Assessing Program Repair In The Wild

Alex Barajas & Shreyes Joshi

Background

Automated Program Repair

- Techniques built to automatically identify and repair bugs in code
- Successful automated program repair can simplify the lives of developers
- Save companies time and money

Problem

Current benchmark datasets for testing APR techniques rely too heavily on future test cases

- Future test cases: Test cases produced after a bug fix
- Made to confirm the bug is fixed

```
1  // In class MathArrays
2  public static double linearCombination(double[] a, double[]
       b) throws ... {
3       ...
4  + if (len == 1) {
5  +   // Revert to scalar multiplication
6  +   return a[0] * b[0];
7  + }
8    ...
9  }
```

```
1  public void testArray() {
2      final double [] a = { 1.23456789 } ;
3      final double [] b = { 98765432.1 } ;
4      Assert.assertEquals(a[0] * b[0], MathArrays.
           linearCombination(a, b), 0d) ;
5  }
```

Problem

Future test cases can overly assist APR techniques

Too specific to a bug

Leads to overly easy fixes or overfitting

Does not translate to the real world

Fewer bugs are going to be heavily related to existing issues

APR testing is inflated

Accurate testing will do more to advance APR techniques in the future

TABLE I
FUTURE TEST CASES IN EXISTING BENCHMARK DATASETS

Benchmark	Total Bugs	Bugs with future test cases
Defects4J	395	381
Bugs.jar	1130	1064
Bears	251	232

Motivation

Introduce a dataset with better testing methods

Only include existing test cases

- Existing test cases: regression test cases that exist before any specific bugs are fixed

```
// In class ConnectionProxy
private final boolean closeOpenStatements() {
  final int size = openStatements.size();
- if (size <= 0) {
+ if (size > 0) {
    boolean success = true;
    for (int i = 0; i < size; i++) {
    ...
}
```

```
public void testAutoStatementClose() throws SQLException {
  Connection connection = ds.getConnection();
  Assert.assertNotNull(connection);
  Statement statement1 = connection.createStatement();
  Assert.assertNotNull(statement1);
  Statement statement2 = connection.createStatement();
  Assert.assertNotNull(statement2);
  connection.close();
  Assert.assertTrue(statement1.isClosed());
  Assert.assertTrue(statement2.isClosed());
}
```

Motivation

Both examples shown are fault revealing test cases

One is incredibly specific to the test case

One other is a naturally occurring regression test

Existing test cases are much more

```
1  public void testArray() {
2      final double [] a = { 1.23456789 } ;
3      final double [] b = { 98765432.1 } ;
4      Assert.assertEquals(a[0] * b[0], MathArrays.
           linearCombination(a, b), 0d) ;
5  }
```

```
1  public void testAutoStatementClose() throws SQLException {
2    Connection connection = ds.getConnection();
3    Assert.assertNotNull(connection);
4    Statement statement1 = connection.createStatement();
5    Assert.assertNotNull(statement1);
6    Statement statement2 = connection.createStatement();
7    Assert.assertNotNull(statement2);
8    connection.close();
9    Assert.assertTrue(statement1.isClosed());
10   Assert.assertTrue(statement2.isClosed());
11 }
```

Research Questions

How different are failure-exposing test cases in our dataset than those included in Defects4J ?

Are there differences in automated program repair effectiveness evaluated using our dataset as compared to Defects4J ?

Paper Solution

Create a benchmark with the following properties

- Regression Bugs from CI
  - No prior knowledge on specific bug fixes is known
- Diversity
  - Represents real world issues in different contexts
  - Categorizable bugs

Paper Solution

- Find GitHub projects that use Travis CI
- Obtain build information from TravisTorrent
  - 316 projects found
- Filter projects by number of commits and framework
  - 91 projects with an average of 2,791 builds
- Find builds that failed due to failed test cases
- Find builds with patches that are recreatable by APR
  - Source code files only (no XML or package files)
  - Change comes from the same file
- 399 patches from 67 projects

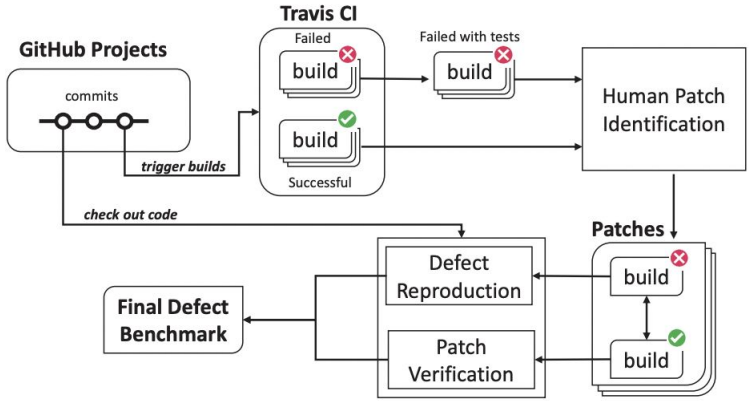


Fig. 5. The process of benchmark construction.

