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 (**Se**lf-Boosted **A**utomated **P**rogram **R**epair), 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:
Samuel Benton, Yuntong Xie, Lan Lu, Mengshi Zhang, Xia Li, and Lingming Zhang. 2022. Towards Boosting Patch Execution On-the-Fly. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3510003.3510117

*The work was done during a remote summer internship at University of Illinois.
†Corresponding authors.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510117

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 (**Se**lf-Boosted **A**utomated **P**rogram **R**epair), 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 "$>$" to "$\geq$") 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 still can analyze the patch-execution information to provide useful hints about potential 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, while 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.

2.2 Cost Reduction for APR

Despite the promising future of APR, it can be extremely time consuming due to the generation and validation of a large number of possible patches. Actually, the patch validation cost has been shown to dominate the overall APR cost [30–33]. Therefore, researchers have also looked into various techniques to further speed up APR. To reduce *the validation time spent on each patch*, Ghanbari et al. [22] and Chen et al. [30] proposed to share the same JVM session across multiple patch executions; in this way, the patch loading and execution time can be substantially accelerated for both source-code and bytecode level APR techniques. In addition, researchers have also proposed to prioritize and reduce the test executions for each patch to reduce the validation time for each patch. For example, Qi et al. [47] proposed TrpAutoRepair to prioritize test executions for each patch based on historical information to falsify implausible patches faster; Mehne et al. [33] further proposed to reduce the number of test executions for each patch, since tests not covering the patched location(s) cannot help validate the patch. Our SeAPR technique is orthogonal to such existing techniques since they aim to reduce the execution cost for each patch while SeAPR aims to reduce the number of patch executions.

To reduce *the number of validated patches*, almost all existing APR techniques leverage fault localization and various other strategies to reduce the possible patch executions. Furthermore, many existing APR techniques also leverage other available dynamic or static information to prioritize patch executions to find the desired patches faster (e.g., based on various fault localization information [20, 23]). Despite various cost reduction techniques have been proposed, APR techniques are still rather time consuming for real-world programs [22]. In this paper, we propose the first technique to leverage on-the-fly patch execution information to help better prioritize patch executions. Note that our technique is orthogonal to all existing patch prioritization techniques and our experimental results demonstrate that our technique can substantially speed up state-of-the-art APR techniques with various original patch prioritization strategies.

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* $\mathbf{M_v}$ *defines the validation results of all tests against all patch candidates. In the matrix, each cell describes the validation result of test* $\mathbf{t} \in \mathbf{T}$ *against patch* $\mathbf{p} \in \mathbf{P}$*. Possible values for each cell are as follows: (1) - if* $\mathbf{t}$ *remains unvalidated, (2)* $\times$ *if* $\mathbf{t}$ *fails on* $\mathbf{p}$ *and (3)* $\checkmark$ *if* $\mathbf{t}$ *passes on* $\mathbf{p}$*.*

Patch ID	t_1	t_2	t_3
p_b (buggy ver.)	×	✓	×
p_1	×	×	×
p_2	✓	×	✓
p_3	×	×	×
p_4	✓	✓	✓

Patch ID	t_1	t_2	t_3
p_b (buggy ver.)	×	✓	×
p_1	×	-	-
p_2	✓	×	-
p_3	×	-	-
p_4	✓	✓	✓

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

Ideally the patch-validation matrix should be *full*, i.e., every cell should be ✓ or ×. 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., $\mathbf{p}_1$, $\mathbf{p}_2$, $\mathbf{p}_3$, and $\mathbf{p}_4$) 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., $\mathbf{p_b}$).

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

Patch ID	e_1	e_2	e_3	e_4	Modified Element(s)
p_1	✓	✓	✓	-	$\{e_1, e_2, e_3\}$
p_2	✓	✓	✓	✓	$\{e_1, e_2, e_3, e_4\}$
p_3	-	✓	✓	-	$\{e_2, e_3\}$
p_4	✓	-	-	-	$\{e_1\}$

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 $\mathbf{p}_1$ patches program elements $\{e_1, e_2, e_3\}$, the first three columns are ✓ for $\mathbf{p}_1$. 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.

