Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code

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

Andrian Marcus
amarcus@utdallas.edu
University of Texas at Dallas
Richardson, Texas, USA

ABSTRACT

Test-based generate-and-validate automated program repair (APR) systems often generate many patches that pass the test suite without fixing the bug. The generated patches must be manually inspected by the developers, so previous research proposed various techniques for automatic correctness assessment of APR-generated patches. Among them, dynamic patch correctness assessment techniques 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, *e.g.*, removing the code implementing correct functionality of the program. In this paper, we propose and evaluate a novel technique, named Shibboleth, for automatic correctness assessment of the patches generated by test-based generate-and-validate APR systems. Unlike existing works, the impact of the patches is captured along three complementary facets, allowing more effective patch correctness assessment. Specifically, we measure the impact of patches on both production code (*via* syntactic and semantic similarity) and test code (*via* code coverage of passing tests) to separate the patches that result in similar programs and that do not delete desired program elements. Shibboleth assesses the correctness of patches *via* both ranking and classification. We evaluated Shibboleth on 1,871 patches, generated by 29 Java-based APR systems for Defects4J programs. The technique outperforms state-of-the-art ranking and classification techniques. Specifically, in our ranking data set, in 43% (66%) of the cases, Shibboleth ranks the correct patch in top-1 (top-2) positions, and in classification mode applied on our classification data set, it achieves an accuracy and F1-score of 0.887 and 0.852, respectively.

CCS CONCEPTS

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

KEYWORDS

Automated Program Repair, Patch Correctness Assessment, Similarity, Branch Coverage

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ISSTA '22, July 18–22, 2022, Virtual, South Korea
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9379-9/22/07.
https://doi.org/10.1145/3533767.3534368

ACM Reference Format:
Ali Ghanbari and Andrian Marcus. 2022. Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '22), July 18–22, 2022, Virtual, South Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3533767.3534368

1 INTRODUCTION

For more than a decade, automated program repair [49] (APR) has been the subject of intense research with many APR techniques being proposed, and due to its potential in reducing software maintenance costs, APR remains an active area of research [65].

In the context of test-based generate-and-validate (G&V) APR techniques, a patch passing all the test cases is referred to as *plausible*. A plausible patch is called *correct* (or *genuine*) if it actually fixes the bug, *i.e.*, it complies with the program specification. Since test suites only partially specify the desired behavior of the system, APR tools generate many patches that merely pass the available tests without fixing the bug [49, 78] and such patches are called *test case overfitted* or *incorrect* patches. Manually searching for the correct patch is a time-consuming activity.

In order to alleviate such manual effort, several techniques for assessing the quality of generated patches are developed (see §6). Dynamic patch correctness assessment approaches operate by comparing the run-time behavior of the patched program with its unpatched version. The approaches differ from one another primarily in the way they capture and quantify the difference in program behavior. For example, PATCH-SIM [91] uses path spectra [35] as an abstraction of the program behavior and utilizes Longest Common Subsequence algorithm to quantify the differences. Another family of approaches [6, 95] use dynamically inferred invariants as an abstraction of the program behavior and rely on syntactic distance metrics to quantify the differences. In contrast, ObjSim [29] and CIP [100] quantify the behavior change by comparing system state snapshots at the exit points of patched methods. All dynamic patch correctness assessment approaches have one thing in common, in that they all are based on the assumption that correct patches do not alter the program behavior significantly, when running the originally passing test cases [6, 24, 29, 91, 95].

Depending on static or dynamic nature of the technique, patch assessment can be carried out before or after the patch validation phase in a test-based G&V APR system (see §2). Patch correctness assessment could be in the form of *ranking* or *classification*. Ranking after patch validation phase makes sense only for APR systems that do not stop searching after finding the first plausible patch, while classification can be applied in either case at the cost of possibly leaving the user with the obligation of manually analyzing the

patches within each class. We argue that there is a need for both kinds of assessment methods and the quest for fixing more complex bugs (*e.g.*, by employing an ensemble of existing repair tools, as in Repairnator [66]) and finding more correct patches within the search space of existing repair tools (*e.g.*, by forcing the existing tools to continue searching their repair search space and validating the generated patches faster, as in UniAPR [8]) necessitates not dismissing automated ranking of plausible patches as a topic without practical importance.

This paper proposes a novel treatment of automated patch correctness assessment in terms of both ranking and classification. Similar to existing dynamic approaches [6, 24, 29, 91, 95], the idea underlying our method is inspired by the *competent programmer hypothesis* [20] and previous research that determined that code-removing program transformations are *anti-patterns* in the context of APR [79]. Specifically, we posit that the programmer writes her program almost correctly insofar as any bug fixing activity involves small changes to the program text and does not remove code implementing existing desired functionality. As such, we expect a correct patch to impact production code less than an incorrect patch and not to decrease code coverage of originally passing test cases by removing tested program elements that are likely to already comply with the program specification. However, unlike the existing techniques, we captured the impact of patches along three complementary facets: syntactic (in terms of textual similarity), semantic (in terms of similarity of execution traces), and the code coverage of the originally passing test cases. With this basic idea in mind, we design and implement Shibboleth, which assesses the quality of APR-generated patches *via* both ranking and classification. In ranking mode, given a collection of patches, the tool groups and sorts the patches in such a way that groups of patches that are more likely to be correct appear before the ones that are less likely to be correct in the fix report. In classification mode, the patches are categorized into two classes of likely correct and likely incorrect.

We construct a large set of 1,871 APR-generated and human-written patches, based on data used by previous research and use this data for the empirical evaluation of Shibboleth. We compare the ranking capability of Shibboleth with state-of-the-art patch ranking systems CIP [100] (a patch ranking/classification sub-system part of ARJA-E [100]) and ObjSim [29], an Ochiai-based [87] ranking approach, and a random baseline. The results show that, in our ranking data set, Shibboleth ranks a correct patch in top-1 or top-2 position for 66% of the 197 bugs we studied, outperforming CIP, ObjSim, Ochiai-based ranking, and the random baseline. We also evaluate Shibboleth in classification mode by conducting cross validation of its underlying learning algorithm on our classification data set. The classification algorithm achieves an accuracy and F1-score of 0.887 and 0.852, respectively, and outperforms state-of-the-art PATCH-SIM [91], CIP [100], ODS [97], and the static patch classification system by Tian *et al.* [80].

This research work is significant from both theoretical and practical points of view. Patch overfitting is a challenging problem in APR, hindering widespread adoption of APR systems in industrial settings [49]. We propose an effective technique to alleviate this problem. Our technique, compared to current techniques, relies on light-weight measures (*i.e.*, taking constant time *vs.* quadratic time) making it suitable for practical use-cases. Our large patch data set is a benchmark, with detailed information about the patches, that supports advancing software engineering research.

Contributions. This paper makes the following contributions.

(1) A novel technique, and a tool named Shibboleth, for patch correctness assessment (*via* ranking and classification) leveraging the impact of the patches on both production code and test suite coverage. Empirical evaluation shows that Shibboleth outperforms state-of-the-art static and dynamic patch ranking and classification techniques.
(2) A curated and annotated data set of 1,871 APR-generated and human-written patches.

Our patch data set and software artifacts are publicly available [32].

Paper Organization. The rest of this paper is organized as follows. A brief overview of APR and a motivating example are given §2. In §3 we describe our approach in detail. We present the results of our empirical evaluation in §4. In §5, we mention the trade-offs made in the design of Shibboleth and discuss its limitations and threats to validity. We discuss related work in §6, and §7 concludes.

2 BACKGROUND AND MOTIVATION

APR aims at reducing debugging costs by generating high-quality patches that either directly fix the bugs or help developers during the course of manual debugging. Depending on the actions taken to conduct a repair, APR techniques can be categorized into different classes [49, 65]. The majority of current APR techniques belong to the search-based class (aka generate-and-validate, or G&V for short) that attempts to fix the bugs through evolutionary search [2, 48], heuristic fix templates [17, 31, 53], code grafting [4, 60, 85], or random mutation [18, 71] and validating the generated patches using certain checks. The majority of G&V techniques are test-based [3, 9, 10, 17, 18, 31, 38, 41, 43, 48, 53, 56, 57, 59, 73–75, 85, 93, 94, 100], meaning that they validate the generated patches by running test suite against the patches, while others are purely static [27, 34, 63, 82], targeting compilation errors, code smells, or specific classes of errors, *e.g.*, resource leaks or data races.