Paper Solution

- Manually try to reproduce the failed build
- Expected failed test cases should be the same
- Reproduce the patch
- Test cases should pass
- Result is a benchmark dataset
- 102 bugs on 40 projects

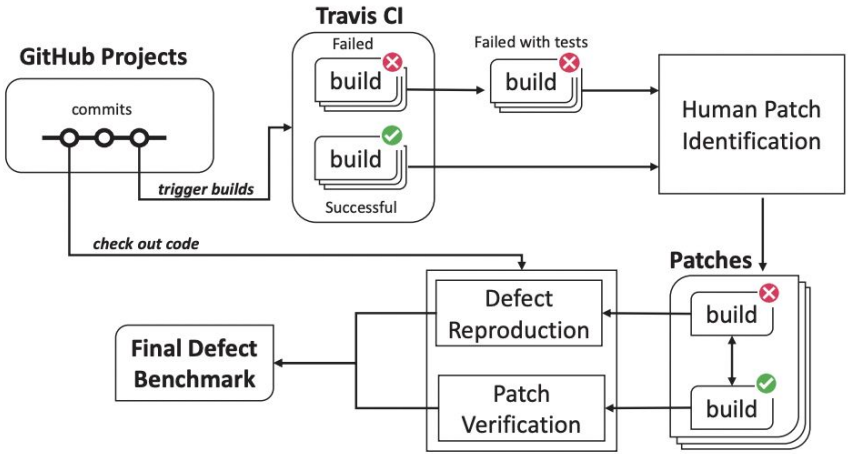


Fig. 5. The process of benchmark construction.

Evaluation

RQ1: How different are failure-exposing test cases in our dataset than those included in Defects4J?

- Methodology
  - S1, S2, S3, S4
  - Quantity, Localizability, and Similarity

Datasets

- Deflects4J, Author's Dataset

ID	Description
Notations	
RC	Number of unique classes that have to be repaired and explicitly called by failure-exposing test cases
C	Number of unique classes explicitly called by failure test cases
RM	Number of unique methods that have to be repaired and explicitly called by failure-exposing test cases
M	Number of unique methods explicitly called by failure test cases
Statistics	
S1	Average number of failure-exposing test cases
S2	Average of $RC \,/\, C$
S3	Average of $RM \,/\, M$
S4	Average number of shared identifiers between failure-exposing test cases and patches

Evaluation cont. Author V.S Defects4J

- S1
  - The average number of failure-exposing test cases per bug is significantly higher in the new dataset (15.39) compared to Defects4J (2.37).
- S2 / S3
  - The results show that the failure-exposing test cases in the new dataset are less localizable compared to those in Defects4J.
- S4
  - The similarity between failure-exposing test cases and bug fix patches is also significantly lower in the new dataset (0.426) compared to Defects4J (0.604).

> Failure-exposing test cases per bug in our dataset are more than six times larger than those for Defects4J; they are also three times less localizable and close to 70% less similar to bug fix patches.

> Overall, the results indicate a lower patch rate on our dataset as compared to Defects4J, suggesting that it is more difficult for APR to generate correct patches on our dataset.

Evaluation cont. APR Effectiveness

RQ1: Are there differences in automated program repair effectiveness evaluated using our dataset as compared to Defects4J?

Methodology

- APR Tools
  - GenProg, Kali, Cardumen, Arja, RsRepair
- Correct Patch Rate metric

Author's Dataset (102 Bugs)

- Mean of 2.16%

Defects4J

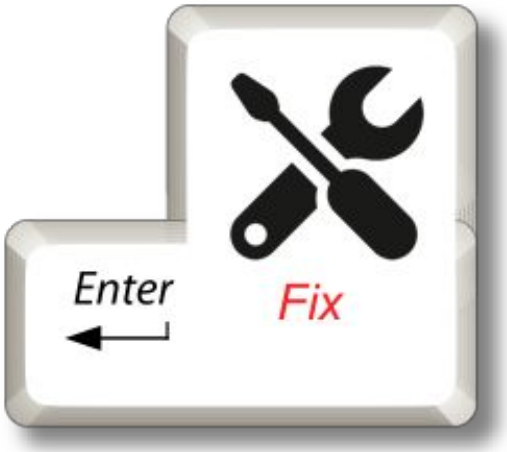
- Mean of 16.9%

$$\text{Correct Patch Rate} = \frac{\#CP}{\#Tools \times \#Bugs}$$

- **#CP (Correct Patches)**: Total number of correct patches generated by all APR tools.
- **#Tools**: Number of APR tools used.
- **#Bugs**: Number of bugs evaluated.

Paper's novelty

- Unique Dataset - New benchmark!
  - Focused on CI bugs
- Evaluation Metrics
- Statistical significance
- Impact of APR Development



Potential Limitation

- Bias in Dataset Construction
- Range of APR Tools used
- Complexity and Diversity of Bugs

Conclusion

- Realilist Dataset
- APR Effectiveness
- Empirical Validation
- Call for Further Development

Discussion Points

1. With the understanding and evidence the paper proposed should we focus on CI bugs for APR applications? Is the Future test issue present and applicable in other bug research areas?
2. Would semantics-based APR techniques perform different on the Author's Dataset? Are they dependent on these future test?
3. While datasets like Defects4j place too much emphasis on future test cases, is there not still some value? Would it be possible to create a dataset mixed with both kinds of tests?
4. Current APR techniques cannot functionally repair bugs in this dataset. Is this a sign that APR techniques are not usable? Is this dataset far too challenging for the current state of techniques?

Questions?