$$\mathbf{P_h} = \{\mathbf{p} | \exists\ \mathbf{t},\ \mathbf{M_v}[\mathbf{p},\mathbf{t}] = ✓ \wedge \mathbf{M_v}[\mathbf{p_b},\mathbf{t}] = \times\} \tag{1}$$

$$\mathbf{P_l} = \{\mathbf{p} | \forall\ \mathbf{t},\ \mathbf{M_v}[\mathbf{p_b},\mathbf{t}] = \times \Rightarrow\ \mathbf{M_v}[\mathbf{p},\mathbf{t}] \neq\ ✓\} \tag{2}$$

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 ⊖ B = (A - B) ∪ (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:

$$s_h[\mathbf{p}] = \sum_{\mathbf{p'}} |\{\mathbf{e} | \mathbf{e} \in \mathbf{M_m}[\mathbf{p}] \cap \mathbf{M_m}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_h}\}| \tag{3}$$

$$s_l[\mathbf{p}] = \sum_{\mathbf{p'}} |\{\mathbf{e} | \mathbf{e} \in \mathbf{M_m}[\mathbf{p}] \cap \mathbf{M_m}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_l}\}| \tag{4}$$

$$d_h[\mathbf{p}] = \sum_{\mathbf{p'}} |\{\mathbf{e} | \mathbf{e} \in \mathbf{M_m}[\mathbf{p}] \ominus \mathbf{M_m}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_h}\}| \tag{5}$$

$$d_l[\mathbf{p}] = \sum_{\mathbf{p'}} |\{\mathbf{e} | \mathbf{e} \in \mathbf{M_m}[\mathbf{p}] \ominus \mathbf{M_m}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_l}\}| \tag{6}$$

Note that $\mathbf{M_m}[\mathbf{p}]$ 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 $|\mathbf{M_m}[\mathbf{p}] \cap \mathbf{M_m}[\mathbf{p'}]|$. 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.

$$\text{Ochiai} = \frac{s_h}{\sqrt{(s_h + d_h) * (s_h + s_l)}} \tag{7}$$

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 1s[1] (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

