Towards Boosting Patch Execution On-the-Fly

Samuel Benton
University of Texas at Dallas
Samuel.Benton1@utdallas.edu

Yuntong Xie∗
Tsinghua University
xieyt18@mails.tsinghua.edu.cn

Lan Lu∗
Southern University of Science and Technology
11810935@mail.sustech.edu.cn

Mengshi Zhang†
Meta Platforms, Inc.
mengshizhang@fb.com

Xia Li
Kennesaw State University
xli37@kennesaw.edu

Lingming Zhang†
University of Illinois at Urbana-Champaign
lingming@illinois.edu

ABSTRACT
Program repair is an integral part of every software system’s lifecycle but can be extremely challenging. To date, various automated program repair (APR) techniques have been proposed to reduce manual debugging efforts. However, given a real-world buggy program, a typical APR technique can generate a large number of patches, each of which needs to be validated against the original test suite, incurring extremely high computation costs. Although existing APR techniques have already leveraged various static and/or dynamic information to find the desired patches faster, they are still rather costly. In this work, we propose SeAPR (Self-Boosted Automated Program Repair), the first general-purpose technique to leverage the earlier patch execution information during APR to directly boost existing APR techniques themselves on-the-fly. Our basic intuition is that patches similar to earlier high-quality/low-quality patches should be promoted/degraded to speed up the detection of the desired patches. The experimental study on 13 state-of-the-art APR tools demonstrates that, overall, SeAPR can substantially reduce the number of patch executions with negligible overhead. Our study also investigates the impact of various configurations on SeAPR. Lastly, our study demonstrates that SeAPR can even leverage the historical patch execution information from other APR tools for the same buggy program to further boost the current APR tool.

ACM Reference Format:

1 INTRODUCTION
Software systems persist everywhere in all facets of today’s society; they drive financial institutions, facilitate communication worldwide, oversee critical systems, and so forth. Software systems, however, are frequently distributed with numerous bugs that will eventually lead to severe disasters. For example, the 5th edition of Tricentis.com’s annual report shows that software failures impact half of the world’s population (3.7 billion users) and $1.7 trillion in assets; it also mentions that there can be far more bugs in the wild than we will likely ever know about [1]. Therefore, it is imperative for developers to fix these bugs as early as possible with minimal resource consumption. However, manual bug fixing can be extremely tedious, challenging, and time-consuming since modern software systems can be extremely complicated [2].

Fortunately, in lieu of manual bug fixing, researchers have also extensively studied Automated Program Repair (APR) [3–18], which aims to automatically fix software bugs to reduce manual debugging efforts. Typical APR techniques leverage off-the-shelf fault localization [19] techniques (such as Ochiai [20] and Tarantula [21]) to identify potential buggy locations. Then, they leverage various techniques to generate potential software patches for the potential buggy locations. Lastly, each generated patch will need to be executed against the original test suite to find the plausible patches (i.e., the patches that can pass all the original tests). Note that not all the plausible patches are the ones that developers want; thus, developers need to further inspect the produced plausible patches to derive the final correct ones. To date, various APR techniques have been proposed, including techniques based on predefined-templates [4, 22, 23], heuristics [8, 10, 24], and constraint solving [7, 25, 26]. Furthermore, APR techniques have also drawn wide attention from industry, e.g., Facebook [27], Fujitsu [28], and Alibaba [29].

Compared with manual bug fixing, APR can automatically fix a number of real-world bugs with minimal human intervention and can be easily integrated with the natural workflow of continuous integration lifecycle (e.g., Facebook’s in-house tool SapFix [27] has been integrated into its workflow). Despite the promising future of APR, it is not perfect yet and numerous issues still plague the area. Among the most paramount of these issues are still the time costs associated with numerous patches for large-scale real-world systems. Existing studies have demonstrated that patch validation dominates the costs of APR [30–33], since each patch needs to be executed against the original test suite.
To reduce the APR costs, researchers have proposed various techniques to reduce the number of patches generated, e.g., based on machine learning [28], code mining [13], and constraint solving [25]. However, prior work has demonstrated that such techniques can incur the dataset overfitting issue, i.e., the correct patches may be skipped for many other unstudied cases [22]. Furthermore, researchers have also proposed to prioritize all the generated patches to find the plausible patches earlier. Such existing techniques primarily utilize static or dynamic information to statically prioritize patches before the patch validation [8, 23, 26], e.g., almost all APR techniques use the suspiciousness values computed by off-the-shelf fault localization techniques to prioritize the patches; no further reprioritization is employed during the patch validation process of these tools, leading to limited improvement.

In this work, we propose SeAPR (Self-Boosted Automated Program Repair), the first general-purpose technique to leverage the patch-execution information during APR to directly boost existing APR techniques themselves on-the-fly. Our basic intuition is that earlier patch execution results can help better prioritize later patch executions on-the-fly to speed up the detection of the desired patches (e.g., plausible/correct patches). In this way, we promote the ranking of the patches similar to the executed high-quality patches, while degrading the ranking of the patches similar to the executed low-quality patches. More specifically, we analyze the modified elements to compute patch similarities as patches modifying similar program elements can exhibit close program behaviors. We have evaluated SeAPR on 13 state-of-the-art APR systems. Our study also investigates the impact of various configurations on SeAPR, e.g., the formula for patch prioritization, the type of patch-validation matrices (full or partial), the number of code elements considered for SeAPR, and the additional patch pattern information for computing patch similarity. Lastly, our study further evaluates the performance of SeAPR with historical patch-execution information from other APR tools on the same buggy program.

To summarize, this paper makes the following contributions:

- **Direction.** This paper opens a new dimension to leverage patch-execution information to boost existing APR techniques on-the-fly and can inspire more future work in this new direction.
- **Design.** We design the first technique, SeAPR, in this new direction to update each patch’s priority score based on its similarity with the executed patches and the quality of the executed patches.
- **Extensive Study.** We have performed an extensive study of the proposed technique on 13 state-of-the-art APR systems for JVM-based languages using the widely studied real-world bugs from Defects4J.
- **Practical Guidelines.** The study reveals various practical guidelines, including (1) the default SeAPR can substantially speed up the studied APR techniques by up to 79% with negligible overhead (regardless of various initial patch prioritization strategies used by the studied APR techniques), (2) SeAPR has stable performance when using different formulae for computing patch priority and different types of patch-execution matrices, (3) additional patch pattern information for patch similarity computation can further substantially improve SeAPR, and (4) SeAPR can even effectively utilize historical patch-execution information from other APR tools to boost current APR tools.

