

Coverage Isn't Enough: SBFL-Driven Insights into Manually Created vs. Automatically Generated Tests

Sasara Shimizu, Yoshiki Higo
The University of Osaka, Japan



Background

- Unit testing is essential for early bug detection
- Manually creating unit tests is time-consuming and challenging
- Automated test generation tools have been developed

MC-Tests = Manually Created Test

AG-Tests = Automatically Generated Test

- Few studies directly compare AG-Tests and MC-Tests

SBFL (Spectrum-Based Fault Localization)

- Technique to estimate defect locations in a given program
- Uses execution path information recorded during test execution
- Statements executed by failed tests are regarded as suspicious

Research Purpose and Methodology

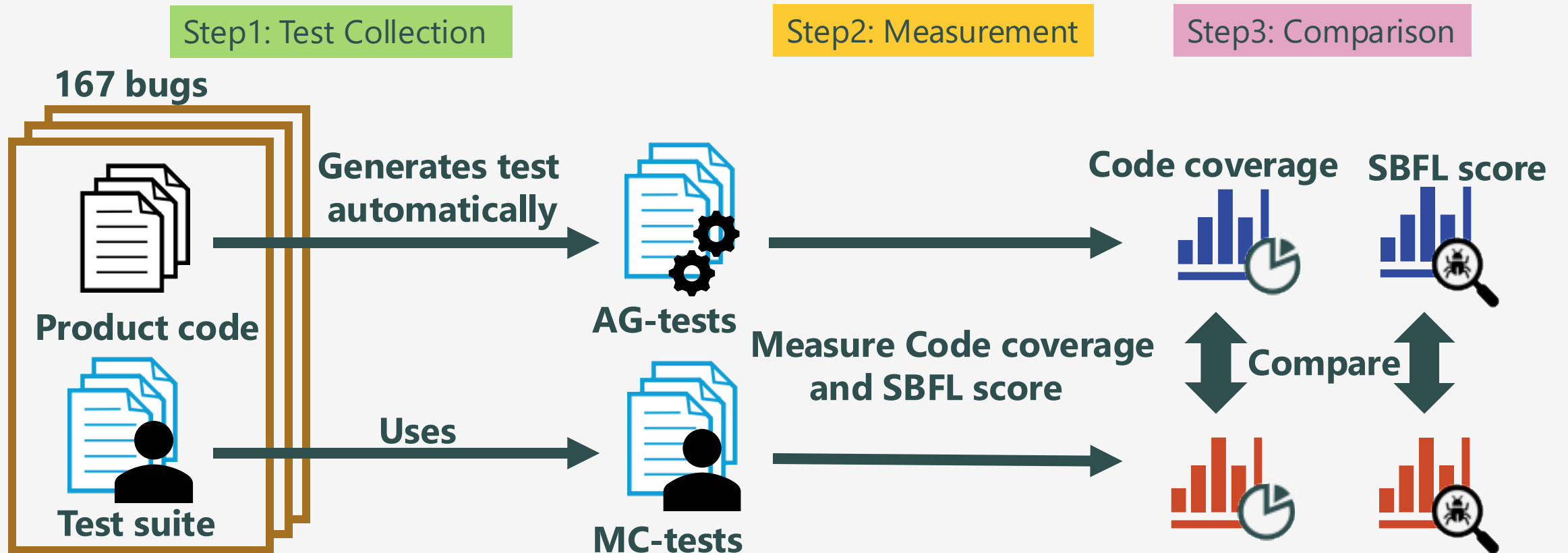
Purpose

