Shibboleth: Hybrid Patch Correctness Assessment in Automated Program Repair

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 generate many patches that pass the test suite without fixing the bug. The generated patches must be manually inspected by the developers, a task that tends to be time-consuming, thereby diminishing the role of APR in reducing debugging costs.

We present the design and implementation of a novel tool, named Shibboleth, for automatic assessment of the patches generated by test-based generate-and-validate APR systems. Shibboleth leverages lightweight static and dynamic heuristics from both test and production code to rank and classify the patches. Shibboleth is based on the idea that the buggy program is almost correct and the bugs are small mistakes that require *small changes* to fix and specifically the fix *does not remove the code implementing correct functionality* of the program. Thus, the tool measures the impact of patches on both production code (*via* syntactic and semantic similarity) and test code (*via* code coverage) to separate the patches that result in similar programs and that do not remove desired program elements. We have evaluated Shibboleth on 1,871 patches, generated by 29 Java-based APR systems for Defects4J programs. The technique outperforms state-of-the-art raking and classification techniques. Specifically, in our ranking data set, in 66% of the cases, Shibboleth ranks the correct patch in top-1 or top-2 positions and, in our classification data set, it achieves an accuracy and F1-score of 0.887 and 0.852, respectively, in classification mode.
A demo video of the tool is available at https://bit.ly/3NvYJN8.

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.
ASE '22, October 10–14, 2022, Rochester, MI, USA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3559519

ACM Reference Format:
Ali Ghanbari and Andrian Marcus. 2022. Shibboleth: Hybrid Patch Correctness Assessment in Automated Program Repair. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3551349.3559519

1 INTRODUCTION

Software comes with bugs and debugging is notorious to be time consuming activity. Automated program repair [9] (APR) aims for reducing these costs by fixing bugs with minimal human intervention. Due to its potential in reducing software maintenance costs, it has remained an active research area [11].

Depending on the actions taken to conduct a repair, APR techniques can be categorized into different classes. The majority of current APR techniques belong to the generate-and-validate (G&V) class that attempt to fix the bugs through evolutionary search, heuristic fix templates, code grafting, or random mutation and validating the generated patches using certain checks. The majority of G&V techniques are test-based, *i.e.*, that they validate the generated patches by running test suite against the patches.

Most test-based G&V APR systems are based the following steps. First, a fault localization algorithm is applied to the buggy program to find suspicious program elements. Next, one or more most suspicious program elements are transformed based on a variety of methods (*e.g.*, random mutation, program synthesis, *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 as *plausible patches*. A fix report containing details about plausible patches is then generated and presented to the users for further manual inspection.

Not all patches that pass the test cases are correct, this is because test suites only partially specify the desired behavior of the system (commonly known as *weak specification problem* [9]), APR tools generate many patches that merely pass the available tests without fixing the bug [9, 14] and such patches are called *test case overfitted* or *incorrect* patches. Manually searching for the correct patch is a time-consuming activity; perhaps even more difficult than fixing the bug without using any APR tool [14]. In order to alleviate such manual effort, many techniques are developed for assessing quality of generated patches. Patch correctness assessment could be in the form of *ranking* or *classification*.

In this paper, we present the engineering details of a tool named Shibboleth [7], which is based on a novel treatment of automated patch correctness assessment. Shibboleth relies on the clues from both *production code* and *test code* to assess the correctness of the generated patches. The idea underlying Shibboleth is inspired by the *competent programmer hypothesis* [4] and the previous research

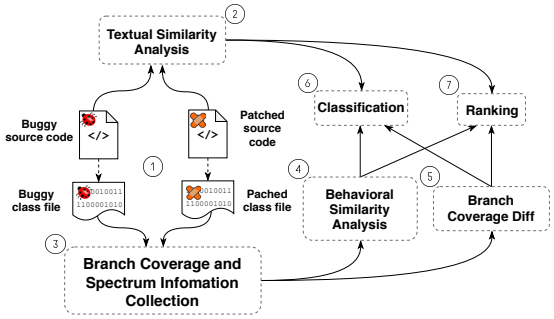

Figure 1: Architecture of Shibboleth

that determined that code-removing program transformations are *anti-patterns* in the context of APR [15]. 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. Therefore, 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 already satisfy the program requirements specification. The impact of patches is captured using three complementary features: *syntactic* (in terms of textual similarity), *semantic* (in terms of similarity of execution traces), and the *code coverage* of the originally passing test cases.