## 2 RELATED WORK

### 2.1 Automated Program Repair

Automated Program Repair (APR) techniques [3–18] aim to automatically fix software bugs to substantially reduce manual debugging efforts and have been extensively studied during the last decade. Typical APR techniques usually modify program code representations based on various patch-generation techniques and then validate each generated patch (e.g., via testing [24], formal specification checking [34], and static analysis [35]) to find the final desired patches. In recent years, APR techniques leveraging testing for patch validation have gained popularity as testing is the dominant methodology for detecting software bugs in practice. Such APR techniques usually include the following phases. (1) **Fault localization:** APR techniques first leverage off-the-shelf fault localization techniques [4, 5, 7, 10, 36, 37] to localize the potential buggy locations. (2) **Patch generation:** APR techniques will leverage various strategies to generate potential patches for the identified potential buggy locations. (3) **Patch validation:** all the generated patches will be executed against the original test suite to detect the patches that pass all the original tests, i.e., plausible patches. Of course, since not all plausible patches are desirable, patch correctness checking (often done via manual inspection in practice) is further involved to find the final correct patches, which are equivalent to developer patches.

According to a recent study [38], most state-of-the-art APR techniques can be divided into the following categories. (1) **Heuristic-based techniques** leverage various heuristics to iteratively explore the search space of all possible program edits. For example, the seminal GenProg technique [24] leverages genetic programming to synthesize donor code for high-quality patch generation, while the recent SimFix technique [8] employs advanced code search to obtain donor code for patch generation. (2) **Template-based techniques** leverage predefined fixing templates (e.g., changing “&gt;” to “\text{\textgreek{g}}”) to perform patch generation. Such predefined fixing templates can be either manually summarized (e.g., KPar [39]), or automatically inferred (e.g., HDRepair [40]) from historical bug fixes. (3) **Constraint-based techniques** transform the program repair problem into a constraint-solving problem and leverage state-of-the-art constraint solvers (e.g., SMT [41]) for patch generation, such as Nopol [25]. More recently, researchers have also looked into learning-based techniques [42, 43] to directly generate patches via learning from historical fixes.

Since it is extremely challenging for APR techniques to fix all possible bugs, researchers have also recently proposed the unified debugging work [29, 44, 45] to extend the application scope of APR to the bugs that cannot be automatically fixed. Its basic intuition is that the massive patch execution information during APR can actually substantially boost fault localization. For example, if a patch passes all the tests, it means the patch is likely to mute the impacts of the bug, even though this patch may not be correct; it can then be inferred that the patched location is highly related to the actual buggy location, since otherwise the bug effect would not be muted.
With unified debugging, even when APR techniques cannot fix a bug, unified debugging can still analyze the patch-execution information to provide useful hints about potentially buggy locations to help with manual repair. In this way, unified debugging extends the application scope of APR to all possible bugs, not only bugs automatically fixable. Inspired by unified debugging, we also aim to leverage the wealth of patch execution information generated during APR. Meanwhile, there are the following major differences. First, when unified debugging aims to leverage patch execution information for manual program repair, our SeAPR leverages such information to directly boost automated program repair, i.e., we aim to boost existing APR tools by prioritizing the desired plausible/correct patches earlier in the validation process. Second, their technical principles are substantially different. Unified debugging analyzes the correlation between patch locations and test outcomes to infer potentially buggy locations, while our work analyzes the correlation among executed and remaining patches via estimating their behavioral similarities to speed up the detection of desired patches. In fact, our work is inspired by prior work on mutation testing [46], which leverages the similarities of modified elements for different mutants to perform test prioritization/reduction for each mutant to speed up mutation testing.

### 3 STUDIED APPROACH

In this section, we first present the necessary preliminaries (Section 3.1). Then, we introduce the detailed SeAPR approach (Section 3.2). We will also discuss different SeAPR variants (Section 3.3). Lastly, we will introduce a further extension of SeAPR to leverage the patch execution information from other APR tools for even faster APR (Section 3.4).

#### 3.1 Preliminaries

**Definition 3.1. Patch validation matrix:** Matrix $M_b$ defines the validation results of all tests against all patch candidates. In the matrix, each cell describes the validation result of test $t \in T$ against patch $p \in P$. Possible values for each cell are as follows: (1) - if $t$ remains unvalidated, (2) $X$ if $t$ fails on $p$ and (3) ✓ if $t$ passes on $p$.

<table>
<thead>
<tr>
<th>Patch ID</th>
<th>$t_1$</th>
<th>$t_2$</th>
<th>$t_3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$ (buggy ver.)</td>
<td>$X$</td>
<td>✓</td>
<td>$X$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$p_3$</td>
<td>$X$</td>
<td>$X$</td>
<td>✓</td>
</tr>
<tr>
<td>$p_4$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 1: Example of full/partial patch-validation matrix**

Ideally the patch-validation matrix should be full, i.e., every cell should be ✓ or $X$. In practice during the APR process however, most modern APR tools terminate the test execution for one patch immediately after observing any failing test on that particular patch, since the primary goal is to find correct patches and patches which fail any test cannot even be plausible. In this way, the APR process can be largely sped up without sacrificing repair effectiveness. Not all APR tools employ this strategy, so we study both types of matrices, where some tests remain unexecuted (partial matrices) versus where all tests always execute (full matrices). Table 1 presents the example full/partial matrices for 4 example patches (i.e., $p_1$, $p_2$, $p_3$, and $p_4$) on 3 example tests (i.e., $t_1$, $t_2$, and $t_3$). Note that the first row for the patch-validation matrix is always the test execution results of the original buggy program (i.e., $p_0$).

**Definition 3.2. Patch modification matrix:** Matrix $M_m$ presents all program elements modified within each patch. Each cell describes if patch $p \in P$ modifies element $e \in E$ (i.e., all possible program elements). Acceptable values for each cell are as follows: (1) ✓ if $p$ modifies element $e$ and (2) - if $p$ does not modify element $e$.

<table>
<thead>
<tr>
<th>Patch ID</th>
<th>$e_1$</th>
<th>$e_2$</th>
<th>$e_3$</th>
<th>$e_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>$p_2$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$p_3$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>$p_4$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

**Table 2: Example of patch modification matrix**

Table 2 presents an example patch modification matrix for the above four example patches on four program elements. In this
way, since $p_1$ patches program elements $\{e_1, e_2, e_3\}$, the first three columns are $\checkmark$ for $p_1$. Note that the patch modification matrix can be defined at different levels (e.g., at the package, class, method, and statement granularities) depending on the granularity of considered program elements. In this paper, we mainly consider the method granularity, e.g., the columns will be the methods modified by each program patch. Compared with the patch validation matrix, the patch modification matrix can be computed much faster (in fact with negligible overhead), and thus can be leveraged to speed up the patch validation process.

3.2 Basic SeAPR

Given the above introduced patch-validation matrix and patch-modification matrix (which are readily available for almost all APR tools) for the already executed/validated patches, our SeAPR performs on-the-fly patch prioritization to speed up APR. Our basic intuition is that patches similar to executed high-quality patches are likely to also be high-quality and should therefore be prioritized earlier; likewise, patches rather similar to executed low-quality patches should be deprioritized. In this section, we first introduce our definitions for patch quality (Section 3.2.1); then, we introduce the detailed strategy to compute patch similarity with high- or low-quality patches (Section 3.2.2); next, we introduce our final priority score computation for all unexecuted patches (Section 3.2.3); lastly, we present our overall algorithm (Section 3.2.4) with corresponding examples (Section 3.2.5).

