Revisiting Object Similarity-based Patch Ranking in Automated Program Repair: An Extensive Study

Ali Ghanbari
alig@iastate.edu
Iowa State University
Ames, IA, USA

ABSTRACT

Test-based generate-and-validate automated program repair (APR) systems often generate plausible patches that pass the test suite without fixing the bug. So far, several approaches for automatic assessment of the APR-generated patches are proposed. Among them, dynamic patch correctness assessment relies on comparing runtime information obtained from the program before and after patching. Object similarity-based dynamic patch ranking approaches, specifically, capture system state snapshots after the impact point of patches and express behavior differences in term of object graphs similarities. Dynamic approaches rely on the assumption that, when running the originally passing test cases, the correct patches will not alter the program behavior in a significant way, but such patches will significantly change program behavior for the failing test cases.

This paper presents the results of an extensive empirical study on two object similarity-based approaches, *i.e.*, ObjSim and CIP, to rank 1,290 APR-generated patches, used in previous APR research. We found that although ObjSim outperforms CIP, in terms of the number of patches ranked in top-1 position, it still does not offer an improvement over random baseline ranking, representing the setting with no automatic patch correctness assessment in place. This observation warrants further research on the validity of the assumptions underlying these two techniques and the techniques based on similar assumptions.

CCS CONCEPTS

• **Software and its engineering → Software testing and debugging**.

KEYWORDS

Automated Program Repair, Patch Ranking, Object Similarity

ACM Reference Format:
Ali Ghanbari. 2022. Revisiting Object Similarity-based Patch Ranking in Automated Program Repair: An Extensive Study. In *International Workshop on Automated Program Repair (APR'22), May 19, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3524459.3527354

1 INTRODUCTION

Automated program repair (APR) techniques are categorized into different classes [20]. The majority of current APR techniques belong to the search-based class, aka generate-and-validate (G&V), that attempt to fix the bugs through evolutionary search [19, 44], heuristic fix templates [8, 15], code grafting [1, 35], or random mutation [9, 29] and validating the generated patches using certain checks. The majority of G&V techniques are *test-based* [8, 9, 15, 18, 19, 22, 24, 25, 30, 35], *i.e.*, they validate the generated patches by running the test suite against the patches, capable of targeting a wide rage of defects, while others are purely static [16, 26, 33], targeting compilation errors or specific classes of bugs.

In the context of test-based G&V APR techniques, a patch passing all the test cases is referred to as a *plausible* patch. A plausible patch is called *correct* if it eliminates the bug. APR tools generate many plausible patches that do not fix the bug [20, 31], and such patches are called *incorrect* patches. Users of APR tools must manually examine many plausible patches to find and apply a correct one. In order to alleviate such manual effort, several techniques are developed for assessing the correctness of plausible patches [27]. Some techniques classify the plausible patches into likely correct (or incorrect) ones, so the users may need to inspect only a subset of the patches [3, 5, 12, 21, 32, 34, 37–39, 41, 43]. Other techniques rank the plausible patches based on their likelihood of being correct, so the users may find a correct patch faster [14, 44]. The techniques rely on static code features [5, 41], run-time information [3, 14, 21, 38, 39, 43, 44], or use a combination of both [34].

Dynamic patch correctness assessment techniques work by comparing the runtime behavior of the patched program with its unpatched version. The approaches differ from one another primarily by the way in which they capture and quantify the difference in program behavior. For example, PATCH-SIM [38] relies on path spectra [17] as an abstraction of the program behavior and uses Longest Common Subsequence algorithm to quantify the differences. Another family of approaches [3, 39] use dynamically inferred invariants as an abstraction of the program behavior and use syntactic distance metrics to quantify the differences. ObjSim [14] and CIP [44] quantify the behavior change by comparing system state snapshots at the exit points of patched methods. The two techniques are designed for programs written in JVM-based programming languages, wherein the system state, observable from the test code, is captured in the form of object graphs, hence the name *object similarity*-based. Dynamic patch correctness assessment approaches are based on the assumption that correct patches do not alter the program behavior significantly, when running the originally passing test cases, but such patches result in larger program behavior differences, when running failing tests [3, 14, 38, 39].

APR'22, May 19, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9285-3/22/05.
https://doi.org/10.1145/3524459.3527354

Ranking of plausible patches makes sense only for APR systems continue searching even after finding the first plausible patch, while classification can be applied in either case at the cost of possibly leaving the user with the task of manually checking the patches within each class of likely correct and likely incorrect patches. We believe there is a need for both kind of assessment methods. The pursuit for fixing more complex bugs (*e.g.*, in Repairnator [28]) and finding more correct patches within the search space of existing repair tools (*e.g.*, in UniAPR [4]) implies that automated ranking of plausible patches is of practical importance. However, we emphasize that evaluating the performance of patch ranking techniques must be accompanied by a comparison with random ranking, representing the setting with no automatic patch correctness assessment in place. Given the fact that manual inspection of patches is time-consuming and patch correctness assessment involves semantic reasoning, which is not automatable [23], we believe that any automatic patch ranking technique, slightly better than random ranking, would be highly valuable.

This paper focuses on object similarity-based dynamic techniques for ranking APR-generated plausible patches, namely, ObjSim and CIP. ObjSim is an improved variant of the technique introduced by Ghanbari [14] (see §2) and CIP is sub-system of ARJA-E APR tool [44]. We present an empirical study with two main goals: (1) compare ObjSim and CIP on a significantly larger data set of patches than that of the original works [14, 44]; (2) compare the champion technique to a random ranking baseline obtained by *hypergeometric probability* [2] of a correct patch appearing in top-1 or top-2, representing a setting with no automated patch correctness assessment mechanism in place. Specifically, we rank 1,290 patches generated by 29 APR systems for real-world bugs from Defects4J data set [10], used in prior research [11, 34, 38, 41]. We observed that in 19.8% (47.7%) of the cases, ObjSim ranks the correct patch in top-1 (top-2) position, meanwhile CIP ranks 9.6% (42.1%) of the correct patches in top-1 (top-2) position. Since these observations are of statistical significance, we conclude that ObjSim outperforms CIP in terms of the number of patches ranked in top-1 (top-2) position. We then compared ObjSim with random baseline ranking, and the comparison could be used to measure the amount of improvement we could possibly get from such a technique in real-world applications. In other words, a ranking technique outperforming random ranking can potentially help popularity of APR in everyday programming by helping the users find the correct patch(es) with minimal effort. As per our analysis, ObjSim does not offer any improvement over random ranking. This observation warrants further research on the validity of the assumptions underlying these two techniques and the techniques based on similar assumptions.