Reveal the characteristics of MC-tests and AG-tests

Methodology

Compare them based on their code coverage and SBFL score

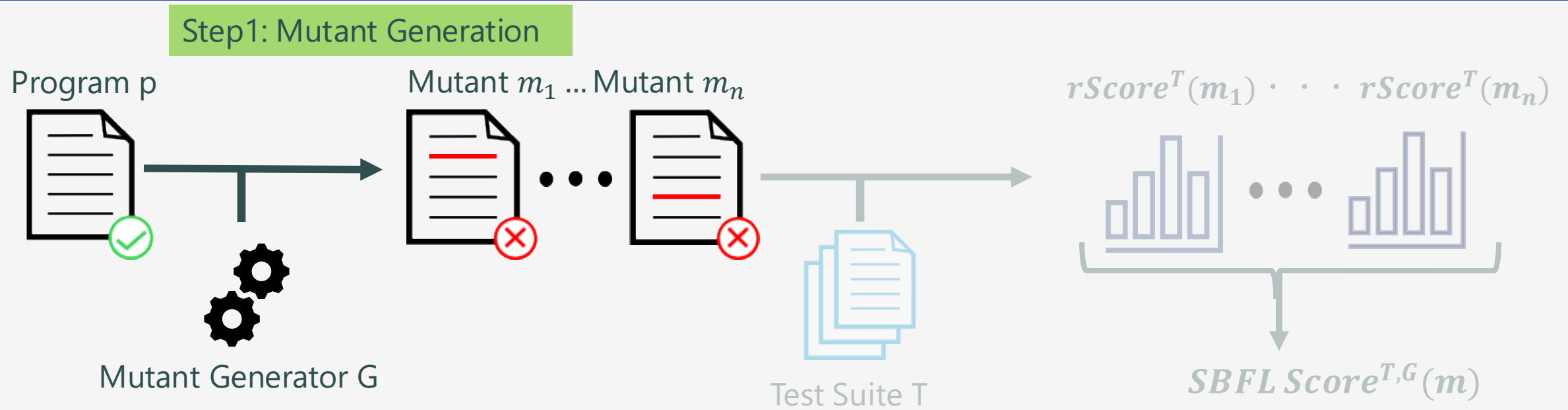
Experimental Flow



SBFL score [13]

- A measure of program suitability for SBFL
- The value ranges from 0 to 1
- A higher value indicates that SBFL works well for a given program

How to measure SBFL score 1/3

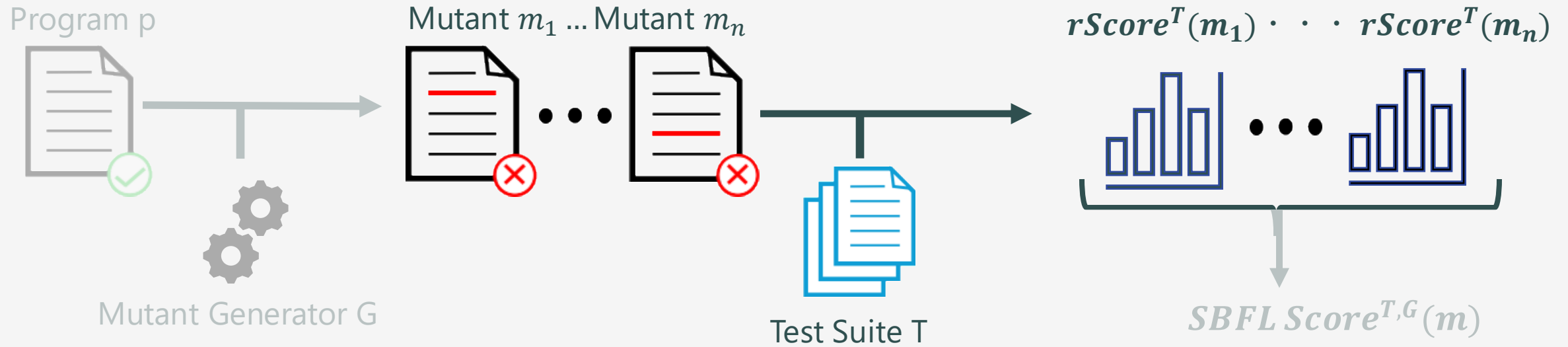


Step1: Mutant Generation

- Only a single change is introduced in each mutant
- Generate as many mutants as possible

How to measure SBFL score 2/3


Step2: SBFL Execution



Step2: Executing SBFL on each mutant

- Calculate suspiciousness values using the test suite
- Rank the statements in descending order of their suspiciousness values
- Normalize the resulting ranks to obtain the r-Score