Most test-based G&V APR systems are based the following steps. First, a fault localization algorithm [87] is applied to the buggy program to find suspicious program elements. Next, the most suspicious program elements are transformed based on a variety of methods (*e.g.*, random mutation [20], code grafting [4], program synthesis [50], *etc.*) to generate a pool of program variants. The available test suite is executed against the program variants and the ones that pass all the test cases are kept, during the next step. A fix report containing details about plausible patches is then generated and presented to the users for further manual inspection.

2.1 Motivating Example

Figure 1 presents a faulty Java program that, given a string `s` and a character `c`, is intended to return `true` if `c` occurs within `s` and `false`, otherwise. The figure also lists a test suite with two test cases that exercise the buggy method `contains` with different inputs, one of which reveals the fault, *i.e.*, the missing null-check. T-1 (colored in green) is a passing test, while T-2 (colored rose) is a failing test case that results in a `NullPointerException` (NPE).

```
1 class Example {
2   public static boolean contains(String s, char c) {
3     int k = s.length(); // bug: NPE when s == null
4     for (int j = 0; j < k; j++) {
5       if (s.charAt(j) == c) {
6         return true;
7       }
8     }
9     return false;
10  }
11 }
```

	Input		Output	
Id	s	c	Expected	Actual
T-1	"abc"	'?'	false	false
T-2	null	'?'	false	NPE

Figure 1: A Java program with a missing null-check. A test suite with two test cases T-1 (passing) and T-2 (failing) is listed to the right of the code.

Patch 1 (correct patch):

```
3  + if (s == null) {
4  + return false;
5  + }
```

Patch 2 (incorrect patch):

```
3  + if (c > 0) {
4  + return false;
5  + }
```

Figure 2: Two plausible patches for the program of Figure 1. Each patch introduces three new lines of code.

Figure 2 shows two possible patches that a typical G&V APR tool (*e.g.*, [60]) may produce in an attempt to fix the example code. While both patches result in a code that passes both T-1 and T-2 (*i.e.*, they are plausible), only one of them is correct (*i.e.*, Patch 1).

By parsing the body of the original version of the method `contains` into tokens and counting the tokens, we get two instances of `k`, two instances of `s`, and so on. The correctly patched version of `contains` has two instances of `k`, three instances of `s`, and so on, which yields a cosine similarity of approximately 0.970725 to the original program. The incorrectly patched version of `contains` has two instances of `k`, two instances of `s`, and so on, which results in a cosine similarity of approximately 0.967994 to the original program. In this particular example, the correct patch was actually more similar to the original program than the incorrect patch, suggesting that the correct patch impacts the program text less significantly than the incorrect one. But there are situations where we might get equal similarity or the opposite of what we expect. This is why we need a complementary dynamic metric to measure the impact of patches on production code.

Looking at the spectrum of JVM bytecode [51] instructions exercised while executing the passing test T-1, we notice that, in the original program, we get 2 instances of load constant, 6 instances of load local variable, and so on. In the correctly patched version, we get 3 instances of load constant, 7 instances of load local variable, and so on. This results in a cosine similarity of approximately 0.982343. Meanwhile, in the incorrectly patched version, we get two instances of load constant and one instances of each of load local variable, comparison, and return instructions. In this version, the rest of the instructions do not appear, indicating that this version quite dissimilar to the original program with a cosine similarity of approximately 0.725901. As we can see, in this example, the correct patch has a more similar execution to the original program than the incorrect patch.

The branch coverage for the originally passing test case T-1 before patching is $\frac{3}{4} = 0.75$, as both true and false branches of the `for`-loop in line 4 are covered and only the false branch of the `if`-condition in line 5 is covered. The branch coverage for the test suite after patching with Patch 1 (the correct patch) increases to $\frac{5}{6} \approx 0.833$, *i.e.*, a delta of 0.083, while it decreases to $\frac{1}{6} \approx 0.167$ after patching with Patch 2 (the incorrect patch), *i.e.*, a delta of -0.583. This suggests that the correct patch does not involve removing functionality that is tested (*i.e.*, desirable), while the incorrect patch passes the test cases by leaving portions of the code untested.

We suggest using these features to rank and/or discriminate APR-generated patches. This idea is implemented in the technique and tool, named Shibboleth, discussed in the rest of the paper.

3 APPROACH

Similar to previous research on patch correctness assessment [6, 24, 29, 91, 95], the main idea behind Shibboleth is inspired by the *competent programmer hypothesis* [20], for which we have empirical evidence due Andrews *et al.* [1], and code-removing program transformations being anti-patterns [79]. Specifically, we rely on the programmer to write her program almost correct, insofar as any bug fixing activity involves *small changes* to the program text and *does not remove its desired elements*. Desired program elements (*e.g.*, lines of code, methods, *etc.*) are the ones whose correctness is verified or we have some level of confidence in their correctness. These could be the program elements covered only by the passing test cases, as they specify desired behavior of the program [87].

However, unlike the existing work, we quantify the size of the changes due to applying a patch *via* syntactic and semantic similarity of a patched program to its original, buggy version. Greater similarity values could mean that the patched code is more likely to behave like the original program, and large code coverage difference values could mean that the patch does not delete desired functionality of the program. Having concrete values representing the size of changes induced by a set of patches, Shibboleth prioritizes the patches that result in large similarity values for production code and greater difference values (*i.e.*, more positive) for test code.

3.1 Measures

We quantify the *amount of deleted desired program elements* by calculating the difference of the code coverage of the originally passing test cases before and after patching. Among common code coverage metrics, we choose branch coverage as it subsumes statement coverage, yet it is equally efficient to calculate. **BC** denotes the difference in branch coverage of the passing test cases before and after patching and we use it as a feature to rank and discriminate the patches. To calculate the *syntactic similarity*, Shibboleth computes token-level syntactic similarity of the patched version of the program with respect to its original version. The basic idea behind token-level textual similarity is that textually similar programs tend to be semantically close. This idea forms the basis of a wide range of techniques for clone detection and code recommendation [42, 58], APR [43, 73], speeding up mutation analysis [101] and static program analysis [81]. To calculate token-level textual similarity, Shibboleth parses the original and patched programs

into tokens and builds vectors of the frequencies of tokens for all the patched methods. We then calculate cosine similarity of the resulting vectors, denoted **TS**, and use it as a feature for ranking, as well as discriminating patches. We dynamically analyze the behavior of the program before and after patching to directly measure the *behavioral similarity* of the patched program to its original version. To this end, we calculate the cosine similarity of what we call Statement-Count Spectra, denoted **SCS**, for the program before and after patching. We emphasize that token-level syntactic similarity and behavioral similarity capture the change to the original program from two different perspectives and are complementary to each other. Combining the two boosts Shibboleth's precision (see §4 for more details).

We now explain the rationale behind using **SCS** as a representative for program behavior. The operations of a computational system, *e.g.*, a program P, whether it is sequential or concurrent, can be modeled as a *transition system* defined as

$$\langle Q, \rightarrow, q_0 \rangle \tag{1}$$

where Q is a set of *states*, $\rightarrow \subseteq Q \times Q$ is a *transition relation* defined over the states, and $q_0 \in Q$ is the *initial state*. Neither Q nor $\rightarrow$ are restricted to be finite. Given states q_1 and q_2, we think of $q_1 \rightarrow q_2$ as an *indivisible* transition from state q_1 to state q_2. This model is general enough to capture most notions of sequential and parallel computation. For sequential programs, given a state q_1, there is at most one state q_2 with $q_1 \rightarrow q_2$, while for concurrent programs, there might be more than one successor for a given state. The variable-value bindings in the initial state q_0 can be seen as "inputs" to the program.

An execution E of the program P is a (possibly infinite) sequence of states, starting with the initial state q_0:

$$E = q_0, q_1, q_2, \ldots$$

where for each $i \geq 0$, $q_i \rightarrow q_{i+1}$. For each input (*i.e.*, initial state) and each interleaving of the states (in the case of concurrent programs), we can get a different execution of the program. The set of all possible executions of a program P is often referred to as the *behavior* of P [47], for it captures all possible states and state transitions (hence the dynamic behavior) of the program.