Given a collection of patches, Shibboleth 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. Shibboleth can also classify the patches into likely correct and likely incorrect ones. As such, the users of test-based G&V APR systems are the envisioned users and beneficiaries of Shibboleth.

Using a large set of APR-generated and human-written patches, based on data used by previous research, we have empirically evaluated Shibboleth. Specifically, we have applied Shibboleth on 1,871 patches [7]. The tool achieves an accuracy and F1-score of 0.887 and 0.852, respectively, outperforming state-of-the-art PATCH-SIM [19], CIP [22], ODS [21], and the static patch classification system by Tian *et al.* [16]. We compare the ranking capability of Shibboleth with state-of-the-art patch ranking systems CIP [22], ObjSim$_{mh}$ [5], an Ochiai-based [18] ranking approach, and a random baseline. Random baseline simulates the situation where no ranking mechanism is in place. The results show that, in our ranking data set, in 43% (66%) of the 197 studied bugs, Shibboleth ranks the correct patch in top-1 (top-2) positions, outperforming CIP, ObjSim$_{mh}$, Ochiai-based ranking, and the random baseline.

The source code of Shibboleth and its usage instructions are publicly available [8]. Additionally, a demo video of the tool is available at https://bit.ly/3NvYJN8.

2 THE SHIBBOLETH APPROACH

Shibboleth uses three measures to assess the impact of patches on production and test code and ranks/classifies the patches based on a combination of these measures.

2.1 Measures

We 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. Statement-Count Spectrum is vector of fixed length, each element of which denotes the number of times a particular type of statement is executed [7].

We quantify the *amount of deleted desired program elements* by calculating the difference in branch coverage of the originally passing test cases before and after patching. We define **BC** as the difference in branch coverage of the passing test cases before and after patching.

To calculate the *syntactic similarity*, Shibboleth computes the token-level syntactic similarity of the patched version of the program with respect to its original version. Then it calculates the cosine similarity of the resulting vectors, denoted as **TS**.

We emphasize that behavioral similarity, branch coverage, and token-level syntactic similarity capture the change to the original program from different perspectives and are complementary to each other. Combining the three boosts Shibboleth's effectiveness. We refer the readers to our technical paper [7] for more details on these measures.

2.2 Shibboleth Architecture

The steps taken by Shibboleth to compute **TS**, **SCS**, and **BC** are sketched in Figure 1. The buggy and patched versions of the program are compiled (step ①) to obtain their 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).

The input to both ranking and classification subsystems is a set Π of patches and the functions **BC**, **TS**, and **SCS** measures for each patch $\pi \in \Pi$.

The output of the classification subsystem is a label, either CORRECT or INCORRECT, describing whether or not the patch is likely correct or likely incorrect. The subsystem is a Random Forest classifier trained on all our data points (the data points corresponding to the example bugs in our demo package [8] are excluded). Among five other classical machine learning algorithms, Random Forest proved to perform the best [7].

The output of the ranking subsystem is a sequence of patches paired with their ranks, which is sorted in ascending order of the ranks. Our ranking subsystem implements an algorithm that recursively sorts and groups the patches *via* the aforementioned measures in tandem. The algorithm groups and sorts the patches (in descending order), first based on their impact on branch coverage, then it breaks the ties by grouping and sorting (in descending order) based on token-level similarity, and finally it breaks any remaining ties by grouping and sorting (in descending order) based on behavioral similarity.

The intuition behind this ranking comes from 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 comparing and sorting according to **TS** and **SCS**, but comparing and sorting using **BC** has to be done first for achieving the best results. We attribute this to the fact that in the cases where **BC** does not rank the correct patch in the top-1 position, all the patches wind up with equal **BC** values and most of such sets of patches can be ranked using **TS** and **SCS**.

3 IMPLEMENTATION

The current implementation of Shibboleth targets Java, but the approach described in the previous section does not make any assumption about the programming language, and we emphasize that our 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 and relies on Java Agent technology to instrument the target program and calculate statement count spectra and branch coverage. The tool performs several optimizations to reduce running all the test cases as well as running the program over and over again. Specifically, 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. The tool also calculates branch coverage and statement count spectrum vector at the same time to reduce the need to run the program twice.