[1]Note 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 (s_h, s_l, d_h, d_l)
Output: Plausible patches: P_✓
1  begin
2      Initialize (s_h, s_l, d_h, d_l) for all patches
3      while P is not empty do
4          p ← pop(P) ; // pop the remaining patch with the highest priority
5          M_v ← execute(p, T); // validate p
6          if p is PLAUSIBLE then
7              P_✓ ← p; // put p into the resulting set for manual
                 inspection
8          r ← computePatchQuality(p, p_b, M_v)
           // Incrementally update the similarity tuples for the remaining
              patches
9          for p' ∈ P do
10             if r = HIGH − QUALITY then
11                 s_h[p'] += |M_m[p] ∩ M_m[p']|
12                 d_h[p'] += |M_m[p] ⊖ M_m[p']|
13             if r = LOW − QUALITY then
14                 s_l[p'] += |M_m[p] ∩ M_m[p']|
15                 d_l[p'] += |M_m[p] ⊖ M_m[p']|
16         P ← computePriorityScore(P, s_h, d_h, s_l, d_l)
```

corresponding APR tools. Then, SeAPR executes the patch against the original test suite and stores the patch execution results into the patch validation matrix $\mathbf{M_v}$ (Line 5). If $\mathbf{p}$ is a plausible patch, it will be stored in the resulting set $\mathbf{P}_\checkmark$ for manual inspection (Lines 6-7). To help with on-the-fly patch prioritization, SeAPR computes the patch quality information for the current patch following Section 3.2.1 (Line 8). Next, SeAPR goes through all the remaining patches to update their similarity tuples following Section 3.2.2 (Lines 9-15). Note that all remaining patches will be compared with the newly executed patch to incrementally update their corresponding similarity tuples. Lastly, the priority scores for all remaining patches will be updated based on the updated similarity tuples following Section 3.2.3 (Line 16). In this way, the algorithm will proceed until all patches have been validated or the developers find the desired patch.

Note that the time complexity of the SeAPR algorithm is $O(n^2)$ at first glance (n denotes the number of patches considered for SeAPR), since all the remaining patches need to be updated after each patch execution. Meanwhile, during our implementation, we realize that the similarity scores do not need to be updated for each remaining patch; instead, we can cluster all remaining patches based on the set of program elements they modify, since all patches with the same set of modified elements will have the same priority. In this way, the time complexity can be reduced to $O(nm)$, where m denotes the number of patch clusters with the same modified element sets. Given $m << n$ in practice, our actual SeAPR implementation incurs negligible overhead.

3.2.5 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_h", "d_h", "s_l", and "d_l" represent the *accumulated* similarity tuples per patch (initialized as 1s); 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, $\mathbf{p}_1$ (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 $\mathbf{p}_1$ has been executed, we can update the similarity tuple for each remaining patch. For example, for $\mathbf{p}_2$, the set intersection and symmetrical set difference with $\mathbf{p}_1$ is $\{e_1, e_2, e_3\} \cap \{e_1, e_2, e_3, e_4\} = \{e_1, e_2, e_3\}$ and $\{e_1, e_2, e_3\} \ominus \{e_1, e_2, e_3, e_4\} = \{e_4\}$, respectively. Therefore, since $\mathbf{p}_1$ is a low-quality patch, s_l increments by 3 and d_l increments by 1 for $\mathbf{p}_2$, resulting in the tuple (s_h=1+0, d_h=1+0, s_l=1+3, d_l=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, $\mathbf{p}_4$, is selected for the next patch execution.

In the second iteration, $\mathbf{p}_4$ gets executed (marked in gray) as shown in Table 4. Note that $\mathbf{p}_4$ is a plausible patch that can pass all tests. Therefore, the developers can immediately start manual inspection to check if $\mathbf{p}_4$ is the correct patch. Of course, the patch execution can still continue if $\mathbf{p}_4$ is not the correct patch. Continuing the algorithm, the remaining patches will be further compared with the newly executed $\mathbf{p}_4$ to update their similarity tuples. For example, for $\mathbf{p}_2$, the set intersection and symmetrical set difference with $\mathbf{p}_4$ is $\{e_1, e_2, e_3, e_4\} \cap \{e_1\} = \{e_1\}$ and $\{e_1, e_2, e_3, e_4\} \ominus \{e_1\} = \{e_2, e_3, e_4\}$, respectively. Since $\mathbf{p}_4$ is a high-quality patch, $\mathbf{p}_2$'s tuple is updated by incrementing s_h by 1 and d_h by 3, resulting in the tuple (s_h=1+1, d_h=1+3, s_l=4+0, d_l=2+0). In this way, we can compute the scores for all remaining patches.

