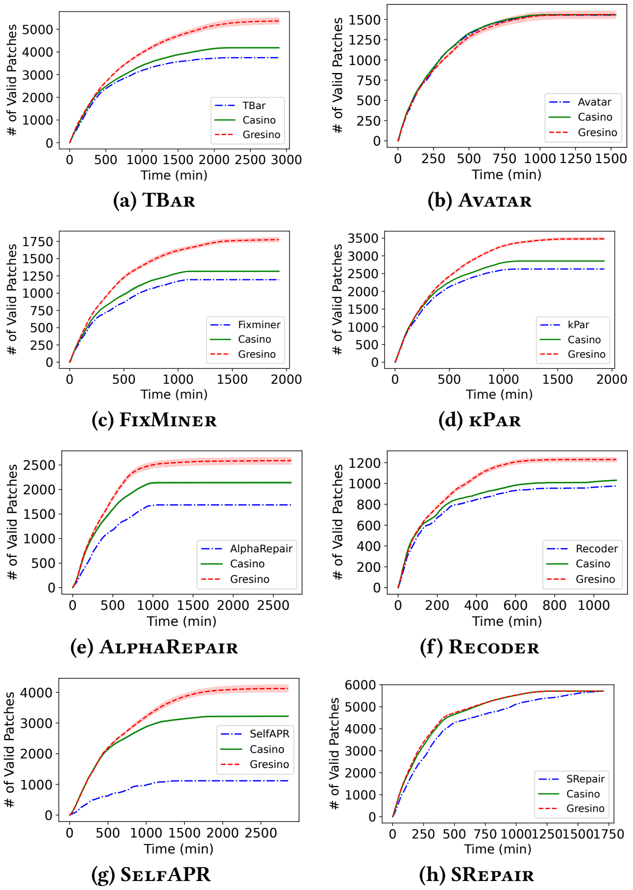
(g) SelfAPR

(h) SRepair

Figure 5: Results for RQ1 on search efficiency, shown in terms of *iterations* required to find valid patches.

each iteration comprises Step 2-1 in Figure 1—i.e., patch candidate selection—and Step 2-2—i.e., patch candidate evaluation. The results for the three algorithms are shown: Gresino in red, Casino in green, and the original patch-scheduling algorithm of the APR tool in blue. The plots shown in the figure are cumulative, meaning that the Y-axis value at each iteration point on the X-axis indicates the total number of valid patches found up to that iteration across all buggy versions in the benchmark. For Gresino and Casino, we conducted 10 runs and present the mean values of the results with 95% confidence intervals (CI) illustrated as shades around the lines. The thin shades indicate that the variance is small across the runs. The plots in the figure show that Gresino outperforms Casino and the original algorithms in all APR tools, except for Avatar and SRepair where the two algorithms perform similarly.

While Figure 5 compares *algorithmic* performance of patch scheduling algorithms, it does not consider the runtime overhead. In contrast, Figure 6 compares the *runtime* performance of patch scheduling algorithms by accounting for the execution time. As before, the plots are cumulative, with the Y-axis value at each time point on the X-axis indicating the total number of valid patches found up to that time across all buggy versions in the benchmark. For each plot, the maximum value of the X-axis corresponds to the time taken to complete 3,000 iterations. The plots in Figure 6 are based on the same data used in Figure 5. Once again, Gresino outperforms Casino and the original algorithms in most of the APR tools, suggesting that the runtime overhead for Gresino's bytecode instrumentation and the execution of instrumented programs does not hurt its search efficiency. Table 1 shows the time overhead of Gresino compared to the original patch scheduling algorithm of the APR tools.



(a) TBar

(b) Avatar

(c) FixMiner

(d) kPar

(e) AlphaRepair

(f) Recoder

(g) SelfAPR

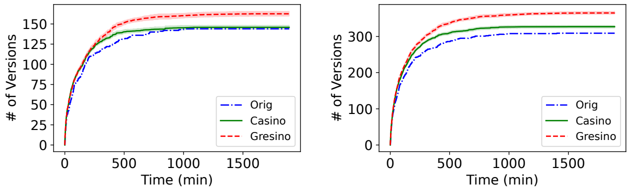
(h) SRepair

Figure 6: Results for RQ1 on search efficiency, shown in terms of *time* taken to find valid patches.