Each state $q \in Q$ in Eq. 1 is a set of variable-value bindings which can be characterized by a formula in propositional logic, hence, an execution E of the program P can alternatively be modeled as a chain of Hoare triples as follows.

$$E = \{\Phi_0\} S_1 \{\Phi_1\} S_2 \{\Phi_2\} S_3 \{\Phi_3\} \ldots,$$

where S_1, S_2, *etc.*, are the statements in P. The formula Φ_0 specifies the initial state q_0 and for each $i \geq 0$, the formula Φ_i is the weakest precondition of S_{i+1} and Φ_{i+1} is a post-condition of S_{i+1}. So, for a given initial state characterized by Φ_0, an execution E of the program P can be represented more succinctly as

$$E = S_1, S_2, S_3, \ldots \tag{2}$$

where S_1, S_2, *etc.*, are the statements in P. All the intermediate formulae Φ_1, Φ_2, *etc.*, can be inferred based on Φ_0 and the statements.

E is a sequence of executed program statements, *i.e.*, an *execution trace*. Similarly, the behavior of the program P can be defined as the set of all execution traces of P paired with the formulae characterizing the initial state for each trace. This characterization of program behavior forms the basis of a long-standing research in software testing [35, 72] and later in automated program repair [24, 83, 91, 100]. Based on the specific application domain, or simply due to efficiency reasons, an execution trace may be represented in different ways during profiling [35]. These representations of an execution trace, and the execution trace itself, are generally referred to as *program spectra* [35]. Different program spectra approximate the program behavior at certain level of abstractions or, as Harrold *et al.* state, a program spectrum *"characterizes, or provides a signature of, a program's behavior"* [35]. Thus, it makes perfect sense to use program spectra as "representatives" of program behavior and use them to heuristically reason about correctness or the impact of various patches on a buggy program.

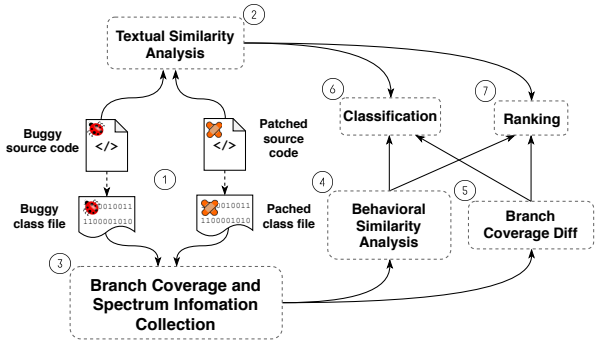

Figure 3: Architecture of Shibboleth

In this paper, we propose a Branch-Count Spectrum (BCS) [35] for statements, hence the name Statement-Count Spectrum. The idea is to count the number of times each statement is exercised during testing and store it in a vector of fixed dimension. This representation of execution traces, although loses the temporal relation between the executed statements, still contains information about the frequency of execution of each statement and is more informative than the traditional statement coverage. As we will see in §4, this enables us to accurately classify the patches and achieve outstanding results compared to state-of-the-art.

3.2 Shibboleth Architecture

The steps taken by Shibboleth to calculate **TS**, **SCS**, and **BC** are sketched in Figure 3. The buggy and patched versions of the program are compiled (step ①) to obtain class files. The source files are parsed and analyzed (in step ②) to obtain **TS**. The binary class files are instrumented so as to gather information about the frequency of each covered instruction, as well as branch coverage (③). The collected coverage and spectrum information are used to calculate **SCS** (in step ④) and **BC** (in step ⑤). Finally, the three calculated values are used as features for classification (in step ⑥) or ranking (in step ⑦).

In the rest of this section, we present the ranking and classification algorithms of Shibboleth. But before that, we introduce a number of notations in the section that follows so as to simplify the presentation of the algorithms.

3.3 Notation

Let P be a program (*e.g.*, a Java project) and $T_{\checkmark}$ be the passing subset of the test suite accompanying program P (*e.g.*, a set of passing JUnit test methods). We use Π to represent a finite set of patches and $\pi(P)$, where $\pi \in \Pi$, to denote the variant of the program P obtained from applying the patch π to it. Operationally speaking, a patch is a diff file and it can be applied using Git or the UNIX `patch` command. Given a patch π from our data set, $\ell(\pi)$ denotes the ground-truth label for the patch, with $\ell(\pi) \in \{\texttt{CORRECT}, \texttt{INCORRECT}\}$.

Common to both ranking and the classification subsystems is a component that calculates syntactic and semantic similarity (*i.e.*, **TS** and **SCS**), as well as branch coverage (*i.e.*, **BC**) of the passing test cases $T_{\checkmark}$ with respect to the original program P and $\pi(P)$ for a given patch π. The **BC** measure, as introduced in the previous subsection, is the difference in branch coverage score before and after patching. In this way, once we fix a program P and a set of passing test cases $T_{\checkmark}$, we can think of **BC** as a function of the patches, that is to say, we use $\mathbf{BC}(\pi)$ to denote the difference in branch coverage of the passing test cases executed against P and $\pi(P)$. Similarly, we use **TS** and **SCS** as functions of patches.

3.4 Ranking

The input to the ranking algorithm is a set Π of patches and the functions **BC**, **TS**, and **SCS**, as defined in the previous section. The output of the algorithm is a sequence of patches paired with their ranks, which is sorted in ascending order of the ranks.

Algorithm 1 lists the steps needed for ranking the patches based on the three observations that we made about the impact of the correct patches on original programs. The algorithm invokes the recursive function `groupAndSort`, which, given a set of patches and a list of comparators $c_1, \ldots, c_n$, groups the patches based on c_1 and sorts the groups also based on c_1. If any tie (a group with two or more patches) is remaining, the grouping and sorting is repeated recursively for each group based on c_i, with $2 \leq i \leq n$.

A comparator is a function that takes two patches π_1 and π_2 and returns a negative value, zero, or a positive value depending on whether π_1 proceeds π_2, π_1 and π_2 coincide, or π_1 comes after π_2, respectively, in terms of their similarity to the original program or their impact on branch coverage.

The function `groupAndSort` depends on a hashtable data structure, called `OrderedMap`, whose keys are ordered based on a given comparator. Lines 18 and 19 do an initial grouping by adding the patches in the ordered hashtable. Thanks to the selected comparator c = `head(comparators)`, after Line 19, the hashtable will be a sorted table of entries in the form of key-value pairs $< \pi, t >$ where t is a set of all patches π' such that $c(\pi, \pi') = 0$. Note that $\pi \in t$ as well, as we always have $c(\pi, \pi) = 0$. Lines 20-26 iterate through ties of non-trivial size and make recursive calls to the function to break the remaining ties using the next comparator in the list.

Lines 2-6 define the three comparators with the heuristic order that we propose. The comparators are defined using lambda notation [70] for the sake of brevity and readability. The ordering dictated in the list, makes `groupAndSort` function (called at line 8) to group and sort the patches (in descending order), first based on their impact on branch coverage. Then the ties are broken by grouping and sorting (in descending order), based on token-level similarity. Finally, break any remaining ties are broken by grouping and sorting (in descending order), based on behavioral similarity.

Algorithm 1: Patch ranking in Shibboleth

Input: Patches Π and functions **BC**, **TS**, and **SCS**
Output: Sequence $\{< \pi, r >\}_{\pi \in \Pi}$, sorted based on ranks r in ascending order

```
1  begin
2      Comparators ← [
3              λ π1, π2. BC(π2) − BC(π1),
4              λ π1, π2. TS(π2) − TS(π1),
5              λ π1, π2. SCS(π2) − SCS(π1)
6          ]
7      Output ← []
8      Ties ← groupAndSort(Π, Comparators)
9      rank ← 0
10     for Tie in Ties do
11         rank ← rank + size(Tie)
12         for π in Tie do
13             Output.append(<π, rank>)
14     return Output
15 function groupAndSort(Patches, Comparators):
16     Result ← []
17     map ← new OrderedMap(head(Comparators))
       // Group patches via first element in comparators:
18     for π in Patches do
19         map[π] ← map[π] ∪ {π}
       // Break ties recursively via the next comparator:
20     for <π, Tie> in map do
21         if size(Tie) > 1 ∧ size(Comparators) > 0 then
22             G ← groupAndSort(Tie, tail(Comparators))
23             for T in G do
24                 Result.append(T)
25         else
26             Result.append(Tie)
27     return Result
```

The intuition behind this ordering comes from the observations we made about **BC**, **TS**, and **SCS** values: **BC** is the most effective measure, while **TS** and **SCS** are equally effective in ranking the patches. We observed that the algorithm is robust to the order of comparators corresponding to **TS** and **SCS**, but the comparator corresponding to **BC** has to appear as the first element in `Comparators` so as to achieve the best results. We attribute this to the fact that in the cases where **BC** falls short of ranking the correct patch in the top-1 position, all the patches wind up with equal **BC** values. Most of these sets of patches can be handled using **TS** and **SCS**.