3.2.1 Patch Quality. When processing patches that have been executed/validated, we need to estimate the patch’s quality by analyzing the patch validation matrix. Intuitively, the ultimate goal of APR is to produce plausible/correct patches that can pass all the original tests. Therefore, in this study, a patch is classified as high-quality (patches we wish to prioritize) if it can make any originally failing test pass; likewise, a patch is classified as low-quality (patches we wish to deprioritize) if it cannot make any originally failing test pass. Formally, the set of high-quality and low-quality patches can be defined as Equations (1) and (2), respectively.

$$
P_h = \{p|\exists t. M_v[p,t] = \checkmark \land M_v[p,t,l] = \times\} \quad (1)$$

$$
P_l = \{p|\forall t. M_v[p,t,l] = \times \land M_v[p,t] = \checkmark\} \quad (2)$$

Note that we can also easily compute the detailed number of originally failing tests that now pass on a patch; however, prior work has demonstrated that the detailed test number can be misleading [29]. Of course, this is just the first work in this new direction, and we highly encourage other researchers to investigate other better ways to estimate patch quality.

3.2.2 Patch Similarity. After calculating patch quality for executed patches, we iterate through all remaining patches within $P$ to compute their similarity information with the executed high/low-quality patches. For each patch $p$ that has not been validated yet, we compare its patch modification matrix information against that of each of the validated patches. During the comparison, we compute the number of elements matching and differing among the two compared patches (i.e., two rows in the patch modification matrix). We calculate the number of matching elements by performing the set intersection on the two patch modification matrix rows representing the two patches. Likewise, we calculate the number of differing elements by performing a symmetric set difference (i.e., $A \ominus B = (A - B) \cup (B - A)$) on the two patch modification matrix rows representing the two patches.

Based on the similarity/dissimilarity with high/low-quality patches, we can compute the following tuple for each unvalidated patch $p$ for prioritization, $(s_h, d_h, s_l, d_l)$. Our basic idea is that $s_h$ should get increased when $p$ shares elements with high-quality patches, $s_l$ should get increased when $p$ shares elements with low-quality patches, $d_h$ should get increased when $p$ has set difference with high-quality patches, and $d_l$ should get increased when $p$ has set difference with low-quality patches. Since the detailed number of the matching/different modified elements between two patches can tell the detailed similarity/dissimilarity information, the increment should also consider such detailed information. In this way, the formulae for computing the tuple for each unvalidated patch $p$ are:

$$
s_h[p] = \Sigma_{p'}|\{e|e \in M_m[p] \land M_m[p'] \land p' \in P_h}\| \quad (3)$$

$$
s_l[p] = \Sigma_{p'}|\{e|e \in M_m[p] \land M_m[p'] \land p' \in P_l}\| \quad (4)$$

$$
d_h[p] = \Sigma_{p'}|\{e|e \in M_m[p] \land \neg M_m[p'] \land p' \in P_h}\| \quad (5)$$

$$
d_l[p] = \Sigma_{p'}|\{e|e \in M_m[p] \land \neg M_m[p'] \land p' \in P_l}\| \quad (6)$$

Note that $M_m[p]$ denotes the set of program elements modified by patch $p$. For example, if a validated patch $p'$ is high-quality and shares elements with the current $p$, the $s_h$ of $p$ is then increased for $[M_m[p] \land M_m[p']]$. All the other tuple elements can be defined in a similar way.

3.2.3 Patch Prioritization. Based on the similarity tuple we computed from the previous step, we can compute the priority score for each unvalidated patch based on the following intuition: (1) a patch more similar/dissimilar with high-quality patches should be promoted/degraded, (2) a patch similar/dissimilar with low-quality patches should be degraded/promoted. Actually, such intuition is quite similar to traditional spectrum-based fault localization (SBFL) [48], where the intuition is (1) a program element executed/unexecuted by more failed tests should be more/less suspicious, (2) a program element executed/unexecuted by more passed tests should be less/more suspicious. In this way, all the traditional fault localization formulae can be directly leveraged here to compute the patch priority. We use the Ochiai formula, shown in Equation (7), as our default formula as it is often the default formula for SBFL [8, 22, 23]. In this way, patches will be promoted/demoted if they are similar/dissimilar with other high-quality patches, consistent with our intuition.

$$
Ochiai = \frac{s_h}{\sqrt{(s_h + d_h) \times (s_l + d_l)}} \quad (7)
$$

3.2.4 Overall Algorithm. Given the above definitions, we can now present the overall SeAPR algorithm. Shown in Algorithm 1, SeAPR first initializes the similarity tuples of all patches considered for SeAPR as $1$s (Line 2). Then, SeAPR iterates through all such patches and validates them in order (Lines 3-16). During each iteration, SeAPR first gets the patch $p$ with the highest priority and pops that from the patch list $P$. Note that for the patches with tied SeAPR priority scores (e.g., all patches are tied before the first patch execution), SeAPR prioritizes them with their original ordering from the

1Note that they are initialized as 1s (not 0s) for numerical stability.
Algorithm 1: SeAPR Algorithm

Input: The original buggy program $p_b$, test suite $T$, the list of candidate patches $P$ considered for SeAPR, the similarity tuples ($q_i, d_i, s_i$)
Output: Plausible patch: $P_f$

1. begin
2. Initialize ($q_i, d_i, s_i$) for all patches
3. while $P$ is not empty do
4.   $p = \text{pop}(P)$; // pop the remaining patch with the highest priority
5.   $M_b \leftarrow \text{execute}(T, p)$; // validate $p$
6.   if $p$ is PLAUSIBLE then
7.     $P_f \leftarrow p$; // put $p$ into the resulting set for manual inspection
8.     $r \leftarrow \text{computePatchQuality}(p, M_b, s_i)$
9.     // incrementally update the similarity tuples for the remaining patches for $p \in P$ do
10.    if $r = \text{HIGH} = \text{QUALITY}$ then
11.       $q_i[p] \leftarrow |M_b[p] \cap M_a[p]|$
12.       $d_i[p] \leftarrow |M_b[p] \cap M_a[p]|$
13.    else if $r = \text{LOW} = \text{QUALITY}$ then
14.       $q_i[p] \leftarrow |M_b[p] \cap M_a[p]|$
15.       $d_i[p] \leftarrow |M_b[p] \cap M_a[p]|$
16.     end
17.     $P = \text{computePriorityScore}(P, q_i, d_i, s_i)$
18. end
19. end

2.5 Example. Let us now use the partial patch validation matrix\(^2\) (shown in Table 1) and its corresponding patch modification matrix (shown in Table 2) as the example to illustrate our SeAPR technique. For this example, if we follow the original patch execution ordering (top-down), we need to execute four patches before finding the final plausible patch. Now we discuss how our SeAPR can help speed up this process.