Tool	Overhead	Tool	Overhead
TBar	33.12 %	AlphaRepair	10.26 %
Avatar	35.35 %	Recoder	5.41 %
FixMiner	11.05 %	SelfAPR	25.64 %
kPar	29.81 %	SRepair	27.56 %

Table 1: Time overhead of Gresino compared to the original patch scheduling algorithm of the APR tools.

> ***RQ1:*** *In six out of eight APR tools, Gresino tends to find valid patches faster than Casino and the original algorithms, accumulating more valid patches over time. In the remaining two tools, Gresino and Casino perform similarly.*



(a) Top-1 (b) Top-5

Figure 7: Results for RQ2 on recall, shown in terms of the number of buggy versions for which acceptable patches are contained within the top-N valid patches.

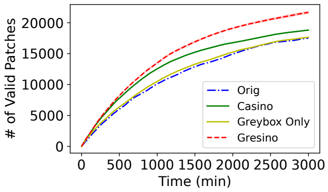


Figure 8: Results for RQ3 on the ablation study.

5.2 RQ2: Recall

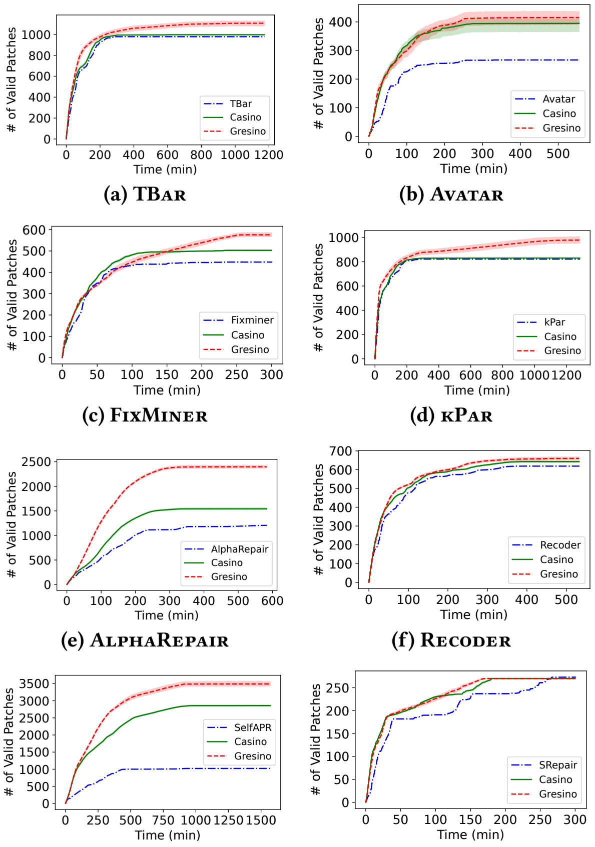
Figure 7 shows the number of buggy versions for which acceptable patches are contained within the top-N valid patches. The plots are cumulative, with the Y-axis representing the total number of buggy versions for which acceptable patches are found up to each time point on the X-axis across all eight APR tools we consider. The results for Gresino are shown in red, Casino in green, and the original patch-scheduling algorithms of the APR tools in blue. For Gresino and Casino, we conducted 10 runs and present the mean values of the results, with 95% confidence intervals (CI) indicated by shaded areas around the lines. We observe the following result:

> ***RQ2:*** *Gresino finds acceptable patches in a greater number of buggy versions than Casino and the original patch-scheduling algorithms of eight APR tools.*

5.3 RQ3: Ablation Study

Figure 8 presents a cumulative plot that aggregates the results of the eight APR tools. The Y-axis represents the total number of valid patches found up to each time point on the X-axis across all buggy versions in the benchmark and all eight APR tools. The results for the four algorithms are shown: Gresino in red, Casino in green, the original patch-scheduling algorithms of the APR tools in blue, and Gresino's variation that employs only the greybox guidance policy in yellow. Recall that Casino uses only the blackbox guidance policy. Among these, Gresino shows the best performance.