Lastly, lines 10-14 produce the output list by giving the patches within each group the *worst-case* ranking. By the worst-case ranking, we mean that if we encounter a tie of size k and there are m patches ranked before them, each patch within the tie shall receive a rank of $m + k$.

3.5 Classification

For classification purposes, Shibboleth can use any binary classifier, however, as we will see in §4, compared to five other classifiers we experimented with, a Random Forest classifier [37] with 100 Decision Trees [88] as the base estimators (default configuration of Python's `scikit-learn` machine learning library [76]), yields the highest accuracy and F1 values. The classifier model is constructed during a *training phase*, which uses a training data set of the following form

$$\{\langle \mathbf{BC}(\pi), \mathbf{TS}(\pi), \mathbf{SCS}(\pi), \ell(\pi)\rangle \mid \pi \in \Pi\},$$

where Π is a set of 1,871 patches in our data set.

In other words, to obtain the training data points, for each patch π, we bundle all the following information together with the ground-truth label for the patch: (1) difference in the branch coverage of the originally passing test cases before and after patching the program with π; (2) the token-level cosine similarity of the body of the method patched by π in the original program P and its patched version $\pi(P)$; (3) the cosine similarity of Statement-Count Spectrum vectors calculated under the originally passing test cases. The user is given the option of down-sampling the training data set to make it balanced. Once the model is trained, it can be used to classify future patches. Classification works as follows. Given a program P and patch π targeting method m of P, Shibboleth constructs tuple of feature values $\langle \mathbf{BC}(\pi), \mathbf{TS}(\pi), \mathbf{SCS}(\pi)\rangle$. The pre-trained classifier is then applied to the tuple to predict the label. The predicted label is returned as the output of the classification algorithm.

3.6 Implementation

The current implementation of Shibboleth targets Java [33], but in the approach design we did not make any assumption about the programming language and we emphasize that the technique is programming language agnostic and it can be implemented for other programming languages with some engineering effort.

Shibboleth uses the ASM bytecode manipulation framework [68] and relies on Java Agent technology [67] to instrument the target program and calculate statement count spectra and branch coverage. To minimize the need for running test cases, Shibboleth calculates method coverage information before profiling and only runs the covering passing test cases for calculating statement count spectra and branch coverage. Shibboleth also calculates statement count spectra and branch coverage both at the same time within a single JVM session. To calculate textual similarity, we used the Java Parser library [40] to parse the program, before and after patching, into tokens and count the tokens to obtain frequency vectors before calculating their cosine similarity. For implementing the ranking subsystem, as ordered hashtables, we use `TreeMap`, which is based on an efficient red-black tree algorithm [12] and is available within the JDK standard library. For implementing the classification subsystem, we use Python's `scikit-learn` machine learning library [76]. Shibboleth is implemented as a one-click Maven-plugin, as well as a command-line tool, and it is publicly available [32].

4 EMPIRICAL EVALUATION

We perform an empirical evaluation of Shibboleth, comparing its ranking and classification performance with a set of state-of-the-art approaches. Shibboleth performs both ranking and classification, so we evaluate these separately. Specifically, we aim to answer the following research questions.

- **RQ1**: How does Shibboleth perform when ranking patches?
- **RQ2**: How does Shibboleth perform when classifying patches?

Table 1: APR systems that generated the patches in our data set, categorized into three classes of heuristic, semantic-based, and template-based techniques

Class	APR Tools
Heuristic	ARJA [99], ARJA-E [100], jGenProg [60], jKali [60], jMutRepair [60], SimFix [41], CapGen [85], RSRepair [71], SequenceR [10], ELIXIR [73], DeepRepair [86], ssFix [90], PraPR [31], 3sFix [11], GenProg-A [99], and Hercules [74]
Semantic-based	ACS [93], NOPOL [94], DynaMoth [23], Cardumen [61], JAID [9], and SketchFix [38]
Template-based	kPAR [52], AVATAR [54], FixMiner [46], TBar [53], SOFix [55], ConFix [44], and HDRepair [17]

4.1 Data Set of Patches

We constructed a patch data set by combining the set of human-written patches from the Defects4J v2.0.0 bug database [19] and four curated patch data sets, used in recent studies [84, 91, 97, 100], generated by 29 APR systems (see Table 1). Combining the five data sets resulted in a total of 3,072 patches consisting of 1,684 patches labelled as *correct* and 1,374 patches labelled as *incorrect*. 14 of the patches were labelled as *unknown*; these patches are from the data set of Xiong *et al.* [91]. Human-written patches are generally regarded as correct-by-definition [48, 49, 64, 78] and APR-generated patches were analyzed and labelled in previous works.

These data sets were curated by different research groups at different times, so they overlap, hence we identified and excluded duplicate patches. To reduce manual work in duplicate elimination, we used a script to automatically remove patches that were identical to one another ignoring any white-spaces. We were able to remove 487 patches with the help of this script. Automatic duplicate elimination is performed by computing the SHA-1 hashcode of the body of the diff files and keeping only one copy of the set of patches with the same hashcode. This duplicate elimination method cannot eliminate semantically equivalent patches, *e.g.*, the expressions `a+b` and `(a)+b` are semantically equivalent while they have different SHA-1 hashcodes. Therefore, a manual inspection is necessary, hence we manually sieved through 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 excluded Defects4J patches that involved creation/removal of files, as annotating these patches requires substantial knowledge about the code base and deeper analyses, and neither our tool nor are any of state-of-the-art systems capable of handling such patches. For the same reason, we excluded all of the patches that involved

creation/deletion of classes or methods. Lastly, we excluded 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 implementation of Shibboleth and/or other studied tools.

After this pre-processing, we were left with 1,871 patches, 778 of which are labelled as correct while 1,093 are labelled as incorrect. All these patched are generated for Defects4J bugs, and each bug has one correct patch and one or more incorrect patch(es) generated for it. We have annotated the patches with additional information that can come in handy in APR research. Here is an outline of the information accompanying each patch in the data set and a brief discussion on their possible applications:

- The Defects4J bug id targeted by the patch,
- Package name, source file name, line number and line range of the patched location, which can be useful for source-level analysis of the patched code,
- Ground-truth label of the patch (*i.e.*, correct/incorrect),
- Fully qualified names of the patched methods, which can be useful for bytecode-level analysis of the patched code,
- The Ochiai suspiciousness values of the patched methods,
- The name of the generating APR tool (or `N/A` in cases where the patch is human-written),
- Provenance information [7], *i.e.*, which data set each patch comes from and the original identifier of the data point in that data set. Using this information, the name and contact information of the original curators can be traced so that any doubts about the validity of the labels can be disputed.

To answer RQ2, we used all the 1,871 patches in our finalized data set. For RQ1, however, we further excluded the correct patches that were not paired with any incorrect patches, as ranking a single correct patch does not make sense. This resulted in 1,290 patches among which 197 are labelled as correct and the rest 1,093 are labelled as incorrect.

4.2 Baseline Approaches

We compare Shibboleth's ranking performance (*i.e.*, RQ1) with the random baseline obtained *via* hypergeometric probability [5] of a correct patch appearing in top-1 or top-2. We also compare Shibboleth with the following state-of-the-art patch ranking techniques:

- Latest version of ObjSim [30], capable of handling multi-hunk patches studied in this paper,
- CIP [100], the patch ranking/classification system, part of ARJA-E [100], and
- Ranking based on Ochiai suspiciousness values [87].

ObjSim and CIP are object similarity-based approaches, *i.e.*, they quantify behavior change between the patched version and the original version of the program by comparing a system state snapshot at the exit points of the patched methods, represented as object graphs. The two systems use similar functions for quantifying the difference between system state snapshots. However, ObjSim takes the effect of both passing and failing tests into account and analyzes several past snapshots, while CIP focuses on passing tests only and takes only the last snapshot into account. Furthermore, CIP ranks the patches by first classifying them into two groups of likely correct and likely incorrect patches. The block of likely correct patches comes before the block of likely incorrect patches in the fix report, and the latter is sorted based on three heuristics in tandem to rank and break the ties. We refer the reader to the original publications [29, 100] for more details.

Ochiai-based fault localization is employed by virtually all test-based G&V APR systems, which affects their effectiveness [52]. APR tools generally transform the most suspicious program elements [49, 52], *i.e.*, Ochiai suspiciousness values are (indirectly) used for ranking. Therefore, by studying Ochiai-based ranking we can gain an understanding on how much improvement each ranking system brings about, compared with no ranking in place.