\(^2\)Note that we use partial since most APR tools will collect partial matrices, but our idea generalizes to full matrices (as studied in Section 5.3).

Shown in Tables 3 and 4, Column "Quality" describes the patches’ actual quality (available after the corresponding patch validation); Column "Match" describes the set of matching elements against the last executed patch for each patch; Column "Differ" describes the set of differing elements against the last executed patch for each patch; Columns \(s_i, d_i, q_i, \) and \(d_i\) represent the accumulated similarity tuples per patch (initialized as 1); lastly, Column "Score" represents the Ochiai priority score as defined in Equation (7).

In the first iteration (shown in Table 3), SeAPR will compute the quality of the executed patch, $p_1$ (marked with gray). We can immediately determine that the patch is low quality simply because it cannot make any originally failing tests pass. Note that we also show the quality for all other unexecuted patches to illustrate the quality computation. Then, given $p_1$ has been executed, we can update the similarity tuple for each remaining patch. For example, for $p_2$, the set intersection and symmetrical set difference with $p_1$ is $\{e_1, e_2, e_3\} \cap \{e_1, e_2, e_3, e_4\} = \{e_1, e_2, e_3\}$ and $\{e_1, e_2, e_3\} \ominus \{e_1, e_2, e_3, e_4\} = \{e_1, e_4\}$, respectively. Therefore, since $p_1$ is a low-quality patch, $q_1$ increments by 3 and $d_1$ increments by 1 for $p_2$, resulting in the tuple $(q_1=1+0, d_1=1+0, 1+3, d_1=1+1)$. Similarly, we can compute the similarity tuples for all the other remaining patches. Then, via applying the default Ochiai formula on the computed tuples, we can compute the priority scores for all the three remaining patches as shown in Column "Score" in Table 3. In this way, the patch with the highest priority, $p_4$, is selected for the next patch execution.

In the second iteration, $p_4$ gets executed (marked in gray) as shown in Table 4. Note that $p_4$ is a plausible patch that can pass all tests. Therefore, the developers can immediately start manual inspection to check if $p_4$ is the correct patch. Of course, the patch execution can still continue if $p_4$ is not the correct patch. Continuing the algorithm, the remaining patches will be further compared with the newly executed patch to update their similarity tuples. For example, for $p_3$, the set intersection and symmetrical set difference with $p_4$ is $\{e_1, e_2, e_3\} \cap \{e_1\} = \{e_1\}$ and $\{e_1, e_2, e_3\} \ominus \{e_1\} = \{e_2, e_3, e_4\}$, respectively. Since $p_4$ is a high-quality patch, $p_2$’s tuple is updated by incrementing $s_1$ by 1 and $d_1$ by 3, resulting in the tuple $(q_2=1+1, d_2=1+1, 3+0, d_2=2+0)$. In this way, we can compute the scores for all remaining patches.

Table 3: SeAPR step-by-step when processing $p_1$

<table>
<thead>
<tr>
<th>ID</th>
<th>Quality</th>
<th>Match</th>
<th>Differ</th>
<th>$s_1$</th>
<th>$d_1$</th>
<th>$d_1$</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>Low</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$p_2$</td>
<td>High</td>
<td>${e_1, e_2, e_3}$</td>
<td>${e_1}$</td>
<td>1+0</td>
<td>1+0</td>
<td>1+3</td>
<td>1+1</td>
</tr>
<tr>
<td>$p_3$</td>
<td>Low</td>
<td>${e_2, e_3}$</td>
<td>${e_1}$</td>
<td>1+0</td>
<td>1+0</td>
<td>1+2</td>
<td>1+1</td>
</tr>
<tr>
<td>$p_4$</td>
<td>Plausible</td>
<td>${e_1}$</td>
<td>${e_2, e_3}$</td>
<td>1+0</td>
<td>1+0</td>
<td>1+1</td>
<td>1+2</td>
</tr>
</tbody>
</table>

Table 4: SeAPR step-by-step when processing $p_4$

<table>
<thead>
<tr>
<th>ID</th>
<th>Quality</th>
<th>Match</th>
<th>Differ</th>
<th>$s_1$</th>
<th>$d_1$</th>
<th>$d_1$</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>Low</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>$p_2$</td>
<td>High</td>
<td>${e_1, e_2, e_3}$</td>
<td>${e_1}$</td>
<td>1+1</td>
<td>1+3</td>
<td>3+0</td>
<td>2+0</td>
</tr>
<tr>
<td>$p_3$</td>
<td>Low</td>
<td>${e_1}$</td>
<td>${e_2, e_3}$</td>
<td>1+0</td>
<td>1+3</td>
<td>3+0</td>
<td>2+0</td>
</tr>
</tbody>
</table>

For this example, we observe that the original patch execution ordering requires 4 patch executions to find the first plausible patch, while our SeAPR reduces the number of required patch executions to only 2, i.e., $\frac{4-2}{4} = 50\%$ patch reduction. In this way, the developers can start manual patch inspection as soon as after 2 patch executions.
3.3 SeAPR Variants

3.3.1 Patch-Prioritization Formulae. Besides Ochiai, other SBFL formulae can also be applied here. In particular, we study all SBFL formulae from prior work [49] in Section 5.2.

3.3.2 Validation-Matrix Types. Besides the partial patch-validation matrices widely used in practice, we also consider the impact of full validation matrices on SeAPR in Section 5.3.

3.3.3 Additional Patch Pattern Information. By default, SeAPR only uses the set of modified program elements to calculate patch similarity for prioritizing patches on-the-fly. Another SeAPR extension is to compute the similarity score with additional information. Therefore, in Section 5.4, we further study another SeAPR variant, which additionally considers that patches sharing the same fixing patterns may also share similar program behaviors. In this way, we can promote patches applying the same fixing patterns as known high-quality patches to further boost SeAPR.

**Definition 3.3.** Patch repair pattern matrix: Matrix $M_p$ presents the applied repair patterns applied to each patch. Each cell describes if patch $p \in P$ applies repair pattern $r \in R$ (i.e., all predefined repair patterns). Acceptable values for each cell are as follows: (1) ✓ if $p$ applies pattern $r$ and (2) - otherwise.