> ***RQ3:*** *Gresino performs better than its two variations, which do not use the greybox or blackbox guidance policy, respectively. Our results suggest that in our algorithmic setting, the search efficiency is higher when both greybox and blackbox guidance policies are used than when only the greybox or blackbox policy is used.*



(a) TBar (b) Avatar (c) FixMiner (d) kPar (e) AlphaRepair (f) Recoder (g) SelfAPR (h) SRepair

Figure 9: Results for RQ4

5.4 RQ4: Generalizability

Figure 9 presents the search efficiency results for the 440 new bugs in Defects4J v.2.0. As in RQ1, cumulative plots are depicted. The following result is observed:

> ***RQ4:*** *Similar overall patterns are observed between the two benchmarks.*

6 DISCUSSION AND THREATS TO VALIDITY

Inspiration from Fuzzing. Similar to [21], our work is inspired by fuzzing, in particular, coverage-guided fuzzing. The underlying intuition of coverage-guided fuzzing is that covering more paths in the program is likely to reveal more bugs. This is a useful intuition because guiding the fuzzer to cover more paths can be done effectively as shown in numerous works. In contrast, guiding the fuzzer directly towards bugs is difficult, due to their sparse occurrence. *Similar to coverage-guided fuzzing, Gresino is designed based on the intuition that the discovery of more plausible patches is likely to increase the chance of finding a correct patch.*

Just as in fuzzing, where discovering new paths does not necessarily lead to new bugs, there is no guarantee that finding more "interesting" patches will lead to correct patches. Nevertheless, our preliminary investigation suggests a tendency for interesting patches to share similar behavioral characteristics with correct patches. We compared how often interesting patches and correct (i.e., developer-written) patches exhibited the same critical branches with the same direction (i.e., positive or negative). We found that at the critical branches of the interesting patches, the correct patches are about four times more likely to exhibit the same change patterns—i.e., either positive or negative—than not.

When Gresino performs better than Casino. To understand when Gresino outperforms Casino, we take a closer look at two APR tools: FixMiner and Avatar, the best and worst performers in our experiments, respectively. As described in § 3.2.2, Gresino can consider program dependence information via the critical branches shared between different patches. To investigate how often this sharing occurs, we extract the following three values: (A) the total number of critical branches, (B) the number of times the α values of the Beta distributions associated with the critical branches are updated, and (C) the number of times the α values of the Beta distributions used for the blackbox guidance policy are updated while running the Gresino algorithm. Then, we compute the ratio between (B/A) and C. This ratio is 1.87 for Avatar and 2.35 for FixMiner, indicating that in FixMiner, the critical branches are shared more frequently than in Avatar.

We also hypothesize that Gresino performs better when the edges of the patch-space tree are more balanced in terms of their probabilities of being chosen while the blackbox guidance policy is used. We conjecture that richer information provided by the greybox guidance policy can help with discerning more promising edges, as discussed in § 3.2.2. To investigate this hypothesis, we compare the entropies of the α values of the Beta distributions used for the blackbox guidance policy. Pearson's moment coefficient of skewness is used to measure the skewness of the distribution of the entropies. In FixMiner, this coefficient is 1.77, while in Avatar, it is 2.23. A smaller positive coefficient indicates a less right-skewed distribution. This suggests that in FixMiner, the entropy tends to be higher—more balanced—than in Avatar, upholding our hypothesis. We acknowledge that the above analyses are preliminary and that we leave a more thorough investigation for future work.

For SRepair, while using the blackbox approach of Casino brings substantial performance improvement, employing Gresino results in only a marginal improvement over Casino. We conjecture that this is related to the abundant availability of interesting patches in the patch space of SRepair. Recall that Gresino employs both blackbox and greybox guidance policies, and the greybox guidance policy is only selectively activated; it is more likely to be activated when the blackbox guidance policy does not perform well. For SRepair where the blackbox guidance policy already performs well, Gresino's greybox guidance policy is less likely to be activated, resulting in similar performance between Gresino and Casino.