In summary, the main contributions of this study are:

(1) Empirical study shedding light on the ranking performance of two dynamic patch ranking systems, ObjSim and CIP. We provide data supporting the claim that ObjSim is more effective than CIP.
(2) We compare the ranking effectiveness of the champion technique to random ranking, and provide empirical evidence that it offers no improvement over random ranking.

Studied software artifacts, *i.e.*, ObjSim and CIP, and our data, are publicly available [13].

2 BACKGROUND

In this section, we describe ObjSim, an improved version of the technique introduced in [14], capable of handling multi-hunk patches, and CIP [44]. Both techniques depend on run-time information and use object similarity to assess the quality of plausible patches.

2.1 The ObjSim Approach

ObjSim [14] is a lightweight automatic patch ranking system based on object similarity, designed to handle patches applied to single program source location. However, patches can extend to multiple locations, commonly referred to as multi-hunk patches. In this paper, we extend ObjSim to handle multi-hunk patches. ObjSim assumes that patches resulting in a system state more similar to that of the unpatched program, when executing passing test cases, and less similar state on failing tests, are more likely to be correct. The tool instruments the patched methods and monitors the execution of the program to record system state snapshots at their exit points, before and after patching. The technique has four main steps described below.

Compilation and pre-processing. ObjSim compiles the original and the patched version of the project (if the patch is not already in bytecode format) to obtain buggy and patched class files. The tool obtains the full name of the patched methods by parsing the patch diff files, extracting patched line numbers of the patched statement(s), and retrieving the method names from compiled class files based on the line numbers.

Profiling. Unlike the older version of ObjSim [14], the improved version of the tool does not depend on the users to provide suspiciousness values for the patched locations, instead, it finds out which test methods cover which lines and calculates the Ochiai spectrum-based suspiciousness values [36] for the patched locations based on that. In this step, the tool also identifies the fields accessed by the patched methods directly or indirectly. This information is used for compactly capturing system state snapshots.

Instrumentation and test execution. ObjSim runs the original and patched program under covering passing and failing test cases and records the system state at the exit point(s) of the patched methods. By system state, we mean: (1) the set of object graphs reachable from the escaping objects passed to the patched method; (2) returned object; and (3) the static fields accessed by the method.

Similarity analysis and patch ranking. Once the system state snapshot pairs for the buggy program and its patched version are obtained, the tool calculates the similarities between snapshots in a pairwise manner and averages them. For each patch covered by a passing test case, an average distance of $dist_p$ is calculated (note that a single test case might result in multiple snapshots as it might call the patched method multiple times). Similarly, for a patch covered by a failing test case, an average distance of $dist_f$ is calculated. The average distances calculated for each patch, associated with each test case, are used for ranking as sketched below.

In order to rank the patches, ObjSim groups the patches based on the set of test cases covering them. The patches within each group are sorted based on the average similarity measures described above. Roughly speaking, within each group, for a given passing test case, the tool assigns the patches scores depending on how large their $dist_p$ are relative to each other. Similarly, for a given failing test, ObjSim assigns the patches scores depending on how large their $dist_f$ values are relative to each other. The scores for each patch are averaged so as to sort the patches in descending order of their average score. The groups are sorted based on the maximum Ochiai suspiciousness value of the patches within each group. Once sorting is finished, the patches get ranked with tied patches receiving worst-case rank.

To keep the paper self-contained, we describe how ObjSim calculates the similarity between two object graphs. Given two objects s_1 and s_2, $dist_{\text{ObjSim}}(s_1, s_2)$ is computed *via* DFS traversal of the object graphs reachable from s_1 and s_2 and accumulating distances of "sub-objects" of the objects, recursively. More concretely, $dist_{\text{ObjSim}}$ is defined recursively as follows.

- $dist_{\text{ObjSim}}(s_1, s_2) = 0$ if s_1, s_2 are equal references or equal primitive-typed objects.
- $dist_{\text{ObjSim}}(s_1, s_2) = 1$ if s_1, s_2 are unequal primitive-typed objects of the same type.
- $dist_{\text{ObjSim}}(s_1, s_2)$ = Levenshtein distance between s_1 and s_2, if s_1, s_2 are arrays of the same component type.
- $dist_{\text{ObjSim}}(s_1, s_2) = \sum_{i=1}^{n} dist_{\text{ObjSim}}(v(f_i, s_1), v(f_i, s_2))$ if s_1, s_2 are objects of the same type τ and $f_1, \ldots, f_n$ are the fields declared or inherited by τ. Furthermore, $v(f, o)$ is defined to be the value of field f for object o.
- $dist_{\text{ObjSim}}(s_1, s_2) = +\infty$ if s_1, s_2 are objects of different types.

2.2 The CIP Approach

CIP is a post-processing system for alleviating the effect of incorrect patches generated by the ARJA-E APR system [44]. We refer the reader to the original publication [44] for more details, yet we describe CIP here to make the paper self-contained.

CIP is a two-tier patch correctness assessment approach. First, it classifies the patches into two classes (*i.e.*, likely correct and likely incorrect). The classification is based on object distance, that is, the patches that do not alter the system state compared to that of resulted from the original program (*i.e.*, an object distance of 0) on passing test cases, are classified as likely correct, while the rest are classified as likely incorrect.