We compare Shibboleth's classification performance (*i.e.*, RQ2) with the following state-of-the-art patch classification techniques.

- PATCH-SIM [91], a dynamic patch classification system based on complete path spectra,
- CIP [100] with ranking information discarded,
- The static patch classification system introduced by Tian *et al.* [80], and
- ODS [97], a static patch classification technique based on 4,199 source code features.

PATCH-SIM records complete path spectra [35] (*i.e.*, induced by the sequence of instructions executed) before and after patching running passing and failing tests. It then uses the Longest Common Subsequence (LCS) algorithm [12] to calculate the distance between sequences before and after patching; it takes the average distance for the passing test cases, and takes the maximum distance for failing tests. The tool filters out the patches that result in a distance value below a certain threshold. PATCH-SIM filters out more incorrect patches with the help of automated test case generation [69]. We postpone studying the effect of test case generation on the performance of Shibboleth for future work. For the time being, we use the default configuration of PATCH-SIM, namely without test case generation, for the sake of a fair comparison.

Tian *et al.* [80] revisit the idea of using embeddings instead of feature engineering [14] to avoid possible overfitting that techniques like ODS are susceptible to. The system combines BERT [21] with a Logistic Regression classifier [39] to classify the patches statically.

We emphasize that the list of related work (both for ranking and classification) studied in this work is not exhaustive. We focus our evaluation only on these techniques, for they each have a prototype implementation, which is publicly available or the data they were evaluated on is part of our patch data set. Re-implementing other (currently unavailable) techniques and comparing them against Shibboleth will be part of future work.

4.3 Performance Measures

4.3.1 Ranking Measures. For assessing Shibboleth's ranking performance (*i.e.*, RQ1) we follow the convention of using worst-case ranking and counting the number of correct patches that appear in the top-N position. This measure has been widely used in fault localization research [87], and it lends itself naturally for assessing the performance of patch ranking systems, as well. Existing studies [45] showed that over 70% of developers inspect only the top-5 ranked elements. However, we focus on the top-1 and top-2, to assess the performance of the patch ranking systems. The reason that we are not focusing on top-5 or higher N-values is that our data set has some cases with fewer than five patches per bug. However,

Table 2: Ranking results per project for the four ranking schemes TS, SCS, BC, and Shibboleth

Project	#Pl	#C	#I		TS	SCS	BC	Shibboleth
Chart	201	19	182	**Top-1**	4	10	6	11
				Top-2	7	14	6	14
Closure	269	64	205	**Top-1**	18	10	23	27
				Top-2	41	32	41	47
Lang	220	35	185	**Top-1**	10	12	10	14
				Top-2	21	21	19	22
Math	541	67	474	**Top-1**	22	23	15	27
				Top-2	36	35	26	38
Mockito	2	1	1	**Top-1**	0	1	0	1
				Top-2	1	1	1	1
Time	57	11	46	**Top-1**	2	4	4	5
				Top-2	7	8	7	8
Total	1290	197	1093	**Top-1**	58	56	60	85
				Top-2	100	113	111	130

*Column **# Pl.** lists total number of plausible patches, **# C** lists number of patches labelled as correct, and **# I** lists the number of incorrect patches. The rest of columns report number of correct patches that appear in top-1/top-2 position.

there are at least two patches for every bug in the data set. The measures are a proxy estimating developer effort. The number of top-1 patches indicates that developers only need to inspect one patch to find the correct one for that bug, while the number of top-2 patches indicate that for those bugs, the users need only to inspect two patches. The systems with higher top-1 and top-2 values are considered better.

4.3.2 Classification Measures. Following previous work in patch correctness assessment [80, 84, 91, 97], we measure the accuracy and F1 score (as a representative of precision and recall) for assessing the Shibboleth's classification performance (*i.e.*, RQ2), which are defined as follows:

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{3}$$

$$\text{F1} = \frac{TP}{TP + 0.5(TP + FN)} \tag{4}$$

In the definitions above, TP (true positive) denotes the number of times a correct patch is identified as correct, TN (true negative) denotes the number of times an incorrect patch is identified as incorrect, FP (false positive) denotes the number of times an incorrect patch is identified as correct, and FN (false negative) denotes the number of times a correct patch is identified as incorrect. Higher accuracy and F1 score indicate a better performance. From a practical point of view, the goal of a patch classifier is to filter out as many incorrect patches as possible, while avoiding the missclasification of correct patches. In the cases where there is more than one correct patch, it is acceptable to have false positives, for maximizing the true negatives. Hence, existing research [80, 84, 91, 97] also analyzed the classification performance by comparing the values of TN and TP, which we also do when answering RQ2.

4.4 Results

We present the results for each research question.

Table 3: Shibboleth versus state-of-the-art patch ranking techniques and a random baseline on the data set of Table 2

Project		Shibboleth	CIP	ObjSim	FL	Rand (avg)
Chart	**Top-1**	11	2	4	3	2.4
	Top-2	14	6	8	6	4.8
Closure	**Top-1**	27	1	17	19	16
	Top-2	47	24	36	38	32
Lang	**Top-1**	14	2	2	1	7
	Top-2	22	17	15	12	14
Math	**Top-1**	27	8	12	10	13.4
	Top-2	38	27	28	30	26.8
Mockito	**Top-1**	1	1	1	0	0.5
	Top-2	1	1	1	1	1
Time	**Top-1**	5	5	3	3	3.7
	Top-2	8	8	6	5	7.3
Total	**Top-1**	85	19	39	36	43
	Top-2	130	83	94	92	85.9

4.4.1 Answering RQ1 (Shibboleth's Ranking Performance). We carried out our ranking analysis on 1,290 patches, wherein every correct patch is paired with at least one incorrect patch.

As described in the previous section, we want to rank the patches in descending order, based on **TS**, **SCS**, and **BC** values. These individual features result in 58 (100), 56 (113), and 60 (111), respectively, correct patches in top-1 (top-2) position (see Table 2). Figure 4 shows the overlap between the three measures for the top-1 ranks. We conclude that the three measures are complementary in that there are patches that only a certain feature can rank in top-1 position. This motivated us to combine these features and rank more patches in the top-1 (and top-2) position. Sorting and ranking the patches first according to **BC** and breaking the ties using **TS** and then using **SCS** results in the best performance (*i.e.*, 85 top-1 and 130 top-2), hence we denote this combination as Shibboleth. Table 2 shows the performance of all four ranking schemes. According to the table, the combined ranking scheme (*i.e.*, Shibboleth) consistently outperforms the single measure-based ranking schemes both in terms of top-1 and top-2.

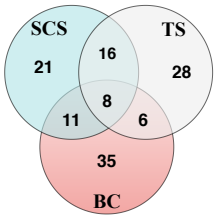

Figure 4: Distribution of bugs wherein correct patches are ranked in top-1 position using individual features.

Table 3 compares Shibboleth with state-of-the-art ranking techniques CIP, ObjSim, Ochiai fault localization-based ranking, and a random baseline based on hypergeometric probability. The results indicate that Shibboleth outperforms other ranking system, and the random baseline, across all of the Defects4J projects.

Table 4: 10-fold cross validation of six machine learning algorithms using Shibboleth. Acc denotes Accuracy.

Learner	TP (avg)	TN (avg)	FP (avg)	FN (avg)	Acc	F1
AdaBoost	66.1	64.4	13.4	11.7	0.839	0.838
Decision Tree	65.7	64.8	13	12.1	0.839	0.838
Logistic Regression	69.3	31.2	46.6	8.5	0.646	0.548
Naïve Bayes	73.4	56.2	21.6	4.4	0.833	0.812
Random Forest	105	60.9	16.9	4.3	0.887	0.852
SVM	72.7	31.6	46.2	5.1	0.67	0.552

Shibboleth ranks a correct patch in the top-1 (top-2) position for 43% (66%) of the bugs and it outperforms state-of-the-art patch ranking techniques.

4.4.2 Answering RQ2 (Shibboleth's Classification Performance). We use the three features **TS** (textual similarity), **SCS** (behavioral similarity), and **BC** (branch coverage delta) for classifying patches as described in §3. With these features we build six Shibboleth classifiers, using different machine learning algorithms: AdaBoost [26], Decision Tree [88], Logistic Regression, Gaussian Naïve Bayes Classifier [39], Random Forest Classifier, and Support Vector Machines (SVM) [13]. We used the 1,871 patches described in §4.1 and performed a 10-fold cross validation to assess the performance of the classifiers. To compensate for the imbalance in our data set, we used undersampling [36] *via* `NearMiss` class in Python's `scikit-learn` library [76]. For all of the learners, except for AdaBoost, we used `scikit-learn`'s default hyper-parameters. AdaBoost performed best with 45 Decision Trees.