Threats to Validity. We also acknowledge the following threats to the validity of our study. First, caution should be exercised when generalizing our findings to other APR tools and benchmarks, although to mitigate this threat, we evaluated Gresino with diverse APR tools and benchmarks.

The second threat comes from the fact that we collected "acceptable" patches using the DiffTGen tool. While DiffTGen is specialized for patch correctness validation, it may fail to detect some incorrect patches. However, manually validating tens of thousands of patches is likely to pose even greater threats to validity due to human error. Acceptable patches, if they turn out to be incorrect, can be viewed as those whose flaws are not easily detectable.

7 RELATED WORK

We have described the closest related work, Casino [21], SeAPR [3], and GenProg [25] in § 1, all of which can be categorized as blackbox approaches. In contrast, Gresino uses a greybox approach. Similar to greybox fuzzing, Gresino monitors branch hit counts during program execution. Meanwhile, semantics-based approaches [2, 38, 39, 41, 64, 65] can be considered a whitebox approach. For example, DirectFix [38] transforms a buggy program into a logical formula φ and then looks for a patched formula φ' using an SMT-solver-based synthesizer. Other semantics-based approaches perform less heavyweight analysis than DirectFix, but logical constraints still play a central role in them.

Unlike Gresino, there have been some approaches to prioritize patch candidates *offline*, i.e., before the search for patches starts. Prioritization is done based on various information such as code context [53], program contracts [43], code comments [60], Q&A sites [11], and existing patches [18, 24, 34]. These approaches typically build a statistical model and use it to guide the search for a valid patch. Recent deep-learning-based APR [8, 19, 27, 28, 35, 56, 57, 62, 68] can be viewed as an extension of this line of work. Online patch-scheduling algorithms like Gresino can be used in conjunction with these offline patch prioritization techniques.

Patch prioritization techniques can also be combined with patch-space reduction methods. These methods exclude from the patch space harmful patch patterns (a.k.a., anti-patterns) [48], patch patterns involving intractably large search space [33], and patch candidates belonging to already covered test-equivalent classes [37].

Running a test-driven APR tool typically involves the repetitive execution of tests. To reduce testing cost, test case prioritization [46] (which prioritizes tests that are likely to reveal test failures early) and regression test selection [10, 45] (which reduce the number of tests to run) have been developed and used in some APR tools [40, 44]. These techniques are orthogonal and complementary to Gresino.

8 CONCLUSION AND FUTURE WORK

In this paper, we have proposed a novel patch-scheduling algorithm, Gresino. In contrast to Casino which uses only the "blackbox" information (i.e., test results), Gresino also leverages the "greybox" information of the program. Specifically, Gresino monitors the hit counts of branches observed during the execution of the program and uses them to guide the search for a valid patch. We have shown the efficacy of Gresino through extensive experiments on the Defects4J benchmark and eight APR tools.

ACKNOWLEDGEMENTS

This work was partly supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No.2021R1A5A1021944) and the Institute for Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2021-0-01001, RS-2024-00337414).

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)*. IEEE, Windsor, UK, 89–98. https://doi.org/10.1109/TAIC.PART.2007.13

[2] Umair Z. Ahmed, Zhiyu Fan, Jooyong Yi, Omar I. Al-Bataineh, and Abhik Roychoudhury. 2022. Verifix: Verified Repair of Programming Assignments. *ACM Transactions on Software Engineering and Methodology* 31, 4 (Oct. 2022), 1–31. https://doi.org/10.1145/3510418

[3] Samuel Benton, Yuntong Xie, Lan Lu, Mengshi Zhang, Xia Li, and Lingming Zhang. 2022. Towards boosting patch execution on-the-fly. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2165–2176. https://doi.org/10.1145/3510003.3510117

[4] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based Greybox Fuzzing as Markov Chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, Vienna Austria, 1032–1043. https://doi.org/10.1145/2976749.2978428

[5] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding Automatically-Generated Patches Through Symbolic Invariant Differences. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, San Diego, CA, USA, 411–414. https://doi.org/10.1109/ASE.2019.00046