Second, it ranks the likely incorrect patches by applying three heuristics in tandem: (1) cumulative Ochiai suspiciousness values of the patched location(s); (2) average normalized object distance values calculated for the patched method(s) on passing tests; and (3) a heuristic ranking of the repair operations used to transform the program in question.

In order to compute the object similarities used for classification and ranking, CIP first applies the patch on a copy of the original program, obtaining the patched method full name and signature, and compiling the original and patched version of the program. The tool then runs the original, buggy program on (originally) passing test cases and records input-output pairs for the patched methods, *i.e.*, the input system state upon which the patched methods are invoked and the system state resulting from running the patched methods. As in ObjSim, by system state, we mean the set of object graphs reachable from the objects passed to the patched method and the static fields accessed by the method. Next, CIP uses Java reflection API to invoke the patched methods upon the recorded inputs and records the output produced by the methods. Finally, the recorded output object graphs are compared with each other to calculate their distance.

The way CIP calculates object distances is rather different than that of ObjSim, as CIP (except in the calculation of Levenshtein distance of string values) takes into account *how different* primitive values are from each other. More formally, given two object references s_1, s_2, CIP calculates normalized distance between the object graphs reachable from the references using $dist_{\text{CIP}}(s_1, s_2) = \dfrac{d(s_1, s_2)}{1 + d(s_1, s_2)}$ where $d(s_1, s_2)$ is defined recursively as follows.

- $d(s_1, s_2) = 1$, if s_1, s_2 are unequal Boolean values.
- $d(s_1, s_2) = |s_1 - s_2|$, if s_1, s_2 are numeric values.
- $d(s_1, s_2)$ = Levenshtein distance between s_1 and s_2, if s_1, s_2 are strings.
- $d(s_1, s_2) = \sum_{i=1}^{n} d(s_1[i], s_2[i])$, if s_1, s_2 are arrays of length n. $s[i]$ denotes the i^{th} component of the array s.
- $d(s_1, s_2) = \sum_{i=1}^{n} d(v(f_i, s_1), v(f_i, s_2))$, if s_1, s_2 are objects of type τ and $f_1, \ldots, f_n$ are the fields declared or inherited by τ and accessed in the original program. Furthermore, $v(f, o)$ is defined to be the value of field f for object o.
- $d(s_1, s_2) = 0$, otherwise.

2.3 Example

We illustrate how ObjSim and CIP work through an example. Figure 1 depicts a faulty Java implementation of a program that given a string `str`, an integer `k`, and a character `c` is intended to return `true` if `c` occurs in `str` at any index 0 ≤ `j` ≤ `k` and `false`, otherwise. Table 1 lists two test cases that exercise the buggy method `find` with different inputs, one of which reveals the fault. T-1 is a failing test, while T-2 is a passing test case. Figure 2 illustrates two possible patches that a typical G&V APR tool (*e.g.*, [25]) may produce in an attempt to fix `Finder`. While both patches result in code that passes both T-1 and T-2 (*i.e.*, they are plausible), only one of them is correct (*i.e.*, Patch 1).

ObjSim instruments the buggy method(s), as well as its patched version, and runs it against (covering) passing test cases so that snapshots of the system state at the exit points of the method(s) are recorded. The recorded snapshots corresponding to the patches are then compared with the snapshots corresponding to the original program to calculate the distances, to rank the patches based on their distances on passing and failing tests. The example program of Figure 1 has two exit points: the `return` statement at Line 9 and the one at Line 10.

Considering the `return` statement at Line 9 and the passing test case T-2, the system state at the exit point of the original, faulty version, is:

$$\{\texttt{str="aba?gc", c='?', k=5, j=3}\} \tag{S1}$$

The same test case for Patch 1 (the correct fix) results in:

$$\{\texttt{str="aba?gc", c='?', k=5, j=3}\} \tag{S2}$$

For Patch 2 (the incorrect fix), T-2 results in:

$$\{\texttt{str="aba?gc", c='?', k=5, j=0}\} \tag{S3}$$

```java
1  class Finder {
2    private final String str;
3    private int j;
4
5    public Finder(String s) {str = s;}
6
7    public boolean find(char c, int k) {
8      for (j = 0; j < k; j++) // bug: j <= k
9        if (str.charAt(j) == c) return true;
10     return false;
11   }
12 }
```

Figure 1: A Java program with a fault at Line 8

Table 1: Test cases for the program of Figure 1

	Input			Output		
Id	**str**	**c**	**k**	**Expected**	**Actual**	**Test Result**
T-1	“ab?”	‘?’	2	true	false	Failing
T-2	“aba?gc”	‘?’	5	true	true	Passing

Patch 1 (correct patch):

```
8  ~~for (j = 0; j < k; j++) // bug: j <= k~~
8  ++ for (j = 0; j <= k; j++)
```

Patch 2 (incorrect patch):

```
9  ~~if (str.charAt(j) == c) return true;~~
9  ++ if ('?' == c) return true;
```

Figure 2: Two plausible patches for the program of Figure 1

The system state for the correct patch (S2) is identical to that of the original program (S1), with a different `j` value for the incorrect patch (S3). Specifically, it is easy to verify $dist_{\text{ObjSim}}$ computes 0 for (S1) and (S2), while it returns 1 for (S1) and (S3). We can do similar calculations for the other exit point, as well as the failing test T-1, and verify that neither of the patches result in different $dist_{\text{ObjSim}}$ values. Therefore, ObjSim prioritizes Patch 1 over Patch 2 and since there are no ties in this case, the correct patch appears in top-1 position.

CIP does similar instrumentation, but it collects not only system state snapshots at the exit points of the patched methods, but also records a snapshot of the system state upon entry of the methods. In this way, it accelerates the patch assessment process by not re-executing test cases for each patch. Instead, it directly invokes the methods. Furthermore, unlike ObjSim, CIP only focuses on passing tests, thus it only runs test T-2.