ID	Quality	Match	Differ	s_h	d_h	s_l	d_l	Score
$\mathbf{p}_1$	**Low**	-	-	-	-	-	-	-
$\mathbf{p}_2$	High	$\{e_1, e_2, e_3\}$	$\{e_4\}$	1+0	1+0	1+3	1+1	0.32
$\mathbf{p}_3$	Low	$\{e_2, e_3\}$	$\{e_1\}$	1+0	1+0	1+2	1+1	0.35
$\mathbf{p}_4$	Plausible	$\{e_1\}$	$\{e_2, e_3\}$	1+0	1+0	1+1	1+2	0.41

Table 3: SeAPR step-by-step when processing $\mathbf{p}_1$

ID	Quality	Match	Differ	s_h	d_h	s_l	d_l	Score
$\mathbf{p}_1$	Low	-	-	-	-	-	-	-
$\mathbf{p}_4$	**Plausible**	-	-	-	-	-	-	-
$\mathbf{p}_2$	High	$\{e_1\}$	$\{e_2, e_3, e_4\}$	1+1	1+3	4+0	2+0	0.33
$\mathbf{p}_3$	Low	$\{\emptyset\}$	$\{e_1, e_2, e_3\}$	1+0	1+3	3+0	2+0	0.22

Table 4: SeAPR step-by-step when processing $\mathbf{p}_4$

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

Patch ID	r_1	r_2	r_3	SeAPR Features	
				Modified Element(s)	Pattern(s)
$\mathbf{p}_1$	✓	-	-	$\{e_1, e_2, e_3\}$	$\{r_1\}$
$\mathbf{p}_2$	-	✓	-	$\{e_1, e_2, e_3, e_4\}$	$\{r_2\}$
$\mathbf{p}_3$	-	-	✓	$\{e_2, e_3\}$	$\{r_3\}$
$\mathbf{p}_4$	-	✓	-	$\{e_1\}$	$\{r_2\}$

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., $s_h[\mathbf{p}]$ in Equation (3) becomes:

$$s_h[\mathbf{p}] = \sum_{\mathbf{p'}} |\{\mathbf{e} | \mathbf{e} \in \mathbf{M_m}[\mathbf{p}] \cap \mathbf{M_m}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_h}\}| + \sum_{\mathbf{p'}} |\{\mathbf{r} | \mathbf{r} \in \mathbf{M_p}[\mathbf{p}] \cap \mathbf{M_p}[\mathbf{p'}] \wedge \mathbf{p'} \in \mathbf{P_h}\}| \quad (8)$$

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 are run in isolation. Our basic idea is that *the execution results of* ***other repair tools on the same program*** *can be used to further boost the current repair tool under SeAPR.* In particular, we use the repair information from *all but one* repair tools to initialize the priority score of all patches. For example, when applying SeAPR to TBar on Chart-1, all patch executions results of other tools on Chart-1 will be treated as the already executed patches to initialize the priority scores of all TBar patches on Chart-1. With the priority scores initialized, SeAPR starts with the most prioritized patch and follows the algorithm outlined in Algorithm 1, updating the already initialized priority scores of each patch. In this way, SeAPR can get a jumpstart for faster validation.

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 $\mathbf{P_h}$ (the set of high-quality patches, which is closely related to SeAPR effectiveness) is not necessarily related to the number of failed tests since $\mathbf{P_h}$ 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.

Subject	Name	# Bugs	# Tests	LOC
Chart	JFreeChart	26	2,205	96K
Lang	Apache Lang	65	2,245	22K
Math	Apache Math	106	3,602	85K
Time	Joda-Time	27	4,130	28K
Mockito	Mockito framework	38	1,366	23K
Closure	Google Closure compiler	133	7,927	90K
	Total	**395**	**21,475**	**344K**

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.

Tool Category	Tool(s)
Constraint-based	~~ACS~~ [26], Cardumen [7], ~~DynaMoth~~ [6], ~~Nopol~~ [25]
Heuristic-based	Arja [10], GenProg-A [10], JGenProg [9], JKali [9], JMutRepair [9], Kali-A [10], RSRepair-A [10], ~~Simfix~~ [8]
Template-based	Avatar [5], FixMiner [36], KPar [37], PraPR [22], TBar [4]

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 practice 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 $\frac{P_{baseline}-P_{new}}{P_{baseline}}$. $\mathbf{P}_{baseline}$ represents the position of the first desired (i.e., plausible/correct) patch, pre-prioritization. $\mathbf{P}_{new}$ 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 ordering 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.

APR Systems	Orig. Rank	SeAPR Rank	Diff.	Reduction
Arja	203.22	121.83	81.39	40.05%
Avatar	57.52	25.96	31.56	54.87%
Cardumen	10.25	11.00	-0.75	-7.32%
FixMiner	85.26	48.13	37.13	43.55%
GenProg-A	226.21	47.71	178.50	78.91%
JGenProg	10.67	9.83	0.84	7.81%
JKali	6.50	4.75	1.75	26.92%
JMutRepair	25.50	23.67	1.83	7.19%
Kali-A	25.33	23.72	1.61	6.36%
KPar	76.59	63.14	13.45	17.56%
RSRepair-A	105.55	82.50	23.05	21.84%
TBar	55.19	49.18	6.01	10.90%
PraPR	2052.8	774.87	1277.93	62.25%

Table 8: Default SeAPR results for plausible patches

APR Systems	Orig. Rank	SeAPR Rank	Diff.	Reduction
Arja	355.00	96.00	259.00	72.96%
Avatar	44.50	8.83	35.67	80.15%
Cardumen	17.00	17.00	0.00	0.00%
FixMiner	3.50	3.50	0.00	0.00%
JMutRepair	28.00	13.00	15.00	53.57%
KPar	58.43	21.00	37.43	64.06%
TBar	33.00	17.05	15.95	48.33%

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