[6] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Urbana, IL, 637–647. https://doi.org/10.1109/ASE.2017.8115674

[7] Liushan Chen, Yu Pei, and Carlo A. Furia. 2021. Contract-Based Program Repair Without The Contracts: An Extended Study. *IEEE Transactions on Software Engineering* 47, 12 (Dec. 2021), 2841–2857. https://doi.org/10.1109/TSE.2020.2970009

[8] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noel Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2021. SEQUENCER: Sequence-to-Sequence Learning for End-to-End Program Repair. *IEEE Transactions on Software Engineering* (2021), 1–1. https://doi.org/10.1109/TSE.2019.2940179

[9] R. Lenglet E. Bruneton and T. Coupaye. Accessed on June, 2024. ASM. https://asm.ow2.io/.

[10] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information and Software Technology* 52, 1 (2010), 14–30. Publisher: Elsevier.

[11] Qing Gao, Hansheng Zhang, Jie Wang, Yingfei Xiong, Lu Zhang, and Hong Mei. 2015. Fixing Recurring Crash Bugs via Analyzing Q&amp;A Sites (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Lincoln, NE, USA, 307–318. https://doi.org/10.1109/ASE.2015.81

[12] 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*. ACM, Beijing China, 8–18. https://doi.org/10.1145/3293882.3330558

[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*. ACM, Beijing China, 19–30. https://doi.org/10.1145/3293882.3330559

[14] Ali Ghanbari and Andrian Marcus. 2022. Patch correctness assessment in automated program repair based on the impact of patches on production and test code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual South Korea, 654–665. https://doi.org/10.1145/3533767.3534368

[15] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. In *Proceedings of the SIGPLAN/SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE '98, Montreal, Canada, June 16, 1998*, Thomas Ball, Frank Tip, and A. Michael Berman (Eds.). ACM, 83–90. https://doi.org/10.1145/277631.277647

[16] Christian Holler, Kim Herzig, Andreas Zeller, and others. 2012. Fuzzing with Code Fragments.. In *USENIX Security Symposium*. 445–458.

[17] Elkhan Ismayilzada, Md Mazba Ur Rahman, Dongsun Kim, and Jooyong Yi. 2024. Poracle: Testing Patches under Preservation Conditions to Combat the Overfitting Problem of Program Repair. *ACM Transactions on Software Engineering and Methodology* 33, 2 (Feb. 2024), 1–39. https://doi.org/10.1145/3625293

[18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Amsterdam Netherlands, 298–309. https://doi.org/10.1145/3213846.3213871

[19] Nan Jiang, Thibaud Lutellier, and Lin Tan. 2021. CURE: Code-Aware Neural Machine Translation for Automatic Program Repair. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, Madrid, ES, 1161–1173. https://doi.org/10.1109/ICSE43902.2021.00107

[20] René Just, Darioush Jalali, and Michael D. Ernst. 2014. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014*. ACM Press, San Jose, CA, USA, 437–440. https://doi.org/10.1145/2610384.2628055

[21] YoungJae Kim, Seungheon Han, Askar Yeltayuly Khamit, and Jooyong Yi. 2023. Automated Program Repair from Fuzzing Perspective. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Seattle WA USA, 854–866. https://doi.org/10.1145/3597926.3598101

[22] YoungJae Kim, Seungheon Han, and Jooyong Yi. 2023. SimAPR. https://github.com/UNIST-LOFT/SimAPR.

[23] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. FixMiner: Mining Relevant Fix Patterns for Automated Program Repair. *Empirical Software Engineering* 25, 3 (May 2020), 1980–2024. https://doi.org/10.1007/s10664-019-09780-z arXiv:1810.01791 [cs].