On the faulty version of the program, upon entry to the patched method `find`, CIP records the system state snapshot

{str=“aba?gc”, c=‘?’, k=5, j=0} (S4)

Before exiting at the `return` statement at Line 9, the tool records the system state snapshot

{str=“aba?gc”, c=‘?’, k=5, j=3} (S5)

After applying either Patch 1 and Patch 2, CIP invokes `find` on a `Finder` object whose `str` field is initialized with “`aba?gc`” and whose `j` field is initialized with `0`, according to the recorded snapshot upon the entry to the method in the original program, *i.e.*, (S4). The tool also records the system state snapshots at the exit points of the method. For Patch 1 it will record

{str=“aba?gc”, c=‘?’, k=5, j=3} (S6)

while for Patch 2 it will record

{str=“aba?gc”, c=‘?’, k=5, j=0} (S7)

Since the system state snapshot for Patch 1, *i.e.*, (S6), is identical to that of the original program, *i.e.*, (S5), CIP classifies Patch 1 (with $dist_{\text{CIP}}$ value of 0) as likely correct and since there is only one likely incorrect patch (with $dist_{\text{CIP}}$ value of 3), no ranking takes place. In this case also, the correct patch appears in top-1 position.

3 EMPIRICAL STUDY

We conduct an empirical study with a two-fold goal of: (1) comparing ranking performance of two object similarity-based approaches, namely ObjSim and CIP, in terms of the number of patches that are ranked in top-1 and top-2 positions; (2) comparing the two techniques with a random ranking baseline based on hypergeometric probability [2], simulating a situation wherein no automated patch correctness assessment is present. Therefore, we define two research questions, corresponding to the study goals.

- **RQ1:** *How do ObjSim and CIP compare to each other?*
- **RQ2:** *How do the champion object similarity-based technique compare to a random baseline ranking?*

In the rest of this section, we describe the process through which we obtained our data set of patches. We describe the details about our measurements before we present the results and answer our research questions.

3.1 Dataset of Patches

For RQ2, we constructed a dataset of patches by combining the set of Defects4J bug database [10], mentioned above, the curated patch datasets, used in recent studies [34, 38, 41, 44], generated by 29 APR systems. After combining the datasets, we obtained 3,072 patches consisting of 1,684 patches labelled as *correct*, 1,374 patches labelled as *incorrect*, and 14 patches labelled as *unknown*.

Since these data sets are curated by different research groups at different times, they overlap, so we identified and excluded duplicate patches. In order to reduce manual work needed for duplicate elimination, we used a script to automatically remove patches that were identical to one another up to white-spaces. Two patches deemed to be identical if they amount to the same SHA-1 hash. We were able to remove 487 patches with the help of this hash-based script. It stands to reason that this method of duplicate elimination cannot detect semantically equivalent patches, *e.g.*, the expressions `a+b` and `(a)+b` are semantically equivalent while they have different hashcodes. Thus, a manual inspection is necessary. We manually examined the remaining patches and removed obvious duplicates that our script was unable to detect due to the unpatched code surrounding the patched lines, extra parentheses around expressions, *etc.* We further excluded 14 patches labelled as *unknown*. We also

```
StopWatch.java:
118    ~~stopTime = System.currentTimeMillis();~~
119    ~~this.runningState = STATE_STOPPED;~~
118    ++ this.splitState=STATE_SPLIT;
```

Figure 3: Incorrect patch generated by ARJA for Lang-55

excluded all the patches patches that involved creating or removal of files, classes, methods, and fields. Finally, we excluded the correct patches that were not paired with any incorrect patch, the patches that resulted in compilation errors, those that did not pass all the test cases, and those that had compatibility issues with the current implementations of ObjSim or CIP. After this pre-processing, we ended up with 1,290 patches, 197 of which are labelled as correct while 1,093 are labelled as incorrect. All these patched are generated for Defects4J bugs, and each bug have one correct patch and one or more incorrect patch(es) generated for it.

3.2 Objects and Measures

We used ObjSim and CIP to rank 1,290 patches generated for 197 Defects4J bugs. As a baseline, we used a random ranking scheme obtained by *hypergeometric probability* [2] of a correct patch appearing in top-1 or top-2.

As for ranking, we consider the *worst case ranking* for correct patches, meaning that upon ties, we pick the worst-case ranking rather than averaging or picking best rank.

We emphasize that while CIP does classification at some stage, the technique still is a ranking technique and our comparison is meaningful. We calculate the CIP rank of correct patches as follows. Given n patches, among which $m < n$ are correct, assume CIP classifies $k \leq m$ of the patches as likely correct and the rest $n - k$ as likely incorrect. We consider the batch of k likely correct patches (all of which have object distance of 0) as a tie of size k and the rest of patches are sorted as prescribed by the heuristic ranking algorithm of CIP (all of which have positive object distances). If all k likely correct patches are in fact correct, we give a worst-case rank of 1 to the correct patch. If at least one correct patch is classified as likely correct and one or more incorrect patches are also classified as likely correct, we give a worst-case rank of k to the correct patch. Otherwise, we add k to the rank of correct patch returned by the ranking algorithm of CIP.

3.3 Results and Discussion

We present and analyze the results we obtained for answering RQ1 and RQ2. Table 2 contains detailed measurements for all the bugs in our dataset.

3.3.1 RQ1: ObjSim vs. CIP. Table 3 summarizes the results. The last three columns of the table report the worst-case ranking (see §3.2) for the top ranked correct patch. The results reveal that in 19.8% (47.7%) of the cases, ObjSim ranks the correct patch in top-1 (top-2) position, meanwhile CIP ranks 9.6% (42.1%) of the correct patches in top-1 (top-2) position.