These three dynamic analyses are implemented carefully using bit manipulation and best practices in writing efficient Java programs to reduce the instrumentation overhead. Thanks to such an implementation, we observed an overhead of only 10.24% on test execution, which makes Shibboleth remarkably more efficient and faster that virtually all the existing dynamic patch correctness assessment techniques.

To calculate the textual similarity, we used the Java Parser library to parse the program into tokens, before and after patching, and count the tokens to obtain frequency vectors before calculating their cosine similarity.

For implementing the classification subsystem, we used Python's `scikit-learn` machine learning library [13].

Shibboleth is implemented as a one-click Maven-plugin, as well as a command-line tool, and is publicly available [8].

4 EMPIRICAL EVALUATION

We evaluated Shibboleth's performance by comparing its ranking and classification effectiveness with other available patch correctness assessment techniques on a data set of APR-generated and human-written patches.

We constructed a patch data set by combining the set of human-written patches from Defects4J v2.0.0 bug database [3] and four curated patch data sets, used in recent studies [17, 19, 21, 22], generated by 29 APR systems. After a thorough pre-processing and manual analysis [7], we were left with 1,871 patches, 778 of which are labelled as correct while 1,093 are labelled as incorrect.

For evaluating ranking performance of Shibboleth, we constructed our ranking data set as follows. We 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.

4.1 Baseline Approaches

We compare Shibboleth's performance with a naive random baseline obtained *via* the hypergeometric probability of a correct patch appearing in top-1 or top-2. We also compare Shibboleth with the latest version of ObjSim$_{mh}$ [5] capable of handling multi-hunk patches studied in this paper, CIP [22], and Ochiai-based ranking.

4.2 Results

Our empirical evaluation shows that Random Forest is the best model to use with Shibboleth and it outperforms state-of-the-art static and dynamic patch classifiers by achieving an accuracy and F1-score of 0.887 and 0.852, respectively.

We also observed that the ranking based on **BC**, **TS**, and **SCS**, with this order, results in the best ranking performance and 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.

5 SHIBBOLETH USAGE

After checking out Shibboleth from [8] and installing it on the local Maven repository, the tool will be available in the form of a Maven plugin. In order to use Shibboleth to assess the correctness of the patches, the user needs to add the following snippet under `<plugins>` tag in the POM file of the target project.

```
<plugin>
    <groupId>edu.iastate</groupId>
    <artifactId>shibboleth-maven-plugin</artifactId>
    <version>1.0-SNAPSHOT</version>
    <configuration>
        <!-- parameters -->
    </configuration>
</plugin>
```

The tool expects a CSV file, named `input-file.csv`, under the base directory of the project. The input file is intended to contain information about the patches. Each row of this file describes a patch and has to have the following format.

```
Id,Unused,Patched-Methods,Patched-Class-Files
```

Where `Id` is a unique integer identifier of the patch corresponding to the line, `Unused` is an unused value that is ignored by the tool, `Patched-Methods` is a semicolon-separated list of fully qualified names of the patched methods, which is used during instrumentation, `Patched-Class-Files` is a semicolon-separated list of class file names for the patched classes.

The user can configure Shibboleth by setting parameter values under the tag `<configuration>`. Table 1 summarizes the parameters and their default values.

After setting up Shibboleth, the tool can be invoked *via* the command `mvn edu.iastate:shibboleth-maven-plugin:rank`

Table 1: Shibboleth Maven plugin parameters

Parameter	Description
`<targetClasses>`	Target application classes to be transformed. By default, `${groupId}*`, *i.e.*, all application classes.
`<excludedClasses>`	Target application classes to be excluded from transformation. By default, all test cases are selected, *i.e.*, `*Tests`, `*Test`, `*TestCase*`.
`<excludeTestClasses>`	Whether or not test classes should be excluded. By default, `true`, *i.e.*, exclude test classes during coverage analysis.
`<includeProductionClasses>`	Whether or not include production classes. By default, `true`, *i.e.*, all classes under target/classes shall be included.
`<targetTests>`	Target test classes to be included. By default, `*Tests`, `*Test`, `*TestCase*`, all classes that end with `Tests` or `Test`, or contain the word `TestCase`.
`<excludedTests>`	Target test classes to be excluded.
`<inputFile>`	By default, `input-file.csv`, a CSV file containing some basic information about the patches.
`<childJVMArgs>`	By default, `-Xmx16g`, *i.e.*, maximum 16 GB of heap space for child JVM processes for profiling, *etc.*

to rank the patches or the command `mvn edu.iastate:shibboleth-maven-plugin:classify` to classify them.