[24] Xuan Bach D. Le, David Lo, and Claire Le Goues. 2016. History Driven Program Repair. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, Suita, 213–224. https://doi.org/10.1109/SANER.2016.76

[25] Claire Le Goues, Michael Dewey-Vogt, Stephanie Forrest, and Westley Weimer. 2012. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, Zurich, 3–13. https://doi.org/10.1109/ICSE.2012.6227211

[26] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-Preserving Test Repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 217–227. https://doi.org/10.1109/ICST.2019.00030

[27] 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*. ACM, Seoul South Korea, 602–614. https://doi.org/10.1145/3377811.3380345

[28] 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*. ACM, Pittsburgh Pennsylvania, 511–523. https://doi.org/10.1145/3510003.3510177

[29] Kui Liu, Anil Koyuncu, Tegawende F. Bissyande, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You Cannot Fix What You Cannot Find! An Investigation of Fault Localization Bias in Benchmarking Automated Program Repair Systems. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, Xi'an, China, 102–113. https://doi.org/10.1109/ICST.2019.00020

[30] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawende F. Bissyande. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, Hangzhou, China, 1–12. https://doi.org/10.1109/SANER.2019.8667970

[31] 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, Beijing China, 31–42. https://doi.org/10.1145/3293882.3330577

[32] 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*. ACM, Seoul South Korea, 615–627. https://doi.org/10.1145/3377811.3380338

[33] Fan Long, Peter Amidon, and Martin Rinard. 2017. Automatic inference of code transforms for patch generation. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, Paderborn Germany, 727–739. https://doi.org/10.1145/3106237.3106253

[34] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, St. Petersburg FL USA, 298–312. https://doi.org/10.1145/2837614.2837617

[35] 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. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Virtual Event USA, 101–114. https://doi.org/10.1145/3395363.3397369

[36] Matias Martinez, Thomas Durieux, Romain Sommerard, Jifeng Xuan, and Martin Monperrus. 2017. Automatic repair of real bugs in java: a large-scale experiment on the defects4j dataset. *Empirical Software Engineering* 22, 4 (Aug. 2017), 1936–1964. https://doi.org/10.1007/s10664-016-9470-4

[37] Sergey Mechtaev, Xiang Gao, Shin Hwei Tan, and Abhik Roychoudhury. 2018. Test-Equivalence Analysis for Automatic Patch Generation. *ACM Transactions on Software Engineering and Methodology* 27, 4 (Nov. 2018), 1–37. https://doi.org/10.1145/3241980

[38] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. DirectFix: Looking for Simple Program Repairs. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. IEEE, Florence, Italy, 448–458. https://doi.org/10.1109/ICSE.2015.63

[39] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2016. Angelix: scalable multiline program patch synthesis via symbolic analysis. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, Austin Texas, 691–701. https://doi.org/10.1145/2884781.2884807

[40] Ben Mehne, Hiroaki Yoshida, Mukul R. Prasad, Koushik Sen, Divya Gopinath, and Sarfraz Khurshid. 2018. Accelerating Search-Based Program Repair. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, Vasteras, 227–238. https://doi.org/10.1109/ICST.2018.00031

[41] 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)*. IEEE, San Francisco, CA, USA, 772–781. https://doi.org/10.1109/ICSE.2013.6606623

[42] Yannic Noller, Ridwan Shariffdeen, Xiang Gao, and Abhik Roychoudhury. 2022. Trust enhancement issues in program repair. In *Proceedings of the 44th International Conference on Software Engineering*. ACM, Pittsburgh Pennsylvania, 2228–2240. https://doi.org/10.1145/3510003.3510040

[43] Yu Pei, Carlo A. Furia, Martin Nordio, Yi Wei, Bertrand Meyer, and Andreas Zeller. 2014. Automated Fixing of Programs with Contracts. *IEEE Transactions on Software Engineering* 40, 5 (May 2014), 427–449. https://doi.org/10.1109/TSE.2014.2312918

[44] Yuhua Qi, Xiaoguang Mao, and Yan Lei. 2013. Efficient Automated Program Repair through Fault-Recorded Testing Prioritization. In *2013 IEEE International Conference on Software Maintenance*. IEEE, Eindhoven, Netherlands, 180–189. https://doi.org/10.1109/ICSM.2013.29

[45] Gregg Rothermel and Mary Jean Harrold. 1996. Analyzing regression test selection techniques. *IEEE Transactions on software engineering* 22, 8 (1996), 529–551. Publisher: IEEE.