In Table 2, we have highlighted the cases where ObjSim performed worse than CIP. Here, we analyze the Lang-55 in more detail; other cases are similar. All 10 patches generated for this bug, target lines 118 and 119 of StopWatch.java which lie inside the method stop. Since these two lines are covered by the same set of failing and passing tests, both receive same Ochai suspiciousness values, so ObjSim cannot tell the difference between the patches. This is because ObjSim, as the *first* step, essentially groups and sorts the patches into equivalence classes with respect to their Ochiai suspiciousness values. However, in CIP, an incorrect patch (presented in Figure 3) get classified as correct while the rest as incorrect. The reason is that the patch deletes assignments to two fields, leaving their value unchanged, and since the third field is not touched by the enclosing method in the original program, CIP does not count the change in its value. Among the remaining 9 patches that are classified as incorrect (all of which have equal Ochiai suspiciousness values), the correct patches result in least system state change and no tie with any incorrect patch remains, thus the correct patch receives a rank of 2.

We performed Mann-Whitney U Test for comparing the rankings pair-wise, between approaches. ObjSim ranks are generally lower than CIP ranks (p=1e-5$\ll$ 0.05), which indicates that ObjSim outperforms CIP. Since these observations are of statistical significance, we conclude that ObjSim outperforms CIP in terms of the number of patches ranked in top-1 (top-2) position.

Answer to RQ1: ObjSim outperforms CIP by ranking more correct patches in top-1 and top-2 positions. ObjSim ranks 19.8% (47.7%) of patches in top-1(top-2) position, while CIP ranks 9.6% (42.1%) of the correct patches in top-1 (top-2) position.

3.3.2 RQ2: Random Baseline Comparison. Based on the discussion of §3.3.1, we learn that ObjSim is the champion technique. At this point, we want to know how do ObjSim compare to a random baseline ranking. Random baseline ranking represents a setting wherein we do not get any machine help in ranking the patches. To simulate this effect, we use *hypergeometric probability* [2] of a correct patch appearing in top-1 or top-2. Specifically, we use the following formula to calculate the probability of a correct patch appearing in top-N position.

$$h(c, s, P, N) = \frac{\begin{pmatrix} c \\ s \end{pmatrix} \begin{pmatrix} P - c \\ P - s \end{pmatrix}}{\begin{pmatrix} P \\ N \end{pmatrix}}$$

where $\begin{pmatrix} k \\ x \end{pmatrix}$ is the number of combinations of k things, taken x at a time. c is the total number of correct patches generated for each bug, s is the number of correct patches we expect to appear in top-N position, P is the total number of plausible (including correct and incorrect) patches for each bug, and N is either 1 (for top-1) or 2 (for top-2). Since in our data set, each bug has exactly one correct patch generated for it, we have $c = s = 1$.

Table 2 reports the probability values for each bugs and for top-1 and top-2. Table 3 summarizes this information by taking the average of probability values for each Defects4J projects. According to our analysis, random ranking, on average, ranks 43 (85.9) of the correct patches in top-1 (top-2) position, while ObjSim ranks 39

Table 2: Worst-case ranking by ObjSim, CIP, and random baseline ranking for the correct patch