Table 4 reports the performance of the six classification algorithms as measured by accuracy and F1 score. The table also lists average *TP*, *TN*, *FP*, and *FN* obtained from 10-fold cross validation. Since Random Forest performs best among other algorithms, we use this algorithm in Shibboleth's classification subsystem.

We compare Shibboleth with state-of-the-art static and dynamic patch classification systems, namely ODS [97] and the static approach of Tian *et al.* [80], named BERT-LR, and the dynamic patch classification systems PATCH-SIM [91] and CIP [100].

Table 5: Comparing Shibboleth (with Random Forest as its underlying classifier), PATCH-SIM, CIP, BERT-LR, and ODS using 130 patches.

	Shibboleth	PATCH-SIM	CIP	BERT-LR	ODS
TP	22 (23)	28 (29)	25 (26)	28 (29)	23 (24)
TN	97 (101)	58 (62)	37 (37)	42 (44)	63 (70)
FP	5 (9)	44 (48)	65 (72)	60 (66)	39 (40)
FN	6 (6)	0 (0)	3 (4)	0 (0)	5 (5)
Acc	0.915 (0.892)	0.662 (0.655)	0.477 (0.453)	0.538 (0.525)	0.662 (0.676)
F1	0.8 (0.754)	0.56 (0.547)	0.424 (0.406)	0.483 (0.468)	0.511 (0.516)

* The values in parentheses correspond to using 139 patches, reported in previous research, which include 9 duplicates. Acc denotes Accuracy.

Unfortunately, due to environment setup and high computational requirements of PATCH-SIM, we could not run it on the entire data set of 1,871 patches. Therefore, following recent works [80, 97], we reused the published results of PATCH-SIM in our experiments, instead. The published evaluations used 139 patches, which are part of our data set. Among these 139 patches, there are 9 duplicate patches [80, 84], which we eliminate them from the evaluation. We trained Shibboleth on 1,160 patches (*i.e.*, 1,871 from which 130 patches are excluded) and used the 130 data points as test data. We ran BERT-LR [80] and CIP [100] on the 139 data points.

In order to facilitate easier comparison with previously published results [80, 97], we include the classification results on 139 patches (including the 9 duplicates), but we report them in parentheses.

Table 5 reports the performance of Shibboleth, compared with the state-of-the-art static and dynamic techniques, as measured by *TN*, *FP*, *FN*, and *TP*, as well as accuracy and F1 score. As we can see from the table, Shibboleth correctly classifies the majority of the patches, and outperforms all the baseline techniques.

Random Forest is the best model to use with Shibboleth and it outperforms state-of-the-art static and dynamic patch classifiers.

5 DISCUSSION

When designing or analyzing patch correctness assessment techniques, classification accuracy is only one dimension to consider. The usability, robustness, and efficiency of the techniques are also important. While these depend on implementation decisions, they are also direct consequences of the design decisions. For example, dynamic techniques tend to have relatively high accuracy, which comes at the expense of efficiency. Conversely, static techniques are often much faster, yet less precise. Using complex machine learning models bring additional complexities related to the training of the models [80].

Our aim with Shibboleth is to strike a balance between these competing requirements. Shibboleth is a hybrid technique that blends features both from production code and test code, and can be implemented quite efficiently with minimal overhead on test execution. When computing the branch coverage, as well as syntactic and semantic similarity, for the 1,871 patches, Shibboleth has a small memory footprint, and we observed an overhead of only 10.24% on test execution.

In contrast, PATCH-SIM relies on detailed complete path spectra and the LCS algorithm (with a non-linear time and space complexity [12]). Existing research [84, 97, 100] reported that running PATCH-SIM is computationally intensive. However, on the 130 data points that we reported, PATCH-SIM achieves better precision compared to other techniques, except Shibboleth.

Despite relying on code embeddings provided by BERT [21], BERT-LR, being a purely static technique is probably the most efficient of the compared approaches, as it only parses the patch files and makes comparisons. BERT-LR also suffers from lower precision than PATCH-SIM.

We consider the Shibboleth as the "sweet spot" between the extremes: it achieves better accuracy than the dynamic and static competitors, while maintaining a low computational complexity (calculating **TS** is linear in the size of the program and **BC** and **SCS** are computed in constant time and space) and runtime overhead, to the extent that, from the point of view of a user, ranking/classifying patches using Shibboleth takes more or less the same time as one execution of the test suite.

5.1 Limitations

Given n copies of a simple statement (*i.e.*, a statement without conditionals) S in a program, we can generate an arbitrary number of patches, which Shibboleth will not be able to distinguish, as follows. Increase the number of copies of S to $n+k$ times or decrease them to $n-k$ times, with $n-k>0$. Such patches amount to points (*i.e.*, **BC**, **TS**, and **SCS** triples) that are along the same direction (*i.e.*, they are on the same vector), with equal cosine similarity, hence Shibboleth will not differentiate them. Despite this shortcoming of our approach, we did not find any pair patches among the studied 1,871 patches that fall in the "blind spot" of Shibboleth.

5.2 Threats to Validity

Like most empirical evaluations, our results and conclusions are impacted by our design decisions and are subjects to certain threats to validity, which we discuss here.

There is an implicit assumption behind dynamic patch correctness assessment approaches [29, 91, 100] that the computation is deterministic. Non-deterministic behavior, whether it is intended or not, can cause problems for dynamic techniques. As an approach that relies on dynamic information, Shibboleth is not immune to the problem caused by non-determinacy in programs. To reduce the effects of this threat, we measured coverage multiple times to make sure that the fluctuations in the code coverage was due to patching and not to some other factor. We compared code coverage before patching to make sure that they are the same. Our conclusions about the Shibboleth's ranking and classification performance depends on the measures we use to assess the performance. To minimize bias in our evaluation, we used a set of measures commonly used in existing related work (see §4.3). In addition, we drew our conclusions on Shibboleth's performance after comparing it with that of several existing approaches that have available implementations or their published results intersects our data set (see §4.2). Comparing with additional approaches is subject of future work. The generalization of our conclusions depends on the data we used. We do not have a working definition of representative sample for patches, but we made efforts to ensure that the data we used in the evaluation is as representative as possible with minimizing potential biases: our patch data (see §4.1) is a superset of data used in previous related research, and the patches come from six different Java systems produced manually (presumably by different developers) or generated by 29 different APR tools.

6 RELATED WORK

In this section we discuss how Shibboleth fits among the APR-related techniques for patch assessment using ranking and classification.

6.1 Prioritization/Ranking

Various APR techniques use static or dynamic methods to prioritize patches for patch validation or rank the plausible patches for easier manual inspection. Virtually all test-based G&V techniques transform the most suspicious program elements (typically measured by spectrum-based Ochiai suspiciousness values [87]) to generate patches [52], thereby inducing an ordering based on Ochiai suspiciosness value of the patched location among the pacthes. Some techniques, notably PraPR [31], mutate all locations with non-zero suspiciousness values and rank plausible patches based on Ochiai suspiciosness value of the patched locations, after the fact. Our experiments show that Shibboleth outperforms Ochiai-based ranking. S3 [16], DirectFix [62], and Qlose [24] use static syntactic and semantic features to quantify proximity of generated patch to the original program and prioritize the patches that are close to the original program over the ones that involve significant modifications. Prophet [57], ELIXIR [73], and ODS (ranking mode) [97] use pre-trained statistical models to rank the patches based on syntactic features. In this paper, we focus on ranking systems that are publicly available, or for which published results are reusable, and leave re-implementing ranking systems based on the static features used in aforementioned techniques as a future work. ObjSim [29] and CIP [100] are dynamic ranking techniques that quantify the behavior change by comparing system state snapshots at the exit points of patched methods. Shibboleth (in ranking mode) outperforms ObjSim and CIP, while it is computationally less demanding.

6.2 Classification

6.2.1 Static. Tan *et al.* [79] propose anti-patterns to avoid incorrect patches in test-based G&V APR techniques. As shown by a recent study [84] anti-patterns alone are not accurate enough and might have destructive effects by dismissing many correct patches, not to mention that some of the rules listed in [79] are not automatable. Our approach is in part inspired by anti-patterns in that we deprioritize patches that decrease code coverage. Ye *et al.* [97] introduce ODS implementing a machine learning algorithm which is an ensemble model based on 4,199 source code features. ODS (in classification mode) achieves better precision compared to dynamic patch classification technique PATCH-SIM [91] and it is significantly faster. Since ODS depends on thousands of handcrafted feature, the resulting model is susceptible to the overfitting problem [14, 80]. Csuvik *et al.* [14] propose to use source code embedding methods instead of performing feature engineering. Tian *et al.* [80] revisit this preliminary idea and show that the technique is as effective as PATCH-SIM [91] and it outperforms ODS. Shibboleth correctly classifies the majority of the patches and it outperforms the technique introduced in [80].