<table>
<thead>
<tr>
<th>Patch ID</th>
<th>$r_1$</th>
<th>$r_2$</th>
<th>$r_3$</th>
<th>SeAPR Features</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1$</td>
<td>✓</td>
<td>-</td>
<td>-</td>
<td>$[e_1, e_2, e_3]$</td>
</tr>
<tr>
<td>$p_2$</td>
<td>-</td>
<td>✓</td>
<td>-</td>
<td>$[e_1, e_2, e_3]$</td>
</tr>
<tr>
<td>$p_3$</td>
<td>-</td>
<td>-</td>
<td>✓</td>
<td>$[e_2, e_3]$</td>
</tr>
<tr>
<td>$p_4$</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>$[e_1, e_2, e_3]$</td>
</tr>
</tbody>
</table>

Table 5: Example of patch repair pattern matrix

This variant only slightly differs from the default SeAPR when computing patch similarity, e.g., this variant considers both (1) the set of modified elements and (2) the applied repair patterns. Based on the above patch-pattern matrix definition, we can recompute the similarity tuples for further improving SeAPR, e.g., $s_h[p]$ in Equation (3) becomes:

$$s_h[p] = \frac{\sum_{r \in R} |\{e | e \in M_0[p] \land M_0[p'] \land p' \in P_h\}|}{\sum_{r \in R} |\{e | e \in M_0[p] \land M_0[p'] \land p' \in P_h\}|}$$ (8)

3.4 Further Leveraging APR Results from Other Tools

In practice, one repair tool is often insufficient to successfully find a correct patch. Thus developers often need to run multiple repair tools to automatically fix a bug. Currently, different repair tools can help achieve even more effective SeAPR for the current APR tool.

4 EXPERIMENTAL DESIGN

4.1 Research Questions

In this paper, we study the following research questions:

- **RQ1:** How does the default SeAPR perform on state-of-the-art APR systems?
- **RQ2:** How do different prioritization formulae impact SeAPR?
- **RQ3:** How do full patch validation matrices impact SeAPR?
- **RQ4:** How does SeAPR perform when leveraging additional patch pattern information?
- **RQ5:** How does SeAPR perform when further leveraging historical repair information from other APR systems?

Note that we first study our default SeAPR configuration with the Ochiai formula, partial patch validation matrices, similarity computation based on modified elements in RQ1. Then, in RQ2 and RQ3, we investigate the impact of different formulae and types of patch validation matrices to study the robustness of SeAPR. In RQ4, we leverage the additional patch pattern information (available for state-of-the-art template-based APR techniques [4, 22]) to further boost the effectiveness of SeAPR. Lastly, in RQ5, we investigate whether historical patch validation information from other APR tools can help achieve even more effective SeAPR for the current APR tool.

4.2 Evaluation Dataset

We choose to evaluate SeAPR against Defects4J (V1.2.0), the most widely used APR dataset to date, which will allow SeAPR to be easily compared with and replicated in the future. The details for the dataset are shown in Table 6. Column "#Bugs" presents the number of buggy versions studied for each subject. Columns "#Tests" and "LOC" present the number of JUnit tests and lines of code available within the head (i.e., most recent) version of each subject, respectively. For each studied buggy version, the average number of failing tests is 2.37 (ranging from 1 to 66). Please also note that $P_h$ (the set of high-quality patches, which is closely related to SeAPR effectiveness) is not necessarily related to the number of failed tests since $P_h$ patches can pass the failed test(s) but fail on the other originally passing tests (i.e., Equation 1 does not check whether the originally passing tests still pass). For example, for the recent PraPR tool, there are 178 high-quality patches for each bug version on average.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Name</th>
<th># Bugs</th>
<th># Tests</th>
<th>LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chart</td>
<td>JFreeChart</td>
<td>26</td>
<td>2,205</td>
<td>96K</td>
</tr>
<tr>
<td>Lang</td>
<td>Apache Lang</td>
<td>65</td>
<td>2,245</td>
<td>22K</td>
</tr>
<tr>
<td>Math</td>
<td>Apache Math</td>
<td>106</td>
<td>3,602</td>
<td>85K</td>
</tr>
<tr>
<td>Time</td>
<td>Joda-Time</td>
<td>27</td>
<td>4,130</td>
<td>28K</td>
</tr>
<tr>
<td>Mockito</td>
<td>Mockito framework</td>
<td>38</td>
<td>1,366</td>
<td>23K</td>
</tr>
<tr>
<td>Closure</td>
<td>Google Closure compiler</td>
<td>133</td>
<td>7,927</td>
<td>90K</td>
</tr>
</tbody>
</table>

**Table 6:** Studied bugs from Defects4J v1.2.0
4.3 Studied APR Tools
Following prior work [38, 44], we consider 17 program repair tools publicly available and applicable to Defects4J in this study, show in Table 7. Of these tool candidates, we exclude ACS, DynaMoth, and Nopol due to low numbers of patches generated (i.e., <500 total patches across all studied Defects4J projects). Evaluating our technique on such tools with small numbers of patches will induce noise into our results; also, in practice, it is not necessary to perform on-the-fly patch prioritization on such tools with a small number of patches. We further exclude Simfix since the tool stops execution after finding the first plausible patch. The validation results of such tools cannot be degraded, since the last patch is always the desired plausible patch; results from such tools can only be improved, biasing our findings. In total, we studied 13 repair tools in this paper. Note that Arja, GenProg-A, and JGenProg generate new patches based on patch executions from earlier iterations due to their evolutionary design. Therefore, we should also have excluded these 3 tools; however, we decided to include these tools simply to demonstrate the potential benefits of SeAPR (excluding them also does not affect our findings as their results are consistent with other tools). Note that among all the three categories of APR tools, the template-based tools have been widely recognized as the state-of-the-art [22, 38], and can generate a large number of patches. Therefore, the template-based APR tools are the main focus of our technique and study.

<table>
<thead>
<tr>
<th>Tool Category</th>
<th>Tool(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constraint-based</td>
<td>Arja [10], Cardumen [9], DynaMoth [6], Nopol [25]</td>
</tr>
<tr>
<td>Heuristic-based</td>
<td>Arja [10], GenProg-A [10], JGenProg [9], JKali [9], JMutRepair [9], Kali-A [10], RSRepair-A [10], Simfix [8]</td>
</tr>
<tr>
<td>Template-based</td>
<td>Avatar [5], FriiMiter [36], KPars [37], PralPR [22], Tbar [4]</td>
</tr>
</tbody>
</table>

Table 7: Repair tools under consideration

4.4 Evaluation Metrics
We have adopted the following two evaluation metrics: the reduction on the number of patch executions before finding (1) the first plausible patch and (2) the first correct patch. We study (1) since in practical developers will begin manual inspection after the first plausible patch is found; in this way, faster plausible patch detection can enable developers to start manual inspection earlier (and potentially speed up the APR process). Similarly, we study (2) since developers will stop the patch validation process once the correct patch is found; in this way, faster correct patch detection can save overall APR time. Also note that we mainly leverage the reduction on the number of patch executions as recommended by prior work [38] since time costs are dependent on many factors (e.g., specific implementations and test execution engines) unrelated to APR approaches and are often unstable. Furthermore, we also discuss the results of time costs in Section 5.6.