[46] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on Software Engineering* 27, 10 (Oct. 2001), 929–948. https://doi.org/10.1109/32.962562

[47] Seemanta Saha, Ripon k. Saha, and Mukul r. Prasad. 2019. Harnessing Evolution for Multi-Hunk Program Repair. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, Montreal, QC, Canada, 13–24. https://doi.org/10.1109/ICSE.2019.00020

[48] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. ACM, Seattle WA USA, 727–738. https://doi.org/10.1145/2950290.2950295

[49] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 981–992. https://doi.org/10.1145/3324884.3416532

[50] 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 Transactions on Software Engineering and Methodology* 28, 4 (Oct. 2019), 1–29. https://doi.org/10.1145/3340544

[51] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated patch correctness assessment: how far are we?. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Virtual Event Australia, 968–980. https://doi.org/10.1145/3324884.3416590

[52] 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*. IEEE, Vancouver, BC, Canada, 364–374. https://doi.org/10.1109/ICSE.2009.5070536

[53] 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*. ACM, Gothenburg Sweden, 1–11. https://doi.org/10.1145/3180155.3180233

[54] 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*. ACM, Athens Greece, 354–366. https://doi.org/10.1145/3468264.3468600

[55] 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*. ACM, Singapore Singapore, 959–971. https://doi.org/10.1145/3540250.3549101

[56] Chunqiu Steven Xia and Lingming Zhang. 2022. Less Training, More Repairing Please: Revisiting Automated Program Repair Via Zero-Shot Learning. http://arxiv.org/abs/2207.08281 arXiv:2207.08281 [cs].

[57] Jiahong Xiang, Xiaoyang Xu, Fanchu Kong, Mingyuan Wu, Haotian Zhang, and Yuqun Zhang. 2024. How Far Can We Go with Practical Function-Level Program Repair? arXiv:2404.12833 [cs.SE]

[58] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, Santa Barbara CA USA, 226–236. https://doi.org/10.1145/3092703.3092718

[59] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, Gothenburg Sweden, 789–799. https://doi.org/10.1145/3180155.3180182

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

[61] He Ye, Jian Gu, Matias Martinez, Thomas Durieux, and Martin Monperrus. 2022. Automated Classification of Overfitting Patches With Statically Extracted Code Features. *IEEE Transactions on Software Engineering* 48, 8 (Aug. 2022), 2920–2938. https://doi.org/10.1109/TSE.2021.3071750

[62] He Ye, Matias Martinez, Xiapu Luo, Tao Zhang, and Martin Monperrus. 2022. SelfAPR: Self-supervised Program Repair with Test Execution Diagnostics. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. ACM, Rochester MI USA, 1–13. https://doi.org/10.1145/3551349.3556926

[63] He Ye and Martin Monperrus. 2024. ITER: Iterative Neural Repair for Multi-Location Patches. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. ACM, Lisbon Portugal, 1–13. https://doi.org/10.1145/3597503.3623337

[64] Jooyong Yi. 2020. On the Time Performance of Automated Fixes. In *Proceedings of 6th International Conference in Software Engineering for Defence Applications*, Paolo Ciancarini, Manuel Mazzara, Angelo Messina, Alberto Sillitti, and Giancarlo Succi (Eds.). Vol. 925. Springer International Publishing, Cham, 312–324. https://doi.org/10.1007/978-3-030-14687-0_28 Series Title: Advances in Intelligent Systems and Computing.

[65] Jooyong Yi and Elkhan Ismayilzada. 2022. Speeding up constraint-based program repair using a search-based technique. *Information and Software Technology* 146 (June 2022), 106865. https://doi.org/10.1016/j.infsof.2022.106865

[66] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *ACM Transactions on Software Engineering and Methodology* 29, 1 (Jan. 2020), 1–53. https://doi.org/10.1145/3360004

[67] Michal Zalewski. Accessed on June, 2024. AFL. https://github.com/google/AFL.

[68] 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*. ACM, Athens Greece, 341–353. https://doi.org/10.1145/3468264.3468544