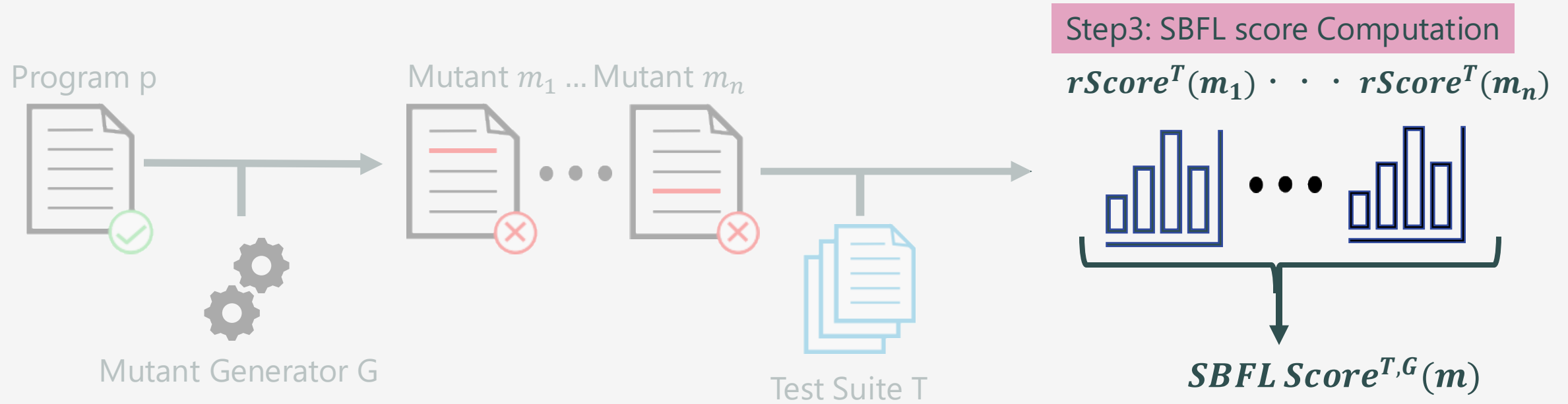
Example of SBFL

	String FizzBuzz(int number) {	suspiciousness values	rank	r-Score
S_1	if (number % 15 == 0) {	0.577	4	0.5
S_2	return "FizzBuzz";	0.0	7	0.0
S_3	} else if (number % 3 == 0) {	0.707	3	0.333
S_4	return "Fizz";	0.0	7	0.0
S_5	 } else if (number % 4 == 0) {	1.0	2	0.833
S_6	return "Buzz";	1.0	2	0.833
	}			
S_7	return String.valueOf(number);	0.0	7	0.0
	}			

Step2: Executing SBFL on each mutant

- Calculate suspiciousness values using the test suite
- Rank the statements in descending order of their suspiciousness values
- Normalize the resulting ranks to obtain the r-Score

How to measure SBFL score 3/3



Step3: SBFL score Computation

- Calculate the average of r-Scores from all the mutants.

Research Questions

RQ1:

Which tests achieve higher **Code Coverage** between AG-tests and MC-tests?

Answer: AG-Tests achieve higher **Branch Coverage**

RQ2:

Which tests achieve higher **SBFL score** between AG-tests and MC-tests?