```
@@ -911,7 +911,6 @@ ...
     public void setRangeAxis(int index, ValueAxis axis) {
-        setRangeAxis(index, axis, true);
     }
```

(a) Low-quality patch

```
@@ -161,7 +161,10 @@
    protected int indexOf(Object object) {
         ...
-       return -1;
+       if (object == null) {
+           throw new IllegalArgumentException("Null 'object' argument."); }
+       return -1;
     }
```

(b) Plausible patch

Figure 1: Arja Chart-19 patches

```
@@ -1456,7 +1456,7 @@
NodeMismatch checkTreeEqualsImpl(Node node2) {
    ...
      res = n.checkTreeEqualsImpl(n2);
      if (res != null) {
-       return res;
+       return null;
      }
    }
    ...
```

(a) High-quality patch

```
@@ -1450,7 +1450,7 @@
NodeMismatch checkTreeEqualsImpl(Node node2) {
    ...
    Node n, n2;
    for (n = first, n2 = node2.first;
         res == null && n != null;
-        n = n.next, n2 = n2.next) {
+        n = n, n2 = n2.next) {
      if (node2 == null) {
        throw new IllegalStateException();
      }
    ...
```

(b) Plausible patch

Figure 2: PraPR Closure-120 patches

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)` 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:

```
@@ -123,7 +123,6 @@
public class StrBuilder implements Cloneable {
     public StrBuilder(String str) {
         ...
             buffer = new char[CAPACITY];
         } else {
             buffer = new char[str.length() + CAPACITY];
-            append(str);
         }
     }...
```

(a) Non-matching High-quality patch

```
@@ -1113,7 +1113,7 @@
public class StrBuilder implements Cloneable {
     private void deleteImpl(int startIndex, int endIndex, int len) {
-        System.arraycopy(buffer, endIndex, buffer, startIndex, size - endIndex);
+        System.arraycopy(buffer, endIndex, buffer, startIndex,
+                         capacity() - endIndex);
         size -= len;
     }...
```

(b) Plausible patch

Figure 3: PraPR Lang-61 patches

```
@@ -530 +530 @@
public final class MathUtils {
    public static boolean equals(double[] x, double[] y){
        ...
        for (int i = 0; i < x.length; ++i) {
-           if (!equals(x[i], y[i])) {
+           if ((!equals(x[i], y[i])) && (x.length != y.length)) {
                return false;
        ...
```

(a) Low-quality patch