Bug Id	#Pl.	#C	#I	ObjSim	CIP	Rand [Top-1]	Rand [Top-2]
Chart-1	24	1	23	23	24	0.04	0.08
Chart-10	2	1	1	2	2	0.50	1.00
Chart-12	13	1	12	12	9	0.08	0.15
Chart-13	36	1	35	31	36	0.03	0.06
Chart-14	5	1	4	5	5	0.20	0.40
Chart-15	15	1	14	1	10	0.07	0.13
Chart-17	3	1	2	1	3	0.33	0.67
Chart-18	4	1	3	2	4	0.25	0.50
Chart-19	3	1	2	1	2	0.33	0.67
Chart-21	4	1	3	1	4	0.25	0.50
Chart-22	2	1	1	2	2	0.50	1.00
Chart-24	2	1	1	2	2	0.50	1.00
Chart-25	27	1	26	10	11	0.04	0.07
Chart-26	13	1	12	12	12	0.08	0.15
Chart-3	9	1	8	3	1	0.11	0.22
Chart-5	16	1	15	16	13	0.06	0.13
Chart-6	5	1	4	3	1	0.20	0.40
Chart-7	10	1	9	10	7	0.10	0.20
Chart-9	8	1	7	6	8	0.13	0.25
Closure-1	4	1	3	1	4	0.25	0.50
Closure-10	6	1	5	4	6	0.17	0.33
Closure-101	4	1	3	4	4	0.25	0.50
Closure-106	2	1	1	1	2	0.50	1.00
Closure-107	4	1	3	3	4	0.25	0.50
Closure-108	4	1	3	4	3	0.25	0.50
Closure-109	3	1	2	3	3	0.33	0.67
Closure-112	5	1	4	5	5	0.20	0.40
Closure-113	2	1	1	2	2	0.50	1.00
Closure-114	6	1	5	2	5	0.17	0.33
Closure-115	4	1	3	4	4	0.25	0.50
Closure-116	2	1	1	1	2	0.50	1.00
Closure-117	3	1	2	3	3	0.33	0.67
Closure-119	5	1	4	4	5	0.20	0.40
Closure-12	5	1	4	2	5	0.20	0.40
Closure-120	3	1	2	2	2	0.33	0.67
Closure-121	3	1	2	2	2	0.33	0.67
Closure-122	3	1	2	3	1	0.33	0.67
Closure-123	2	1	1	2	2	0.50	1.00
Closure-124	6	1	5	1	6	0.17	0.33
Closure-125	4	1	3	2	4	0.25	0.50
Closure-126	9	1	8	9	9	0.11	0.22
Closure-127	4	1	3	2	4	0.25	0.50
Closure-129	5	1	4	1	5	0.20	0.40
Closure-130	4	1	3	3	4	0.25	0.50
Closure-133	6	1	5	2	4	0.17	0.33
Closure-14	2	1	1	2	2	0.50	1.00
Closure-15	3	1	2	1	2	0.33	0.67
Closure-16	2	1	1	1	2	0.50	1.00
Closure-17	4	1	3	1	4	0.25	0.50
Closure-18	6	1	5	5	6	0.17	0.33
Closure-19	2	1	1	2	2	0.50	1.00
Closure-2	4	1	3	3	4	0.25	0.50
Closure-21	15	1	14	15	15	0.07	0.13
Closure-22	13	1	12	13	13	0.08	0.15
Closure-26	3	1	2	1	3	0.33	0.67
Closure-31	2	1	1	2	2	0.50	1.00
Closure-33	7	1	6	6	7	0.14	0.29
Closure-35	3	1	2	3	3	0.33	0.67
Closure-38	5	1	4	5	5	0.20	0.40
Closure-45	3	1	2	3	2	0.33	0.67
Closure-48	4	1	3	4	4	0.25	0.50
Closure-49	2	1	1	1	2	0.50	1.00
Closure-50	2	1	1	1	2	0.50	1.00
Closure-55	7	1	6	5	7	0.14	0.29
Closure-57	2	1	1	2	2	0.50	1.00
Closure-59	6	1	5	4	6	0.17	0.33
Closure-60	2	1	1	2	2	0.50	1.00
Closure-62	6	1	5	6	6	0.17	0.33
Closure-64	3	1	2	2	2	0.33	0.67
Closure-66	3	1	2	2	3	0.33	0.67
Closure-67	5	1	4	5	5	0.20	0.40
Closure-68	3	1	2	1	3	0.33	0.67
Closure-7	3	1	2	1	3	0.33	0.67
Closure-73	2	1	1	2	2	0.50	1.00
Closure-75	6	1	5	2	3	0.17	0.33
Closure-76	2	1	1	1	2	0.50	1.00
Closure-78	5	1	4	5	4	0.20	0.40
Closure-79	2	1	1	1	2	0.50	1.00
Closure-8	2	1	1	2	2	0.50	1.00
Closure-86	5	1	4	5	5	0.20	0.40
Closure-89	2	1	1	1	2	0.50	1.00
Closure-90	2	1	1	1	2	0.50	1.00
Closure-92	11	1	10	11	11	0.09	0.18
Lang-1	3	1	2	2	3	0.33	0.67
Lang-10	4	1	3	4	3	0.25	0.50
Lang-12	2	1	1	2	2	0.50	1.00
Lang-13	2	1	1	1	2	0.50	1.00
Lang-14	2	1	1	2	2	0.50	1.00
Lang-15	2	1	1	2	2	0.50	1.00
Lang-16	6	1	5	6	2	0.17	0.33
Lang-18	2	1	1	2	2	0.50	1.00
Lang-20	4	1	3	3	4	0.25	0.50
Lang-21	2	1	1	2	2	0.50	1.00
Lang-22	10	1	9	10	10	0.10	0.20
Lang-24	2	1	1	2	2	0.50	1.00
Lang-27	11	1	10	11	9	0.09	0.18
Lang-29	2	1	1	1	2	0.50	1.00
Lang-31	2	1	1	2	2	0.50	1.00
Lang-35	2	1	1	2	2	0.50	1.00
Lang-39	11	1	10	11	11	0.09	0.18
Lang-40	2	1	1	2	1	0.50	1.00
Lang-41	5	1	4	3	5	0.20	0.40
Lang-43	8	1	7	7	7	0.13	0.25
Lang-44	9	1	8	9	8	0.11	0.22
Lang-45	7	1	6	7	7	0.14	0.29
Lang-46	3	1	2	2	2	0.33	0.67
Lang-50	9	1	8	5	9	0.11	0.22
Lang-51	13	1	12	13	13	0.08	0.15
Lang-53	5	1	4	5	5	0.20	0.40
Lang-55	10	1	9	10	2	0.10	0.20
Lang-57	2	1	1	2	2	0.50	1.00
Lang-58	9	1	8	9	5	0.11	0.22
Lang-59	28	1	27	28	21	0.04	0.07
Lang-6	3	1	2	3	2	0.33	0.67
Lang-60	5	1	4	2	1	0.20	0.40
Lang-61	8	1	7	7	4	0.13	0.25
Lang-63	13	1	12	12	13	0.08	0.15
Lang-7	12	1	11	11	12	0.08	0.17
Math-1	2	1	1	2	1	0.50	1.00
Math-101	3	1	2	3	2	0.33	0.67
Math-103	2	1	1	1	2	0.50	1.00
Math-104	5	1	4	5	2	0.20	0.40
Math-105	5	1	4	5	3	0.20	0.40
Math-11	2	1	1	2	2	0.50	1.00
Math-15	5	1	4	5	5	0.20	0.40
Math-16	2	1	1	2	1	0.50	1.00
Math-18	3	1	2	1	3	0.33	0.67
Math-2	35	1	34	35	35	0.03	0.06
Math-20	19	1	18	3	11	0.05	0.11
Math-22	2	1	1	2	2	0.50	1.00
Math-24	3	1	2	3	2	0.33	0.67
Math-28	34	1	33	34	1	0.03	0.06
Math-29	4	1	3	1	1	0.25	0.50
Math-3	3	1	2	3	3	0.33	0.67
Math-30	2	1	1	2	2	0.50	1.00
Math-31	9	1	8	7	9	0.11	0.22
Math-32	7	1	6	6	7	0.14	0.29
Math-33	6	1	5	6	5	0.17	0.33
Math-39	2	1	1	1	2	0.50	1.00
Math-4	4	1	3	2	4	0.25	0.50
Math-40	10	1	9	10	8	0.10	0.20
Math-41	4	1	3	2	2	0.25	0.50
Math-42	6	1	5	1	2	0.17	0.33
Math-43	3	1	2	1	3	0.33	0.67
Math-44	4	1	3	3	2	0.25	0.50
Math-49	19	1	18	9	7	0.05	0.11
Math-5	8	1	7	1	8	0.13	0.25
Math-50	14	1	13	14	13	0.07	0.14
Math-52	2	1	1	2	2	0.50	1.00
Math-53	6	1	5	6	3	0.17	0.33
Math-56	5	1	4	5	5	0.20	0.40
Math-57	8	1	7	5	8	0.13	0.25
Math-58	7	1	6	2	7	0.14	0.29
Math-59	2	1	1	2	2	0.50	1.00
Math-6	5	1	4	5	5	0.20	0.40
Math-60	3	1	2	1	3	0.33	0.67
Math-61	2	1	1	2	2	0.50	1.00
Math-62	5	1	4	4	4	0.20	0.40
Math-63	17	1	16	16	16	0.06	0.12
Math-64	2	1	1	1	2	0.50	1.00
Math-68	2	1	1	2	1	0.50	1.00
Math-69	4	1	3	1	4	0.25	0.50
Math-7	6	1	5	4	4	0.17	0.33
Math-70	5	1	4	2	5	0.20	0.40
Math-71	7	1	6	4	5	0.14	0.29
Math-72	2	1	1	1	2	0.50	1.00
Math-73	12	1	11	10	12	0.08	0.17
Math-74	4	1	3	4	4	0.25	0.50
Math-77	2	1	1	2	2	0.50	1.00
Math-78	8	1	7	3	6	0.13	0.25
Math-79	3	1	2	3	3	0.33	0.67
Math-8	13	1	12	13	13	0.08	0.15
Math-80	49	1	48	47	47	0.02	0.04
Math-81	24	1	23	24	22	0.04	0.08
Math-82	22	1	21	10	4	0.05	0.09
Math-84	10	1	9	10	1	0.10	0.20
Math-85	35	1	34	35	35	0.03	0.06
Math-87	4	1	3	4	2	0.25	0.50
Math-88	8	1	7	7	6	0.13	0.25
Math-93	3	1	2	2	3	0.33	0.67
Math-94	3	1	2	3	3	0.33	0.67
Math-95	16	1	15	14	2	0.06	0.13
Math-96	2	1	1	2	1	0.50	1.00
Math-97	4	1	3	3	4	0.25	0.50
Math-99	2	1	1	1	1	0.50	1.00
Mockito-38	2	1	1	1	1	0.50	1.00
Time-11	18	1	17	18	12	0.06	0.11
Time-12	2	1	1	1	1	0.50	1.00
Time-14	5	1	4	3	2	0.20	0.40
Time-17	4	1	3	4	1	0.25	0.50
Time-18	2	1	1	1	2	0.50	1.00
Time-19	2	1	1	2	1	0.50	1.00
Time-20	2	1	1	2	2	0.50	1.00
Time-24	3	1	2	2	1	0.33	0.67
Time-4	12	1	11	12	12	0.08	0.17
Time-7	4	1	3	3	3	0.25	0.50
Time-9	3	1	2	1	1	0.33	0.67
Total	1,093	197	1,290				