Experimental Setup

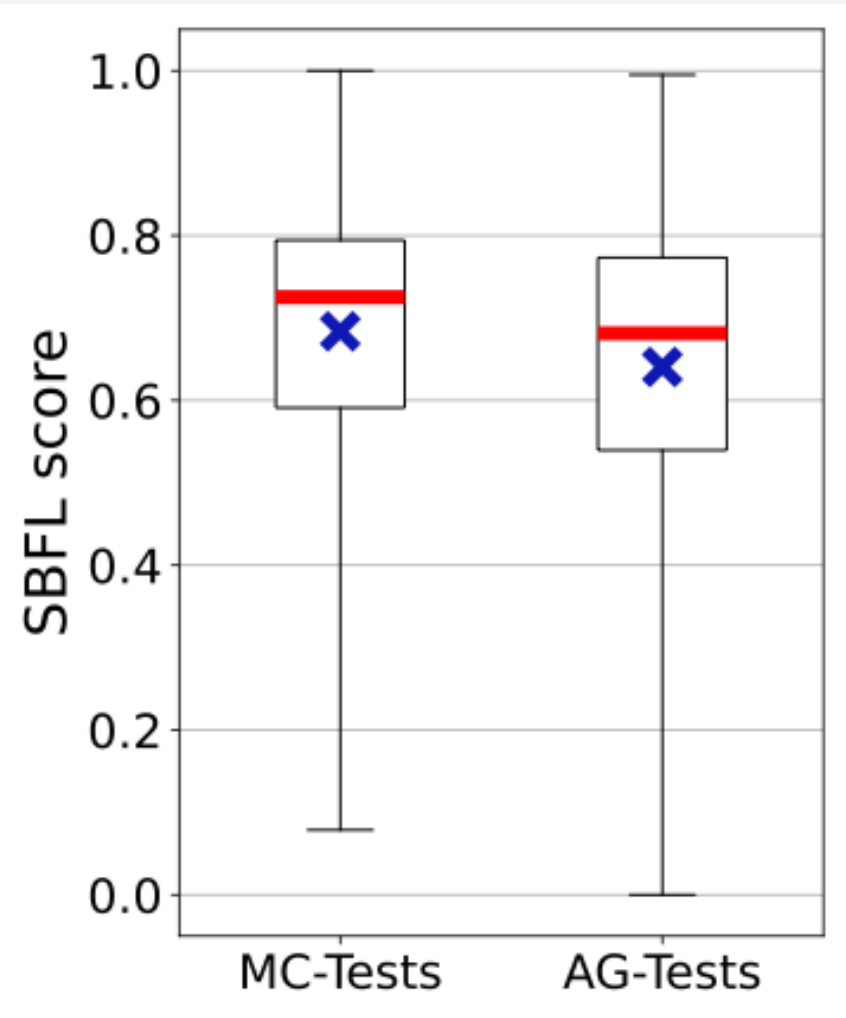
Target

- 167 bug instances in Defects4J (v1.3.0)
from Lang and Math projects
- Compare MC-tests (developer-written) and AG-tests (EvoSuite v1.2.0)

Measurement & Metrics

- Metrics Measured: Statement Coverage, Branch Coverage, and SBFL Score
- Coverage Measurement: JaCoCo
- SBFL Measurement: Gzoltar

RQ2 Results: SBFL Score



- The median (red line)
- The mean (cross mark)

Both the median and the mean scores are **lower for AG-Tests**

RQ2 Results: SBFL Score

We find a statistically significant difference in SBFL scores between MC-Tests and AG-Tests

	Two-Sided Wilcoxon	One-sided Wilcoxon for MC-tests	One-sided Wilcoxon for AG-tests
p-value	p=0.02	p=0.01	p=0.98

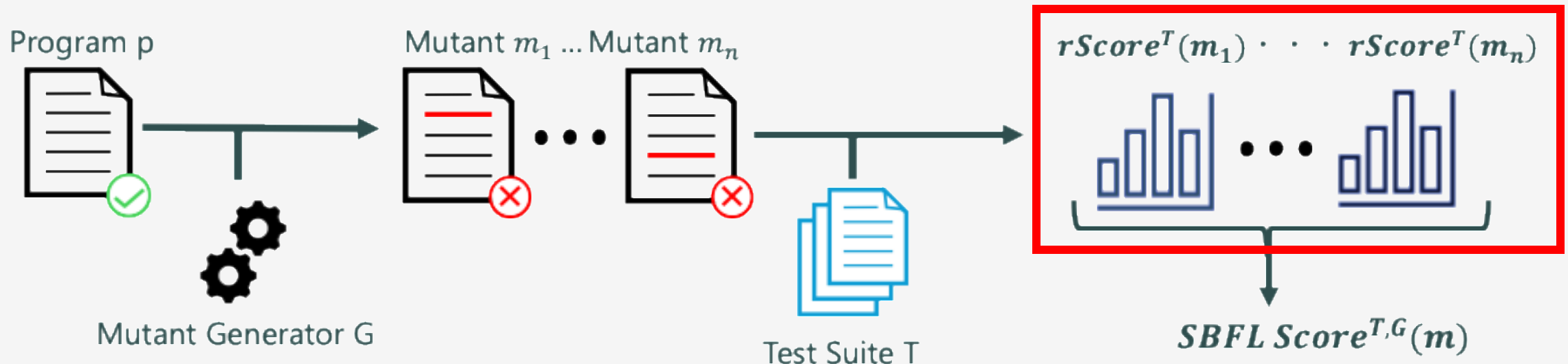
Our experimental results show that:

MC-Tests achieved **higher SBFL scores** than AG-Tests

Further Analysis of the SBFL Score

We focused on the **r-Score**

- This value is obtained by applying SBFL on each individual mutant
- It indicates how accurately the bug location was identified



Further Analysis of the SBFL Score: Key Perspective

We focused on two key perspectives for analysis:

Mutation Operator

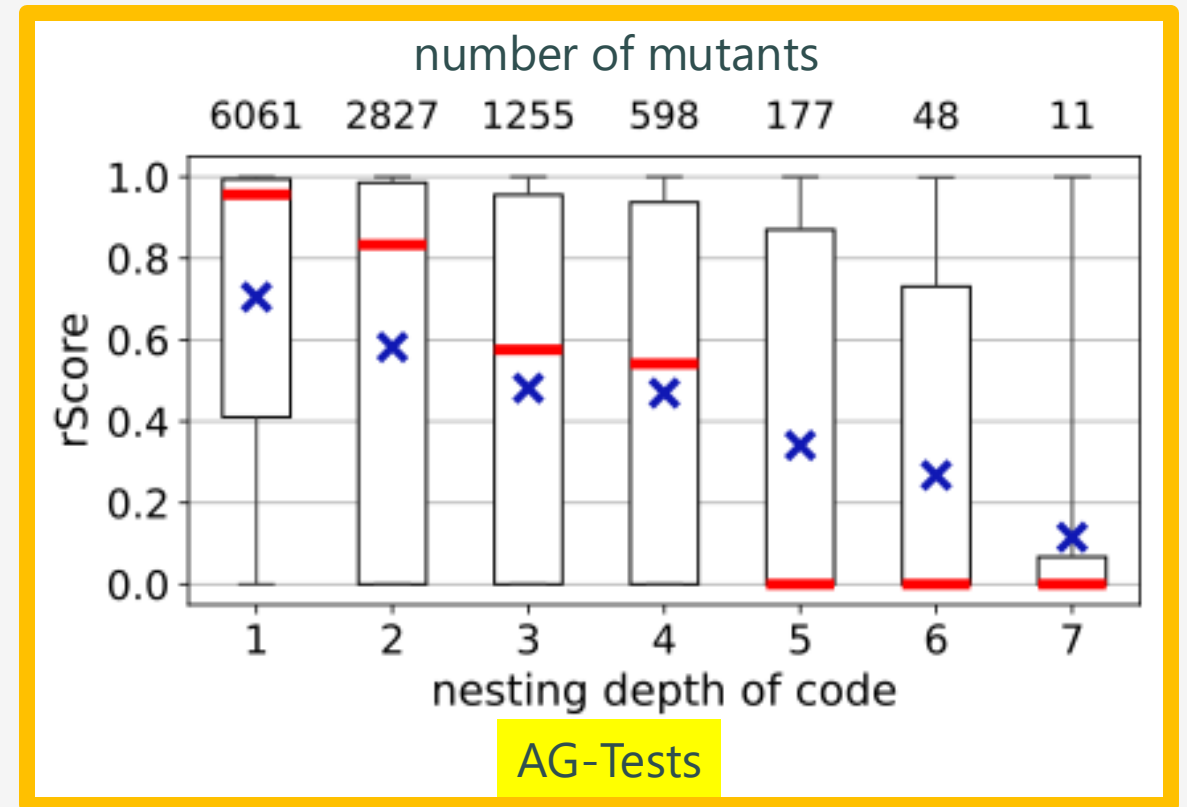
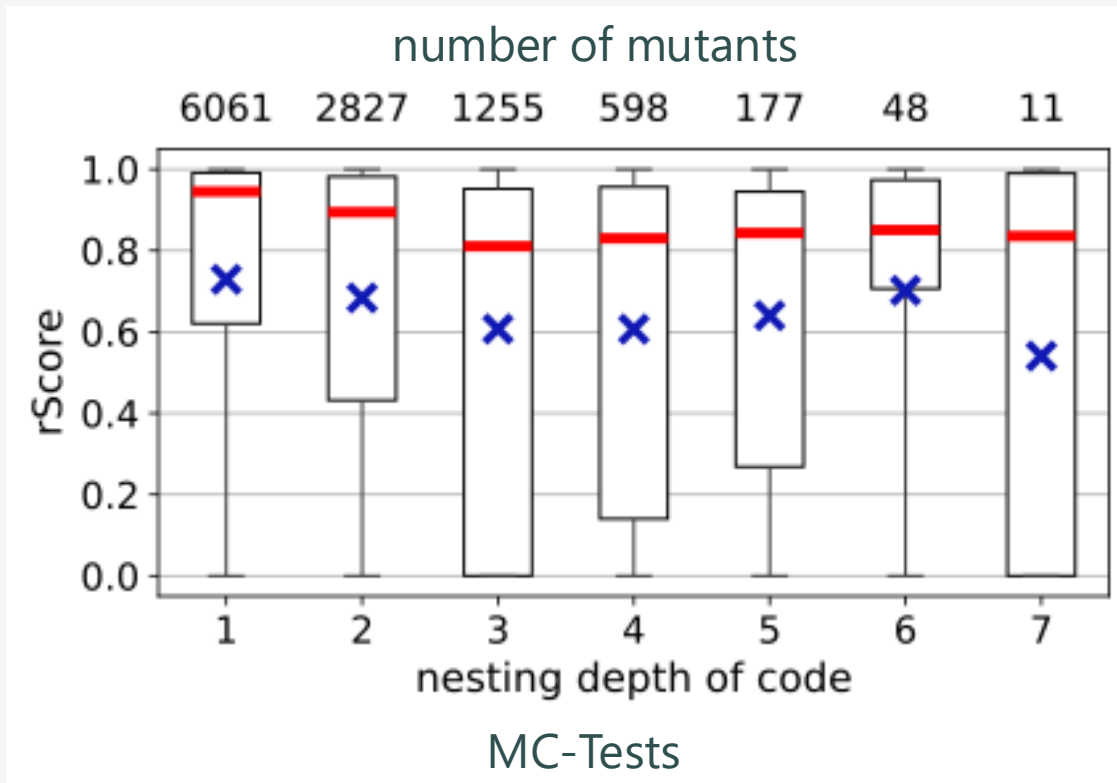
The type of change introduced to the original source code