To this end, our primary evaluation metric (patch reduction) can be computed as \( \frac{P_{\text{baseline}} - P_{\text{new}}}{P_{\text{baseline}}} \), where \( P_{\text{baseline}} \) represents the position of the first desired (i.e., plausible/correct) patch, pre-prioritization. \( P_{\text{new}} \) represents the position of the first desired plausible/correct patch, post-prioritization. Note that when multiple desired patches are produced, the initial desired patch and the final desired patch are not necessarily the same patch.

4.5 Experimental Procedure
For each studied APR tool, we evaluate the effectiveness of SeAPR on all patches that can be generated and validated by the tool within its original time limit (used in its original paper). We first analyze the original execution of each tool on a subject-by-subject basis to obtain (1) the original patch execution order per repair tool and (2) the position of the earliest plausible/correct patch. After information collection, we then repeat the patch validation process for each tool again with our SeAPR on-the-fly patch prioritization. For each given subject, SeAPR initially executes the first patch produced by the tool. After the first patch execution, SeAPR iterates through all patches not yet validated, following Algorithm 1, to record the new position for the first plausible/correct patch. Note that when computing patch similarity based on patch modification information, SeAPR will only be applied to the patches belonging to the Top-30 methods since most APR tools only patch such top methods (the impact of applying SeAPR to different numbers of top methods is also studied in Section 5.6.1); the remaining patches are simply executed with their original ranking.

All our experiments were conducted within the following environment: 36 3.0GHz Intel Xeon Platinum Processors, 60GB of memory, and Ubuntu 18.04.4 LTS operating system.

5 RESULT ANALYSIS
5.1 RQ1: Overall SeAPR Effectiveness
5.1.1 Quantitative Analysis. In this section, we first investigate the overall effectiveness of our default SeAPR on all 13 studied repair tools against Defects4J. Table 8 shows the patch reduction in terms of the first plausible patches. In this table, Column 1 presents the APR systems studied in this work; Columns 2 and 3 present the average rank of the first plausible patches before and after applying SeAPR to each studied APR system, while Columns 4 and 5 present the absolute improvement and the reduction ratio achieved by SeAPR. Note that not all the studied APR tools can generate plausible patches for all the studied bugs. Therefore, in this table, for each APR tool, we only present the results for the buggy versions on which it can produce plausible patches. From this table, we have the following observations. First, SeAPR improves the overall effectiveness of patch validations for almost all repair tools. For example, SeAPR reduces patch validation by 78.91% on GenProg-A and 54.87% on Avatar. Meanwhile, SeAPR slightly degrades the performance of Cardumen. One reason is that Cardumen only generates a small number of patches for the few buggy versions with plausible patches, leaving limited room for further improvement. Second, the SeAPR reduction rates tend to be higher for APR systems with more patch executions (i.e., the APR systems that tend to be more expensive and need more optimizations). For example, the reduction rate is above 20% for all APR systems with over 100 patch executions before the first plausible patch before applying SeAPR, and is above 40% for all APR systems with over 200 patch executions before the first plausible patch before applying SeAPR. This further demonstrates the effectiveness of SeAPR in boosting potentially expensive APR systems.

Similarly, Table 9 further shows the patch reduction ratios for finding the first correct patches. In this table, for each APR tool, we only present the results for the buggy versions on which it can
produce correct patches. Note that there exists no precise information on the number of correct patches reported in the original or subsequent papers for GenProg-A, RSRepair-A, and Kali-A [10], as well as PraPR (due to the large number of plausible patches generated at the bytecode level) [22]. For JKali and JGenProg, we cannot generate any correct patches as reported by the tool’s original paper [9], likely due to the tools’ nondeterminism and different execution environments. Therefore, we exclude these six repair tools from Table 9. We can find that SeAPR can achieve even higher patch reduction rates in terms of the first correct patch (compared with reductions for the first plausible patches). For example, SeAPR improves the overall performance of Arja, Avatar, JMutRepair, KPar, and TBar by 72.96%, 80.15%, 53.57%, 64.06%, and 48.33%, respectively. Another interesting finding is that SeAPR does not degrade the performance of any of the studied tools, although some tools remain unimproved. Note that the patch reductions for correct patches do not apply to all studied repair tools. Also, the results are mainly consistent with (and even better than) that for plausible patches, since SeAPR aims to improve the ranking of plausible patches while whether a plausible patch is correct is orthogonal to our technique. Therefore, we use patch reductions for plausible patches for all following RQs due to the space limit.

Table 8: Default SeAPR results for plausible patches

<table>
<thead>
<tr>
<th>APR Systems</th>
<th>Orig. Rank</th>
<th>SeAPR Rank</th>
<th>Diff</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arja</td>
<td>203.22</td>
<td>121.83</td>
<td>81.39</td>
<td>80.05%</td>
</tr>
<tr>
<td>Avatar</td>
<td>57.52</td>
<td>25.96</td>
<td>31.56</td>
<td>54.87%</td>
</tr>
<tr>
<td>Cardumen</td>
<td>10.25</td>
<td>11.00</td>
<td>0.75</td>
<td>7.32%</td>
</tr>
<tr>
<td>JMutRepair</td>
<td>85.26</td>
<td>48.13</td>
<td>37.13</td>
<td>43.55%</td>
</tr>
<tr>
<td>GenProg-A</td>
<td>226.21</td>
<td>47.71</td>
<td>178.50</td>
<td>78.91%</td>
</tr>
<tr>
<td>JGenProg</td>
<td>10.67</td>
<td>9.83</td>
<td>0.84</td>
<td>8.18%</td>
</tr>
<tr>
<td>JKali</td>
<td>6.50</td>
<td>4.75</td>
<td>1.75</td>
<td>26.92%</td>
</tr>
<tr>
<td>TBar</td>
<td>25.33</td>
<td>23.72</td>
<td>1.61</td>
<td>6.36%</td>
</tr>
<tr>
<td>KPar</td>
<td>76.59</td>
<td>6.14</td>
<td>13.45</td>
<td>17.56%</td>
</tr>
<tr>
<td>RSRepair-A</td>
<td>105.55</td>
<td>82.50</td>
<td>23.05</td>
<td>21.84%</td>
</tr>
<tr>
<td>PraPR</td>
<td>2052.8</td>
<td>774.47</td>
<td>1277.29</td>
<td>62.25%</td>
</tr>
</tbody>
</table>

Table 8: Default SeAPR results for plausible patches

<table>
<thead>
<tr>
<th>APR Systems</th>
<th>Orig. Rank</th>
<th>SeAPR Rank</th>
<th>Diff</th>
<th>Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arja</td>
<td>355.00</td>
<td>96.00</td>
<td>259.00</td>
<td>72.96%</td>
</tr>
<tr>
<td>Avatar</td>
<td>44.50</td>
<td>8.83</td>
<td>35.67</td>
<td>80.15%</td>
</tr>
<tr>
<td>Cardumen</td>
<td>17.00</td>
<td>17.00</td>
<td>0.00</td>
<td>0.00%</td>
</tr>
<tr>
<td>FixMiner</td>
<td>3.50</td>
<td>3.50</td>
<td>0.00</td>
<td>0.00%</td>
</tr>
<tr>
<td>JMutRepair</td>
<td>28.00</td>
<td>13.00</td>
<td>15.00</td>
<td>53.57%</td>
</tr>
<tr>
<td>KPar</td>
<td>58.45</td>
<td>21.00</td>
<td>37.45</td>
<td>64.06%</td>
</tr>
<tr>
<td>TBar</td>
<td>33.00</td>
<td>17.05</td>
<td>15.95</td>
<td>48.33%</td>
</tr>
</tbody>
</table>