* **#Pl.**, **#C**, and **#I** denote number of plausible, correct, and incorrect patches, resp. **Rand [Top-1]** and **Rand [Top-2]** represent probability of a correct patch appearing in top-1 and top-2 position, resp.

Table 3: Top-1 and Top-2 ranking performance of the three techniques, summarizing Table 2

Project	#Pl	#C	#I		ObjSim	CIP	Rand [Avg]
Chart	201	19	182	**Top-1**	4	2	2.4
				Top-2	8	6	4.8
Closure	269	64	205	**Top-1**	17	1	16
				Top-2	36	24	32
Lang	220	35	185	**Top-1**	2	2	7
				Top-2	15	17	14
Math	541	67	474	**Top-1**	12	8	13.4
				Top-2	28	27	26.8
Mockito	2	1	1	**Top-1**	1	1	0.5
				Top-2	1	1	1
Time	57	11	46	**Top-1**	3	5	3.7
				Top-2	6	8	7.3
Total	1290	197	1093	**Top-1**	39	19	43
				Top-2	94	83	85.9

(94) of the correct patches in top-1 (top-2) positions. Therefore, we conclude that ObjSim (and hence CIP) does not offer any improvement over random ranking in terms of the number of correct patches ranked in top-1 position. This is while ObjSim still ranks more correct patches in top-2 position.

Answer to RQ2: ObjSim ranks more correct patches in top-2 position than random baseline, but in terms of the number of correct patches ranked in top-1 position, ObjSim does not offer any improvement over the random ranking scheme.

3.4 Threats to Validity

Threats to Internal Validity. There is an accuracy issue that plagues both CIP and ObjSim but other dynamic techniques (most notably PATCH-SIM [38]) are likely immune to. Note that there is an implicit assumption behind ObjSim and CIP that the computation is deterministic. As a simple example that violates this assumption, consider two (identical) copies of a program that stores the value of a call to `System.currentTimeMillis()` in a field of an object. PATCH-SIM can determine that the two copies are identical, as the sequence of instructions executed by the two copies are equal. However, ObjSim and CIP depend on the value of the field in which the returned value of `System.currentTimeMillis()` is stored, and in two different executions they will observe different values for the field. In other words, given a set of plausible patches for a nondeterministic program, ObjSim and CIP are likely to rank the patches differently for different executions. This fact might pose some threat to the validity of our conclusions. To reduce this threat, we ran our experiments two times ensuring that the computed distance values do not fluctuate.

Threats to External Validity. We have drawn our conclusions based on a set of bugs from a specific data set (namely Defects4J), and CIP and ObjSim might perform differently on different set of bugs. The tools are publicly available [13, 44], so the research community can repeat these experiments on larger scale.