No trend was observed across different mutation operators

Nesting Depth

The depth of code nesting where the bug (in the mutant) is located

Further Analysis of the SBFL Score: Nesting Depth



The median r-Score of AG-tests decreases as the depth increases

Discussion

- AG-tests excel in terms of branch coverage
- MC-tests excel in terms of SBFL score
- AG-tests' SBFL is ineffective for deeply nested structures

Hybrid Approach:

- Use **AG-tests** to quickly achieve high code coverage
- Use **MC-tests** to find bugs effectively in complex parts of the code

Summary

Conclusion

We compare MC-tests and AG-tests on code coverage and SBFL score

- AG-tests perform better than MC-Tests in branch coverage
- But their utility for fault localization declines severely in complex code
- An effective testing strategy should combine both tests

Future Work

- Use other automated test generation tools
- Explore intelligent test generation frameworks that statically analyze code structure and recommend an optimal mix of the two tests

Appendix

Code Coverage

We use **Statement Coverage** and **Branch Coverage**

Statement Coverage

$$= \frac{\text{Number of lines executed by the test suite}}{\text{Number of executable lines in the target code}}$$

Branch Coverage

$$= \frac{\text{Number of branches executed by the test suite}}{\text{Number of branches in the target code}}$$

Ochiai

The Ochiai formula

Suspiciousness value of Statement $s = \frac{fail(s)}{\sqrt{totalFail \times (fail(s) + pass(s))}}$

- $fail(s)$: the number of failed test cases that executed statement s
- $pass(s)$: the number of successful test cases that executed statement s
- $totalFail$: the total number of failed test cases

Further Analysis of the SBFL Score: Nesting Depth

We calculated the **effect size r** for each nesting depth

- It indicates the magnitude of the difference between AG/MC-Tests
- Larger absolute values mean greater difference

Nesting depth	1	2	3	4	5	6	7
effect size r	-0.095	0.12	0.21	0.28	0.55	0.67	0.65

The difference in r-Score tends to **increase** with greater nesting depth

AG-tests become **less effective** at accurately locating bugs in **more deeply nested** code structures

Example of Mutation Operator

Table 1: Mutation operators

Mutation operators	Original conditional	Mutated conditional
Conditionals Boundary	<code>a<b</code>	<code>a<=b</code>
Increments	<code>n++</code>	<code>n--</code>
Invert Negatives	<code>-n</code>	<code>n</code>
Math	<code>a+b</code>	<code>a-b</code>
Negate Conditionals	<code>a==b</code>	<code>a!=b</code>
Void Method Calls	<code>method();</code>	<code>;</code>
Primitive Returns	<code>return 5;</code>	<code>return 0;</code>

Evosuite

- An automated tool for Java unit test generation
- Widely used in software research
- Uses exploratory approaches to find effective tests