```
@@ -530 +530 @@
public final class MathUtils {
    public static boolean equals(double[] x, double[] y){
        ...
        for (int i = 0; i < x.length; ++i) {
-           if (!equals(x[i], y[i])) {
+           if (!equals(x[i], y[i], i)) {
                return false;
        ...
```

(b) Plausible patch

Figure 4: TBar Math-63 patches

Example 3: Figure 3b presents the only plausible patch produced by PraPR on Lang-61, which modifies method `deleteImpl(int, int, int)`. However, various *non-plausible high-quality patches* generated by PraPR modify a number of other methods, such as the patch shown in Figure 3a modifying `StrBuiler(String)`. After detecting such high-quality patches, other patches sharing the same modified elements are prioritized, causing some patches originally ranked below the plausible patch to be executed earlier. As a result, the rank of the plausible patch is degraded from 182nd to 347th.
Example 4: The only plausible patch which is also the only high-quality patch generated by TBar for Math-63 is shown in Figure 4b. However, as depicted in Figure 4a, there are many other patches which modify the same method as the plausible patch but cannot make any original failing tests pass. Many of such low-quality patches are originally ranked above the only plausible patch. As a result, validation of these patches will degrade the execution of the plausible patch. Eventually, the plausible patch initially ranked at 28th is then degraded to 55th.

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).

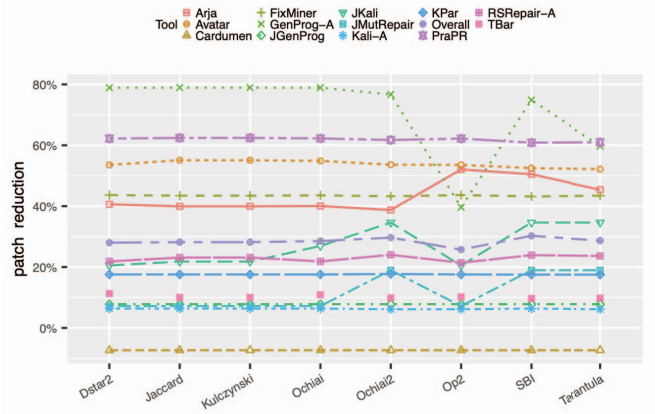

Figure 5: Impact of different formulae on patch reduction

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 formulae [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 Kulczynski and Jaccard). Furthermore, for different SBFL formulae, the overall patch reduction on all the studied 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 formulae is only 4.5 percentage points **(pp)**. Op2 performs worse than the other formulae because it mainly considers the s_h information while largely ignoring other valuable information from the similarity 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%, demonstrating 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 overall 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

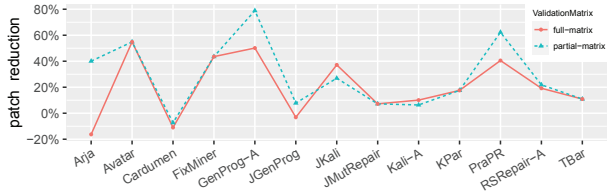

Figure 6: Impact of validation matrices on SeAPRs

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 high 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.

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

	Method	Pattern	Method+Pattern
Avatar	54.87%	61.96%	80.53%
KPar	17.56%	31.25%	33.87%
TBar	10.90%	35.73%	40.65%
PraPR	62.25%	42.27%	66.88%

Table 10: SeAPR with patch pattern information

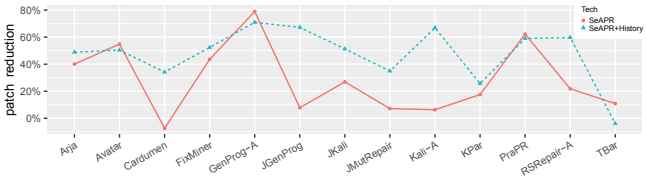

Figure 7: SeAPR with historical APR results from other tools

Figure 8: Impact of top method number on 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.

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/Avatar) are deterministic. Meanwhile, to investigate the impact of APR non-determinism on SeAPR effectiveness, we 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.

REFERENCES

[1] "Tricentis reports," 2020. [Online]. Available: https://www.tricentis.com/resources/software-fail-watch-5th-edition/

[2] C. Boulder, "University of cambridge study," https://www.roguewave.com/company/news/2013/university-of-cambridge-reverse-debugging-study, 2013, accessed: Jan. 8, 2019.

[3] S. Wang, M. Wen, B. Lin, X. Mao, H. Wu, D. Zou, H. Jin, and Y. Qin, "Automated Patch Correctness Assessment: How Far are We?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 1166–1178.

[4] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "TBAR: Revisiting template-based automated program repair," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[5] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyande, "AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations," in *Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, 2019.

[6] T. Durieux and M. Monperrus, "DynaMoth: Dynamic code synthesis for automatic program repair," in *Proceedings - 11th International Workshop on Automation of Software Test, AST 2016*, 2016.

[7] M. Martinez and M. Monperrus, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Lecture Notes in Computer Science*, 2018.

[8] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[9] M. Martinez and M. Monperrus, "ASTOR: A program repair library for Java (Demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[10] Y. Yuan and W. Banzhaf, "ARJA: Automated Repair of Java Programs via Multi-Objective Genetic Programming," *IEEE Transactions on Software Engineering*, 2018.

[11] Y. Lou, J. Chen, L. Zhang, D. Hao, and L. Zhang, "History-driven build failure fixing: How far are we?" in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[12] M. Wu, L. Zhang, C. Liu, S. H. Tan, and Y. Zhang, "Automating CUDA synchronization via program transformation," in *IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019.

[13] J. Jiang, L. Ren, Y. Xiong, and L. Zhang, "Inferring program transformations from singular examples via big code," in *IEEE/ACM International Conference on Automated Software Engineering, ASE 2019*, 2019.

[14] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016, pp. 298–312.

[15] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," *International Symposium on Software Testing and Analysis*, pp. 24–36, 2015.

[16] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 2015, pp. 166–178.

[17] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 727–739.