Table 9: Default SeAPR results for correct patches

5.1.2 Qualitative Analysis. Next, we present detailed examples to investigate the performance of SeAPR. We first look into cases where SeAPR can help boost APR:

**Example 1:** Figure 1a shows one of the low-quality patches produced by Arja on Chart-19 while Figure 1b shows one of several generated plausible patches. Note that these two patches modify different files. In this example, we observe how low-quality patches may help prioritize plausible patches. According to our technique, other patches sharing similar modified elements with these low-quality patches are deprioritized. Thus, as any low-quality patch modifying one particular set of methods is validated, all other similar patches are deprioritized. This essentially results in (1) the clustering of patches based on the set of modified methods and (2) prioritizing/deprioritizing these clusters. Upon validation of consecutive low-quality patches, this phenomenon culminates to a breadth-first exploration of the patch clusters, mitigating the risk of some high-quality patches getting starved. This process repeats until Arja finds plausible patch from Figure 1b. With SeAPR, we observe Arja validates the plausible patch 33rd instead of 626th, a patch reduction of 94.7%.

**Example 2:** Since PraPR tends to generate substantially more patches than other APR tools, we further present how SeAPR can help prioritize the plausible patches for PraPR. The non-plausible high-quality patch generated by PraPR as shown in Figure 2a modifies method NodeMismatch.checkTreeEqualsImpl(Node node2) { …
  res = n.checkTreeEqualsImpl(n2); …
} and is able to pass some originally failed tests since the method is quite influential for the failed tests. In this way, it can help substantially improve the ranking of a plausible patch modifying the same method as shown in Figure 2b. After applying SeAPR, the first plausible patch can be ranked at 46th now (at 3165th originally), a patch reduction of 98.5%.

Since SeAPR is a heuristic-based technique, it does not work for all cases. Therefore, we also look into the negative cases:
which modify the same method as the plausible patch but cannot
result, validation of these patches will degrade the execution of the
strBuiler(String)

Example 4:
The only plausible patch which is also the only high-
ranked below the plausible patch to be executed earlier. As a result,

Figure 3b presents the only plausible patch produced

(a) Non-matching High-quality patch
(b) Plausible patch
Figure 3: PraPR Lang-61 patches

(a) Low-quality patch
(b) Plausible patch
Figure 4: TBar Math-63 patches

Example 3: Figure 3b presents the only plausible patch produced by

Example 4: The only plausible patch which is also the only high-

Finding 1: SeAPR can substantially reduce patch executions
before finding the first plausible/correct patches for almost all
studied repair tools, with a maximum improvement of 78.91%
(plausible) / 80.15% (correct).

5.2 RQ2: Impact of Different Formulae
In RQ1, we studied SeAPR with the Ochiai formula. In fact, SeAPR
is a general approach and can leverage any other existing SBFL
formula. Therefore in this RQ, we further investigate the impact
of different SBFL formulae on the effectiveness of SeAPR. Figure 5
shows the experimental results of 8 widely studied SBFL formul-
ae [49] on the 13 repair tools in terms of patch reduction. From
the results, we observe that SeAPR reduces the patch validations
of studied SBFL formulae across almost all repair tools, by up to
78.94% (GenProg-A with Kuleczynski and Jaccard). Furthermore, for
different SBFL formulae, the overall patch reduction on all the stud-
ied 13 tools is rather stable. For example, the formula with the best
overall improvement is SBI (30.27%) and the formula with the worst
performance is Op2 (25.74%), i.e., the difference of all studied for-
mlae is only 4.5 percentage points (pp). Op2 performs worse than
the other formulae because it mainly considers the $s_g$ information
while largely ignoring other valuable information from the simi-
larity tuples, demonstrating the importance of all the information
traced by SeAPR.

Finding 2: All studied formulae achieve stable performance,
speeding up the overall validation by at least 25.74%, demon-
strating the effectiveness of our design.

5.3 RQ3: Impact of Full Validation Matrix
We now have studied SeAPR with the default partial patch-validation
matrices. In this section, we further investigate the performance of
SeAPR with full patch-validation matrices. Figure 6 presents the
patch reductions achieved by SeAPR with the partial and full patch-
validation matrices. From this figure, we see that SeAPR can achieve
significant reductions on both full and partial patch-validation
matrices, indicating the general applicability of SeAPR. For example,
the overall reduction by SeAPR with the partial/full matrices is
28.53%/20.06% on all studied tools. Interestingly, despite the over-
all positive reductions, SeAPR degrades the performance of some
repair tools when using full matrices, e.g., the patch reduction on
Arja changes from 40.05% (with partial matrices) into -16.27% (with
full matrices). One possible reason for why SeAPR performs a bit
worse with full matrices is that a significant portion of low-quality
patches with partial matrices are considered high-quality patches with full matrices, since full matrices execute all tests against each patch and can potentially make more failing tests pass.

**Finding 3:** SeAPR substantially reduces patch executions on both partial and full patch-validation matrices and tends to perform slightly better with partial matrices.

### 5.4 RQ4: Impact of Additional Patch Pattern Information

In this RQ, we examine the impact of utilizing additional information for computing patch similarity. We execute the default configuration of SeAPR and compute patch similarity using both (1) the set of modified elements and (2) the applied patch patterns. We achieve this by looking specifically only at tools which apply predefined repair patterns, i.e., those tools categorized as template-based. Note that we collect the applied fix patterns as directly implemented by each tool. The studied template-based tools are TBar, KPar, Avatar, and PraPR with 18, 13, 19, and 18 repair patterns, respectively.3

Table 10 shows the result of this configuration evaluated against our default SeAPR setting. In this table, Column 1 presents the APR systems studied in this RQ; Column 2 presents the reduction results when using the method location information to compute patch similarity (i.e., the default SeAPR); Column 3 presents the reduction results when using only the new patch pattern information to compute patch similarity; finally, Column 4 presents the reduction results when combining both method and patch pattern information for computing patch similarity. From the table, we distinctly observe that SeAPR with only method location information or patch pattern information can both achieve nontrivial reductions. For example, the reduction rates range from 10.90% (TBar) to 62.25% (PraPR) when only using method location information, while ranging from 31.25% (KPar) to 61.96% (Avatar) when only using patch pattern information. Furthermore, the SeAPR with both method location and patch pattern information can achieve even higher reductions. For example, the reduction rates now range from 33.87% to 80.53% for all the four studied APR systems.