For more information and a demo, please see the companion video at https://bit.ly/3NvYJN8.

6 RELATED WORK

We discuss how Shibboleth fits among patch correctness assessment systems for APR. Readers are encouraged to refer to Monperrus' literature review [11], for a more comprehensive account of related work on patch classification, and also APR approaches that generate patches aiming to avoid incorrect patches in the first place.

Some techniques, notably PraPR [6], mutate all the locations with non-zero suspiciousness values and rank plausible patches based on Ochiai suspiciosness value of the patched locations, after the fact. Shibboleth significantly outperforms Ochiai-based ranking.

S3 [2] uses static syntactic and semantic features to quantify proximity of the 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 [10], ELIXIR [12], and ODS [21] use pre-trained statistical models to rank and/or classify the patches based on syntactic features. We observed that Shibboleth outperforms state-of-the-art ODS in classification and we leave re-implementing these tools and comparing their ranking performance for a future work.

Tian *et al.* [16] revisit the preliminary idea by Csuvik *et al.* [1] and show that the technique is as effective as PATCH-SIM [19] and it outperforms ODS. Shibboleth correctly classifies the majority of the patches and it outperforms the technique introduced in [16].

PATCH-SIM [20], ObjSim [5] and CIP [22] are dynamic patch classification/ranking tools. Shibboleth outperforms these tools, while it is computationally less demanding.

7 CONCLUSIONS

Shibboleth is a novel tool for patch correctness assessment techniques which, unlike existing approaches, takes into account the impact of patches on both production and test code and it relies on a simpler set of assumption. Shibboleth uses light-weight measurements making it computationally efficient. Our empirical evaluation 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 ranking approaches. The tool also outperforms state-of-the-art patch classification techniques by achieving an accuracy and F1-score of 0.887 and 0.852, respectively. Shibboleth is publicly available [8].

REFERENCES

[1] Viktor Csuvik, Dániel Horváth, Ferenc Horváth, and László Vidács. 2020. Utilizing Source Code Embeddings to Identify Correct Patches. In *IBF*. 18–25.
[2] 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.
[3] Defects4J Contributors. 2020. http://bit.ly/2PY3yDa. Accessed: 05/22.
[4] 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.
[5] Ali Ghanbari. 2020. ObjSim: Lightweight Automatic Patch Prioritization via Object Similarity. In *ISSTA*. 541–544.
[6] Ali Ghanbari, Samuel Benton, and Lingming Zhang. 2019. Practical program repair via bytecode mutation. In *ISSTA*. 19–30.
[7] 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 *ISSTA*. to appear.
[8] Ali Ghanbari and Andrian Marcus. 2022. Shibboleth: Hybrid Patch Correctness Assessmentin Automated Program Repair. https://github.com/ali-ghanbari/shibboleth-demo Accessed: 05/22.
[9] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated Program Repair. *CACM* 62 (2019), 56–65.
[10] Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *POPL*. 298–312.
[11] Martin Monperrus. 2018. *The Living Review on Automated Program Repair.* Technical Report hal-01956501. HAL/archives-ouvertes.fr.
[12] Ripon K. Saha, Yingjun Lyu, Hiroaki Yoshida, and Mukul R. Prasad. 2017. Elixir: Effective object-oriented program repair. In *ASE*. 648–659.
[13] scikit-learn Contributors. 2020. `scikit-learn`: Machine Learning in Python. http://bit.ly/3a70cZt Accessed: 05/22.
[14] 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.
[15] Shin Hwei Tan, Hiroaki Yoshida, Mukul R. Prasad, and Abhik Roychoudhury. 2016. Anti-patterns in search-based program repair. In *FSE*. 727–738.
[16] 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.
[17] 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.
[18] W. Eric Wong, Ruizhi Gao, Yihao Li, Rui Abreu, and Franz Wotawa. 2016. A survey on software fault localization. *TSE* 42 (2016), 707–740.
[19] Yingfei Xiong, Xinyuan Liu, Muhan Zeng, Lu Zhang, and Gang Huang. 2018. Identifying patch correctness in test-based program repair. In *ICSE*. 789–799.
[20] 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.
[21] 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.
[22] Yuan Yuan and Wolfgang Banzhaf. 2020. Toward Better Evolutionary Program Repair: An Integrated Approach. *TOSEM* 29 (2020), 1–53.