6.2.2 Dynamic. PATCH-SIM [91] is the state-of-the-art dynamic patch classification technique, which quantifies program behavior changes by computing LCS distance of complete path spectra [35] before and after patching. It uses a heuristic cut-off threshold to filter out the patches that fall below certain level of similarity to the original program. CIP [100], which uses a two-tiered ranking system (classification first and then ranking of likely incorrect patches) has been evaluated as a patch classification system also. Although CIP is able to discriminate certain patches that techniques like PATCH-SIM are unable to distinguish and it is more efficient than PATCH-SIM, CIP is less effective than PATCH-SIM in terms of accuracy and precision [100].

A family of approaches [6, 22, 95] use dynamically inferred invariants (*e.g.*, using Daikon [25]) as an abstraction of the program behavior and use syntactic distance metrics to quantify the differences. However, inferring invariants is computationally demanding and off-the-shelf tools are still in their infancy [95].

DiffTGen [89] helps researchers classify the patches generated by their APR systems into plausible and genuine. In order to conduct classification, DiffTGen needs a human-written patch as a reference. The tool also relies on the end-user to provide oracles for the generated test cases based on the observed semantic differences. DiffTGen and Shibboleth are related in that they both focus on the test suite adequacy. However, Shibboleth and DiffTGen pursue fundamentally different goals. Shibboleth is a technique and a fully automatic tool that aims to improve the usability of APR systems in real-world applications. DiffTGen is a manual tool intended to help APR researchers in classifying generated patches with more confidence, which has gained success as witnessed by recent studies [15, 98].

A group of techniques [28, 96] apply fuzzing methods to filter out the patches that crash in various forms. Although such approaches work satisfactorily for the APR systems targeting C/C++ programming language, they are not as effective in the context of managed programming languages like Java [91]. Generalizing these techniques with inferred and/or user-defined assertions would be an interesting topic to explore in the future.

Most recently, Shariffdeen *et al.* [77] present CPR for detecting incorrect patches patches through systematic exploration of the patch space and input space *via* concolic path exploration.

6.2.3 Hybrid. Wang *et al.* [84] propose combining the static features of S3 [16], the complete path spectra similarity computed by PATCH-SIM [91], and the anti-patterns [79] to obtain an integrated model that is more effective than any individual technique. PATCH-SIM as stated in its GitHub repository [92], and reported by at least three studies [84, 97], is computationally demanding. Since the integrated model of Wang *et al.* depends on PATCH-SIM, the model suffers from the same performance bottleneck. In this work, we complement previous studies by taking into account static and dynamic measures from both production and test code to assess correctness of the patches. We plan re-implementing the model by Wang *et al.* and comparing with Shibboleth in a future work.

7 CONCLUSIONS

Shibboleth is a novel technique for patch correctness assessment which, unlike existing patch evaluation approaches, takes into account the impact of patches on both production and test code and it relies on a simpler set of assumption. Shibboleth also relies on light-weight measurements, making it computationally efficient. Our empirical evaluation with 1,290 patches showed that Shibboleth ranks a correct patch in the top-1 (top-2) position for 43% (66%) of the bugs outperforming state-of-the-art approaches. When used as a classifier on 1,871 patches, Shibboleth achieves an accuracy and F1-score of 0.89 and 0.85, respectively, thereby outperforming sate-of-the-art patch classification techniques.

ACKNOWLEDGEMENTS

We thank Anonymous ISSTA Reviewers for their constructive feedback.

REFERENCES

[1] James H Andrews, Lionel C Briand, and Yvan Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *ICSE*. 402–411.
[2] Andrea Arcuri. 2011. Evolutionary repair of faulty software. *ASC* 4 (2011), 3494–3514.
[3] Johannes Bader, Andrew Scott, Michael Pradel, and Satish Chandra. 2019. Getafix: Learning to fix bugs automatically. *OOPSLA* (2019), 1–27.
[4] Earl T. Barr, Yuriy Brun, Premkumar Devanbu, Mark Harman, and Federica Sarro. 2014. The plastic surgery hypothesis. In *FSE*. 306–317.
[5] Matthew A Carlton and Jay L Devore. 2017. *Probability with applications in engineering, science, and technology*. Springer.
[6] Padraic Cashin, Carianne Martinez, Westley Weimer, and Stephanie Forrest. 2019. Understanding automatically-generated patches through symbolic invariant differences. In *ASE*. 411–414.
[7] Adriane Chapman, James Cheney, and Simon Miles. 2017. Guest Editorial: The Provenance of Online Data. *TIT* 4 (2017), 1–3.
[8] Lingchao Chen, Yicheng Ouyang, and Lingming Zhang. 2021. Fast and Precise On-the-fly Patch Validation for All. In *ICSE*. 1123–1134.
[9] Liushan Chen, Yu Pei, and Carlo A. Furia. 2017. Contract-based program repair without the contracts. In *ASE*. 637–647.
[10] Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. SequenceR: Sequence-to-sequence learning for end-to-end program repair. *TSE* (2019), 1–1.
[11] Zimin Chen and Martin Monperrus. 2018. The remarkable role of similarity in redundancy-based program repair. *arXiv* (2018).
[12] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms*. MIT press.
[13] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *ML* 20 (1995), 273–297.
[14] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *IBF*. 18–25.
[15] 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.
[16] 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.
[17] Xuan-Bach D. Le, David Lo, and Claire Le Goues. 2016. History driven automated program repair. In *SANER*. 213–224.
[18] Vidroha Debroy and W. Eric Wong. 2010. Using mutation to automatically suggest fixes for faulty programs. In *ICST*. 65–74.
[19] Defects4J Contributors. 2020. http://bit.ly/2PY3yDa. Accessed: 01/22.
[20] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on test data selection: Help for the practicing programmer. *IEEE Computer* 11 (1978), 34–41.
[21] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv* (2018).
[22] Zhen Yu Ding, Yiwei Lyu, Christopher S. Timperley, and Claire Le Goues. 2019. Leveraging Program Invariants to Promote Population Diversity in Search-Based Automatic Program Repair. In *GI*. 2–9.
[23] Thomas Durieux and Martin Monperrus. 2016. Dynamoth: dynamic code synthesis for automatic program repair. In *WAST*. 85–91.
[24] Loris D'Antoni, Roopsha Samanta, and Rishabh Singh. 2016. Qlose: Program repair with quantitative objectives. In *CAV*. 383–401.
[25] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. 2007. The Daikon system for dynamic detection of likely invariants. *SCP* 69 (2007), 35–45.
[26] Yoav Freund, Robert Schapire, and Naoki Abe. 1999. A short introduction to boosting. *JSJA* 14, 771–780 (1999).
[27] Qing Gao, Yingfei Xiong, Yaqing Mi, Lu Zhang, Weikun Yang, Zhaoping Zhou, Bing Xie, and Hong Mei. 2015. Safe memory-leak fixing for c programs. In *ICSE*. 459–470.
[28] Xiang Gao, Sergey Mechtaev, and Abhik Roychoudhury. 2019. Crash-avoiding Program Repair. In *ISSTA*. 8–18.
[29] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *ISSTA*. 541–544.
[30] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. http://bit.ly/2I62OIs Accessed: 05/22.
[31] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *ISSTA*. 19–30.
[32] Ali Ghanbari and Andrian Marcus. 2022. Patch Correctness Assessment in Automated Program Repair Based on the Impact of Patches on Production and Test Code. https://github.com/ali-ghanbari/shibboleth Accessed: 05/22.
[33] James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. 2014. *The Java Language Specification, Java SE 8 Edition*. Addison-Wesley Professional.
[34] Rahul Gupta, Soham Pal, Aditya Kanade, and Shirish Shevade. 2017. Deepfix: Fixing common c language errors by deep learning. In *AAAI*. 1345–1351.
[35] Mary Jean Harrold, Gregg Rothermel, Rui Wu, and Liu Yi. 1998. An Empirical Investigation of Program Spectra. In *PASTE*. 83–90.