**Finding 4:** Patch pattern information further improves SeAPR’s performance on all studied repair tools by up to 29.75 pp. Furthermore, SeAPR with only patch pattern information can also achieve competitive patch reduction.

### 5.5 RQ5: Boosting SeAPR via Other APR Tools

Figure 7 shows the impact of historical information for our default configuration following Section 3.4. Note, we filter out all known correct patches from the historical information from other APR tools, since developers will stop the repair process after finding any correct patch with other tools, thus alleviating the need for SeAPR. Compared to Table 8, we observe improvements across most repair tools, up to 60.53 pp (Kali-A). Most notably, the only tool with originally negative performance (Cardumen) now has 34.15% reduction, i.e., over 41 pp improvement. On the other hand, the only instances of degradation come from GenProg-A, Avatar, TBar, and PraPR. The maximum degradation is only 14.71 pp (i.e., TBar degrades from 10.90% to -3.81%), while the degradation is less than 8 pp and the reductions are still positive for all other instances. Overall, such historical information can further boost SeAPR by 18.98 pp on average for all the studied APR tools.

**Finding 5:** Supplementing SeAPR with patch-execution information from other repair tools can further boost SeAPR. This extra historical information, compared with SeAPR’s default configuration, can further boost SeAPR by 18.98 pp on average and up to 60.53 pp.

---

3Please note we excluded FixMiner as we failed to dump its fixing-pattern information.
5.6 Discussion

5.6.1 Impact of Top Methods Considered. As shown in Section 4.5, to avoid degrading high-quality or plausible patches to the very end of all patches for different modified locations, we by default apply SeAPR to the Top 30 methods for the programs under test. In this section, we further investigate the effectiveness of SeAPR when applied to different numbers of top methods. Figure 8 shows the experimental results for applying SeAPR to Top 10 to Top 100 methods (with the interval of 10). In this figure, the x axis shows the number of top methods considered for SeAPR, while the y axis presents the corresponding reduction achieved by different APR systems (denoted by colored lines). According to the figure, we have the following observations. First, the reduction rates increase dramatically for almost all APR systems when increasing the number of top methods from 10 to 20. The reason is that SeAPR will have the chance to improve more patches when more top methods are considered for SeAPR. Second, the reduction rates for most APR systems will become stable when SeAPR is applied to beyond Top 20 methods. One reason is that for all other APR systems (except PraPR), they rarely patch more than 20 methods in a program under test, making the SeAPR results largely unchanged when considering more than Top 20 methods. Meanwhile, we can observe that even for PraPR (the purple line), which can patch far more methods than other APR systems, the reduction rate is still all positive, indicating the scalability and wide applicability of SeAPR.

5.6.2 SeAPR Overhead. The SeAPR algorithm is only related to the patch-validation phase, which is a pretty standard thing across APR tools. Therefore, the changes are minimal for applying SeAPR over existing APR tools. It is also important to analyze the overhead of SeAPR, since it does not pay off if SeAPR itself is extremely costly. Interestingly, we found that although SeAPR’s overhead often increases when considering more top methods, the average overhead for running SeAPR on each buggy version never exceeds 2s for any of the APR tools studied (even when considering 100 top methods). The reason is that our implementation has been highly optimized as shown in Section 3.2.4. Such overhead is negligible for APR tools, which typically take hours to fix a bug.

5.6.3 Nondeterminism in APR Tools. We currently report the results for one run since our goal is to speed up state-of-the-art APR techniques, while the recent state-of-the-art APR techniques are mostly deterministic, e.g., all the studied APR tools that can correctly fix over 20 bugs for Defects4J (including PraPR/Tbar/Fixminer/Aavatar) are deterministic. Meanwhile, we investigate the impact of APR non-determinism on SeAPR effectiveness, further rerun the experiments for the non-deterministic RSRepair-A and Kali-A tools for 5 times. The experimental results demonstrate that SeAPR can achieve an average reduction of 23.68%/4.88% for RSRepair-A/Kali-A, which is similar to the results shown in Table 8.

5.6.4 Metrics. As shown by recent work [38], using time costs for evaluating APR efficiency often depends on many random factors (such as the execution environments, test execution engines, and specific implementation choices) and can be quite unstable. Furthermore, for the same APR tool, the reduction in patch executions is largely proportional to the reduction in time cost since SeAPR is oblivious to the patch execution time distribution. Therefore, we followed the recommendation of the prior work [38] and have used the number of patch executions as our main metric. Meanwhile, it is also important to check if the reduction in terms of patch executions aligns well with the reduction in terms of time costs. Therefore, we further trace the detailed time cost reduction on an example tool, i.e., state-of-the-art PraPR. The experimental result shows that the reduction is 62.25% and 58.56% (including SeAPR overhead) in terms of patch executions and time cost, respectively. This further demonstrates the validity of our used metric.

5.7 Threats to Validity

5.7.1 Internal Validity. All of our results are dependent on the correctness of the implementation of all the studied techniques. We mitigate this threat by obtaining the source code of APR tools from their websites/authors. Also, three authors implemented three separate versions of SeAPR variants to perform differential testing to ensure the result correctness. Following prior APR work [50], three authors have participated in patch correctness checking to ensure the inspection correctness. Still, there may be human errors in the manual inspection process that may affect our findings.

5.7.2 External Validity. While our approach is generalizable to any type of patch-and-validate system, we only evaluate Java-based APR tools, which may skew results. To mitigate this threat we 1) studied a wide variety of APR tools, and 2) consider tools actively used in recent and related work. We also actively evaluate our technique on the most widely studied Defects4J dataset, with hundreds of real-world bugs. Adding more benchmark suites can definitely further reduce this threat. However, some of the studied APR tools cannot be easily applied to other benchmarks (due to implementation/design limitations of the original APR tools).

5.7.3 Construct Validity. A major threat to validity lies in our evaluation metrics. To mitigate this, we compute the number of patch executions recommended by recent work [38]. Meanwhile, we further discuss the reduction results in terms of time cost.

6 CONCLUSION

We have proposed the first self-boosted APR technique, SeAPR, which leverages the execution information of validated patches during APR to prioritize the remaining patches on-the-fly for faster APR. Our study on state-of-the-art APR systems and the widely used Defects4J benchmark demonstrates that (1) the default SeAPR can substantially speed up the studied APR techniques by up to 79% with negligible overhead, (2) SeAPR has stable performance when using different formulae for computing patch priority and different types of patch-execution matrices, (3) additional patch pattern information for patch similarity computation can further boost SeAPR, and (4) SeAPR can even utilize historical patch-execution information from other APR tools to boost current APR tools.

ACKNOWLEDGMENTS

We appreciate the insightful comments from all the anonymous reviewers. This work was partially supported by National Science Foundation under Grant Nos. CCF-2131943 and CCF-2141474, as well as Ant Group.