Threats to Construct Validity. In our study we focus only on existing object similarity-based approaches, which interpret program behavior changes differently from other dynamic approaches. Different abstractions of the program behavior may lead to other conclusions.

4 RELATED WORK

Patch prioritization based on static code features and/or fault localization information had been an integral part of a number of APR systems [7, 8, 15, 30, 35]. The research community also focused on patch correctness assessment as an standalone research [5, 6, 12, 14, 32, 34, 37, 38, 40–44]. Here we briefly discuss the empirical studies on patch assessment, which are most related to our work.

Ye *et al.* [42] present an empirical study on the correctness of APR-generated patches assessed *via* DiffTGen tool [37]. A recent study by Wang *et al.* [34] reveals that static and dynamic automatic patch correctness assessment techniques are complementary in that the former achieve high recall while the latter tend to achieve higher precision. None of these studies concern the techniques based on object similarity.

5 CONCLUSIONS

We studied a family of dynamic patch ranking techniques that we call object similarity-based patch ranking techniques. Specifically, we ran ObjSim [14] and CIP [44] on 1,290 APR-generated patches. Our results provide empirical evidence that ObjSim outperforms CIP in terms of the number of correct patches ranked in top-1 (top-2) position, yet it does not offer any improvement over a random baseline ranking. These results warrant further research on the assumptions underlying these two techniques and techniques based on similar assumptions.

REFERENCES

[1] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE*. 306–317.

[2] Matthew A Carlton and Jay L Devore. 2017. *Probability with applications in engineering, science, and technology.* Springer.

[3] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding automatically-generated patches through symbolic invariant differences. In *ASE*. 411–414.

[4] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-fly Patch Validation for All. In *ICSE*. 1123–1134.

[5] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *IBF*. 18–25.

[6] Xuan-Bach D. Le, Lingfeng Bao, David Lo, Xin Xia, Shanping Li, and Corina Pasareanu. 2019. On reliability of patch correctness assessment. In *ICSE*. 524–535.

[7] Xuan-Bach D. Le, Duc-Hiep Chu, David Lo, Claire Le Goues, and Willem Visser. 2017. S3: syntax-and semantic-guided repair synthesis via programming by examples. In *FSE*. 593–604.

[8] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven automated program repair. In *SANER*. 213–224.

[9] Vidroha Debroy and W. Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST*. 65–74.

[10] Defects4J Contributors. 2020. http://bit.ly/2PY3yDa. Accessed: 01/22.

[11] Thomas Durieux, Fernanda Madeiral, Matias Martinez, and Rui Abreu. 2019. Empirical review of Java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts. In *FSE*. 302–313.

[12] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding Program Repair. In *ISSTA*. 8–18.

[13] Ali Ghanbari. 20022. Revisiting Patch Ranking in Automated Program Repair using Object Similarity: An Extensive Study. https://bit.ly/35Y5QOg Accessed: 01/22.

[14] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *ISSTA*. 541–544.

[15] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *ISSTA*. 19–30.

[16] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI*. 1345–1351.

[17] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. In *PASTE*. 83–90.

[18] Jiajun Jiang, Yingfei Xiong, Hongyu Zhang, Qing Gao, and Xiangqun Chen. 2018. Shaping program repair space with existing patches and similar code. In *ISSTA*. 298–309.

[19] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *TSE* 38 (2012), 54–72.

[20] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* 62 (2019), 56–65.

[21] Jingjing Liang, Jiru Yi, Jiajun Jiang, YIling Lou, Yingfei Xiong, and Gang Huang. 2020. Interactive Patch Filtering as Debugging Aid. *arXiv* (2020).

[22] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*. 166–178.

[23] Z. Manna. 2003. *Mathematical Theory of Computation*. Dover Publications.

[24] Alexandru Marginean, Johannes Bader, Satish Chandra, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, and Andrew Scott. 2019. Sapfix: Automated end-to-end repair at scale. In *ICSE-SEIP*. 269–278.

[25] Matias Martinez and Martin Monperrus. 2016. Astor: A Program Repair Library for Java. In *ISSTA*. 441–444.

[26] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *FSE*. 925–936.

[27] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.

[28] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator Patches Programs Automatically. *Ubiquity* (2019), 1–12.

[29] Yuhua Qi, Xiaoguang Mao, Yan Lei, Ziying Dai, and Chengsong Wang. 2014. The strength of random search on automated program repair. In *ICSE*. 254–265.

[30] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE*. 648–659.

[31] Edward K. Smith, Earl T. Barr, Claire Le Goues, and Yuriy Brun. 2015. Is the cure worse than the disease? overfitting in automated program repair. In *FSE*. 532–543.

[32] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.

[33] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. 151–162.

[34] Shangwen Wang, Ming Wen, Bo Lin, Hongjun Wu, Yihao Qin, Deqing Zou, Xiaoguang Mao, and Hai Jin. 2020. Automated Patch Correctness Assessment: How Far are We?. In *ASE*. 968–980.

[35] Ming Wen, Junjie Chen, Rongxin Wu, Dan Hao, and Shing-Chi Cheung. 2018. Context-aware patch generation for better automated program repair. In *ICSE*. 1–11.

[36] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* 42 (2016), 707–740.

[37] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation.. In *ISSTA*. 226–236.

[38] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. 789–799.

[39] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *IBF*. 1–8.

[40] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *FSE*. 831–841.

[41] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus. 2021. Automated Classification of Overfitting Patches with Statically Extracted Code Features. *TSE* (2021), 1–1.

[42] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *arXiv* (2019).

[43] Zhongxing Yu, Matias Martinez, Benjamin Danglot, Thomas Durieux, and Martin Monperrus. 2019. Alleviating patch overfitting with automatic test generation: a study of feasibility and effectiveness for the Nopol repair system. *ESE* 24 (2019), 33–67.

[44] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *TOSEM* 29 (2020), 1–53.