[36] Haibo He and Edwardo A Garcia. 2009. Learning from Imbalanced Data. *TKDE* 9 (2009), 1263–1284.
[37] Tin Kam Ho. 1995. Random decision forests. In *ICDAR*. 278–282.
[38] Jinru Hua, Mengshi Zhang, Kaiyuan Wang, and Sarfraz Khurshid. 2018. Towards practical program repair with on-demand candidate generation. In *ICSE*. 12–23.
[39] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2013. *An introduction to statistical learning*. Vol. 112. Springer.
[40] JavaParser Contributors. 2020. JavaParser. http://bit.ly/381qqvu. Accessed: 01/22.
[41] 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.
[42] Iman Keivanloo, Chanchal K Roy, and Juergen Rilling. 2014. SeByte: Scalable clone and similarity search for bytecode. *SCP* 95 (2014), 426–444.
[43] Dongsun Kim, Jaechang Nam, Jaewoo Song, and Sunghun Kim. 2013. Automatic patch generation learned from human-written patches. In *ICSE*. 802–811.
[44] Jindae Kim and Sunghun Kim. 2019. Automatic patch generation with context-based change application. *ESE* 24 (2019), 4071–4106.
[45] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. 2016. Practitioners' Expectations on Automated Fault Localization. In *ISSTA*. 165–176.
[46] Anil Koyuncu, Kui Liu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, Martin Monperrus, and Yves Le Traon. 2020. Fixminer: Mining relevant fix patterns for automated program repair. *ESE* 25 (2020), 1–45.
[47] Fred Kröger. 1987. *Temporal logic of programs*. Vol. 8. Springer Science & Business Media.
[48] Claire Le Goues, ThanhVu Nguyen, Stephanie Forrest, and Westley Weimer. 2012. GenProg: A generic method for automatic software repair. *TSE* 38 (2012), 54–72.
[49] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* 62 (2019), 56–65.
[50] A. Solar Lezama. 2008. *Program synthesis by sketching*. Ph.D. Dissertation. University of California, Berkeley.
[51] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. 2014. *The Java virtual machine specification*. Pearson Education.
[52] Kui Liu, Anil Koyuncu, Tegawendé F. Bissyandé, Dongsun Kim, Jacques Klein, and Yves Le Traon. 2019. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *ICST*. 102–113.
[53] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. TBar: revisiting template-based automated program repair. In *ISSTA*. 31–42.
[54] Kui Liu, Anil Koyuncu, Dongsun Kim, and Tegawendé F. Bissyandé. 2019. AVATAR: Fixing Semantic Bugs with Fix Patterns of Static Analysis Violations. In *SANER*. 456–467.
[55] Xuliang Liu and Hao Zhong. 2018. Mining stackoverflow for program repair. In *SANER*. 118–129.
[56] Fan Long and Martin Rinard. 2015. Staged program repair with condition synthesis. In *FSE*. 166–178.
[57] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. 298–312.
[58] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. 2019. Aroma: Code recommendation via structural code search. *OOPSLA* (2019), 1–28.
[59] 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.
[60] Matias Martinez and Martin Monperrus. 2016. Astor: A Program Repair Library for Java. In *ISSTA*. 441–444.
[61] Matias Martinez and Martin Monperrus. 2018. Ultra-large repair search space with automatically mined templates: The cardumen mode of astor. In *SBSE*. 65–86.
[62] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. 2015. Directfix: Looking for simple program repairs. In *ICSE*. 448–458.
[63] Ali Mesbah, Andrew Rice, Emily Johnston, Nick Glorioso, and Edward Aftandilian. 2019. DeepDelta: learning to repair compilation errors. In *FSE*. 925–936.
[64] Martin Monperrus. 2018. Automatic Software Repair: A Bibliography. *CSUR* 51 (2018), 1–17.
[65] Martin Monperrus. 2018. *The Living Review on Automated Program Repair*. Technical Report hal-01956501. HAL/archives-ouvertes.fr.
[66] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator Patches Programs Automatically. *Ubiquity* (2019), 1–12.
[67] Oracle Corporation. 2020. Java Agent. https://bit.ly/3czmzFV Accessed: 01/22.
[68] OW2 Consortium. 2020. ASM Bytecode Manipulation Framework. http://bit.ly/3fsPL2r. Accessed: 01/22.
[69] Carlos Pacheco and Michael D. Ernst. 2007. Randoop: Feedback-directed Random Testing for Java. In *OOPSLA*. 815–816.
[70] Benjamin C. Pierce. 2002. *Types and programming languages*. MIT press.
[71] 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.
[72] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. 1997. The Use of Program Profiling for Software Maintenance with Applications to the Year 2000 Problem. In *FSE*. 432–449.
[73] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE*. 648–659.
[74] Seemanta Saha et al. 2019. Harnessing evolution for multi-hunk program repair. In *ICSE*. 13–24.
[75] Eric Schulte, Stephanie Forrest, and Westley Weimer. 2010. Automated program repair through the evolution of assembly code. In *ASE*. 313–316.
[76] scikit-learn Contributors. 2020. `scikit-learn`: Machine Learning in Python. http://bit.ly/3a70cZt Accessed: 01/22.
[77] Ridwan Shariffdeen, Yannic Noller, Lars Grunske, and Abhik Roychoudhury. 2021. Concolic program repair. In *PLDI*. 390–405.
[78] 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.
[79] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.
[80] Haoye Tian, Kui Liu, Abdoul Kader Kaboreé, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F. Bissyandé. 2020. Evaluating representation learning of code changes for predicting patch correctness in program repair. In *ASE*. 981–992.
[81] Ganesha Upadhyaya and Hridesh Rajan. 2018. Collective program analysis. In *ICSE*. 620–631.
[82] Rijnard van Tonder and Claire Le Goues. 2018. Static automated program repair for heap properties. In *ICSE*. 151–162.
[83] Christian Von Essen and Barbara Jobstmann. 2015. Program repair without regret. *FMSD* 47 (2015), 26–50.
[84] 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.
[85] 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.
[86] Martin White, Michele Tufano, Matias Martinez, Martin Monperrus, and Denys Poshyvanyk. 2019. Sorting and transforming program repair ingredients via deep learning code similarities. In *SANER*. 479–490.
[87] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* 42 (2016), 707–740.
[88] Xindong Wu, Vipin Kumar, J. Ross Quinlan, Joydeep Ghosh, Qiang Yang, Hiroshi Motoda, Geoffrey J. McLachlan, Angus Ng, Bing Liu, Philip S. Yu, Zhi-Hua Zhou, Michael Steinbach, David J. Hand, and Dan Steinberg. 2007. Top 10 Algorithms in Data Mining. *KIS* 14 (2007), 1–37.
[89] Qi Xin and Steven P. Reiss. 2017. Identifying test-suite-overfitted patches through test case generation.. In *ISSTA*. 226–236.
[90] Qi Xin and Steven P. Reiss. 2017. Leveraging syntax-related code for automated program repair. In *ASE*. 660–670.
[91] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. 789–799.
[92] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. A tool for identifying patch correctness in test-based program repair. http://bit.ly/390riQb Accessed: 01/22.
[93] Yingfei Xiong, Jie Wang, Runfa Yan, Jiachen Zhang, Shi Han, Gang Huang, and Lu Zhang. 2017. Precise condition synthesis for program repair. In *ICSE*. 416–426.
[94] Jifeng Xuan, Matias Martinez, Favio Demarco, Maxime Clement, Sebastian R. Lamelas Marcote, Thomas Durieux, Daniel Le Berre, and Martin Monperrus. 2017. Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs. *TSE* 43 (2017), 34–55.
[95] Bo Yang and Jinqiu Yang. 2020. Exploring the Differences between Plausible and Correct Patches at Fine-Grained Level. In *IBF*. 1–8.
[96] Jinqiu Yang, Alexey Zhikhartsev, Yuefei Liu, and Lin Tan. 2017. Better test cases for better automated program repair. In *FSE*. 831–841.
[97] 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.
[98] He Ye, Matias Martinez, and Martin Monperrus. 2019. Automated Patch Assessment for Program Repair at Scale. *arXiv* (2019).
[99] Yuan Yuan and Wolfgang Banzhaf. 2018. ARJA: Automated repair of java programs via multi-objective genetic programming. *TSE* 46 (2018).
[100] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *TOSEM* 29 (2020), 1–53.
[101] Jie Zhang, Lingming Zhang, Mark Harman, Dan Hao, Yue Jia, and Lu Zhang. 2018. Predictive mutation testing. *TSE* 45 (2018), 898–918.