[18] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Proceedings of the International Conference on Software Engineering*, 2016.

[19] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[20] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proceedings - Testing: Academic and Industrial Conference Practice and Research Techniques, TAIC PART-Mutation 2007*, 2007.

[21] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *20th IEEE/ACM International Conference on Automated Software Engineering, ASE 2005*, 2005.

[22] A. Ghanbari, S. Benton, and L. Zhang, "Practical program repair via bytecode mutation," in *ISSTA 2019 - Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[23] M. Wen, J. Chen, R. Wu, D. Hao, and S. C. Cheung, "Context-aware patch generation for better automated program repair," in *Proceedings - International Conference on Software Engineering*, vol. 2018-January, 2018.

[24] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," *Proceedings of International Conference on Software Engineering*, pp. 3–13, 2012.

[25] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, 2017.

[26] Y. Xiong, J. Wang, R. Yan, J. Zhang, S. Han, G. Huang, and L. Zhang, "Precise condition synthesis for program repair," in *IEEE/ACM 39th International Conference on Software Engineering*, 2017, pp. 416–426.

[27] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale," in *IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 269–278.

[28] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "Elixir: Effective object-oriented program repair," in *IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 648–659.

[29] Y. Lou, A. Ghanbari, X. Li, L. Zhang, H. Zhang, D. Hao, and L. Zhang, "Can automated program repair refine fault localization? a unified debugging approach," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020.

[30] L. Chen, Y. Ouyang, and L. Zhang, "Fast and precise on-the-fly patch validation for all," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1123–1134.

[31] C. Le Goues, S. Forrest, and W. Weimer, "Current challenges in automatic software repair," *Software quality journal*, vol. 21, no. 3, pp. 421–443, 2013.

[32] W. Weimer, Z. P. Fry, and S. Forrest, "Leveraging program equivalence for adaptive program repair: Models and first results," in *IEEE/ACM International Conference on Automated Software Engineering*, 2013, pp. 356–366.

[33] B. Mehne, H. Yoshida, M. R. Prasad, K. Sen, D. Gopinath, and S. Khurshid, "Accelerating search-based program repair," in *International Conference on Software Testing, Verification and Validation*, 2018, pp. 227–238.

[34] Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller, "Automated fixing of programs with contracts," *IEEE Transactions on Software Engineering*, vol. 40, no. 5, 2014.

[35] R. van Tonder and C. L. Goues, "Static automated program repair for heap properties," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[36] A. Koyuncu, K. Liu, T. Bissyandé, D. Kim, J. Klein, M. Monperrus, and Y. LeTraon, "FixMiner: Mining relevant fix patterns for automated program repair," *Empirical Software Engineering*, 2020.

[37] K. Liu, A. Koyuncu, T. Bissyande, D. Kim, J. Klein, and Y. LeTraon, "You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems," *Proceedings of 12th IEEE International Conference on Software Testing, Verification and Validation*, 2019.

[38] K. Liu, S. Wang, A. Koyuncu, K. Kim, T. F. Bissyande, D. Kim, P. Wu, J. Klein, X. Mao, and Y. L. Traon, "On the efficiency of test suite based program repair a systematic assessment of 16 automated repair systems for java programs," in *Proceedings of International Conference on Software Engineering*, 2020.

[39] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proceedings of International Conference on Software Engineering*, 2013.

[40] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *IEEE International Conference on Software Analysis, Evolution, and Reengineering*, vol. 1, 2016, pp. 213–224.

[41] L. De Moura and N. Bjørner, "Z3: An efficient SMT Solver," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 4963 LNCS, 2008.

[42] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "CoCoNuT: Combining context-aware neural translation models using ensemble for program repair," *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pp. 101–114, 2020.

[43] Y. Ding, B. Ray, and V. J. Hellendoorn, "Patching as Translation : the Data and the Metaphor," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020, pp. 327–338.

[44] S. Benton, X. Li, Y. Lou, and L. Zhang, "On the Effectiveness of Unified Debugging: An Extensive Study on 16 Program Repair Systems," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, 2020.

[45] ——, "Evaluating and improving unified debugging," *IEEE Transactions on Software Engineering*, 2021.

[46] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *International Symposium on Software Testing and Analysis*, 2013.

[47] Y. Qi, X. Mao, and Y. Lei, "Efficient automated program repair through fault-recorded testing prioritization," in *IEEE International Conference on Software Maintenance, ICSM*, 2013.

[48] J. A. Jones, M. J. Harrold, and J. Stasko, "Visualization of test information to assist fault localization," in *Proceedings of International Conference on Software Engineering*, 2002.

[49] M. Zhang, X. Li, L. Zhang, and S. Khurshid, "Boosting spectrum-based fault localization using pagerank," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 261–272.

[50] L. Gazzola, D. Micucci, and L. Mariani, "Automatic Software Repair: A Survey," *IEEE Transactions on Software Engineering*, vol. 45, no. 1, 2019.