Measuring and Mitigating Gaps in Structural Testing

Soneya Binta Hossain, Matthew Dwyer, Sebastian Elbaum, Anh Nguyen-Tuong

Apache Commons CSV

README.md

Apache Commons CSV

Java CI passing | codecov 98% | maven central 1.10.0 | javadoc 1.10.0 | CodeQL passing | openssf scorecard 8.6

The Apache Commons CSV library provides a simple interface for reading and writing CSV files of various types.

Documentation

More information can be found on the Apache Commons CSV homepage. The Javadoc can be browsed. Questions related to the usage of Apache Commons CSV should be posted to the [user mailing list][ml].

Where can I get the latest release?

Used by 29.4k

+ 29,404

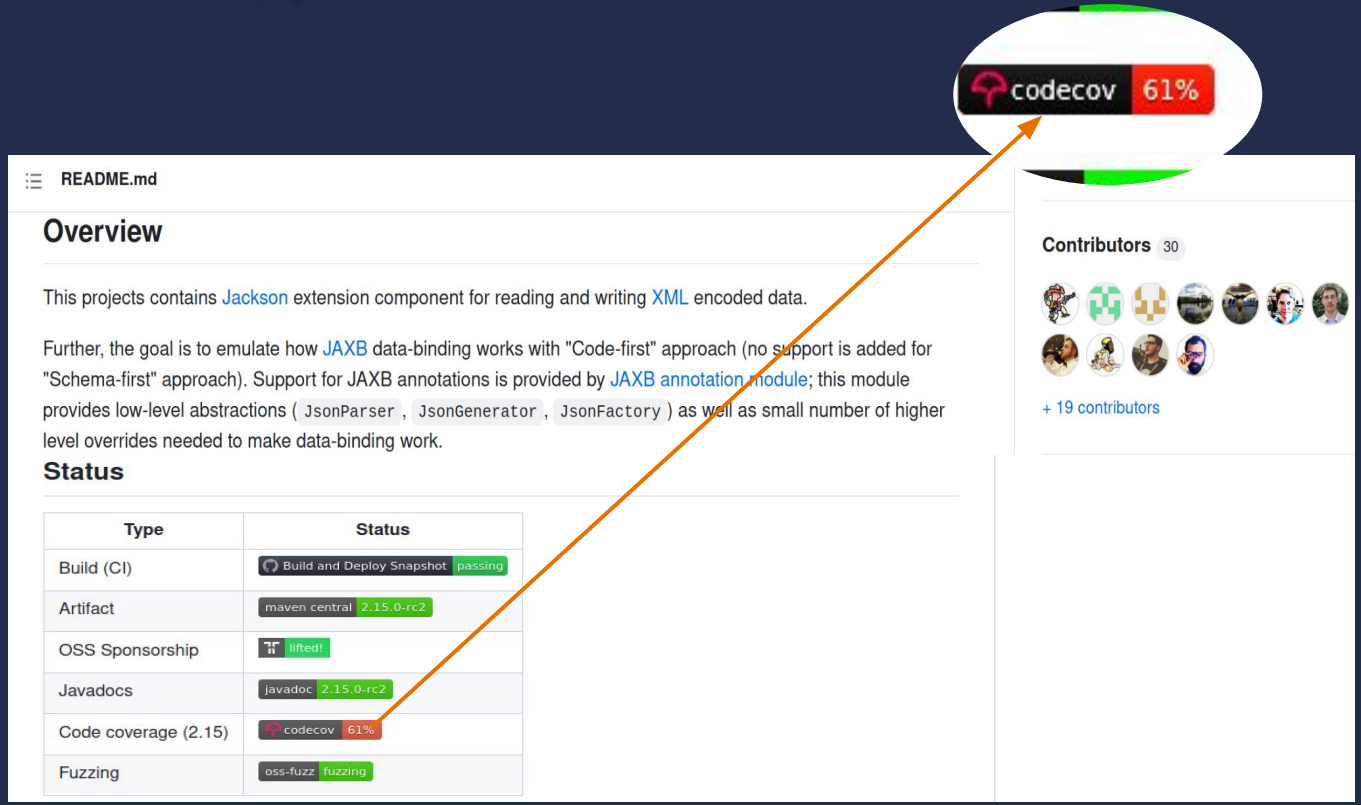
Contributors 38

+ 27 contributors

Languages

Java 99.8% Shell 0.2%

FasterXML/jackson-dataformat-xml



Based on statement coverage CSV seems to be better tested than jackson-dataformat.

Would we say the same thing if we considered the quality of the test oracles in addition to coverage?

Code Coverage, Test Oracle and Fault-detection

- Code coverage is *essential* but *insufficient*

Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems

Pavneet Singh Kochhar, Ferdian Thung, and David Lo
Singapore Management University, Singapore
{kochharps.2012, ferdiant.2013, davidlo}@smu.edu.sg

Using Simulation for Assessing the Real Impact of Test Coverage on Defect Coverage

Lionel Briand, Dietmar Pfahl
Fraunhofer Institute for Experimental Software Engineering (IES)
Sauerwiesen 6, D-67661 Kaiserslautern, Germany
{briand, pfahl}@iese.fhg.de

The Effect of Code Coverage on Fault Detection under Different Testing Profiles

Xia Cai
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
xcai@cse.cuhk.edu.hk

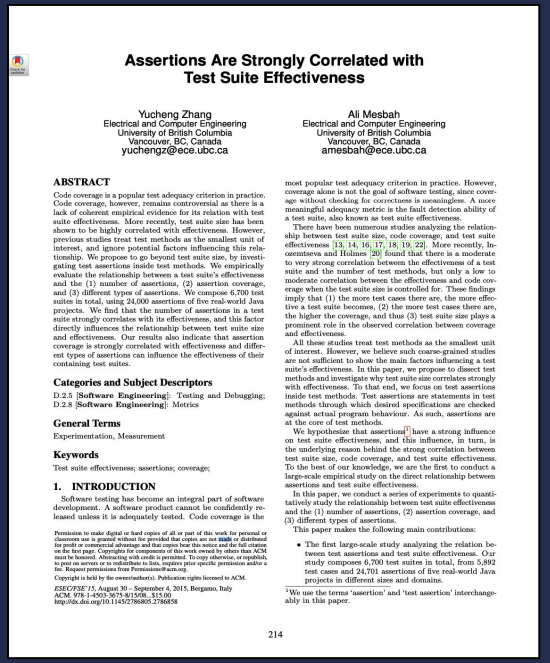
Michael R. Lyu
Dept. of Computer Science and Engineering
The Chinese University of Hong Kong
lyu@cse.cuhk.edu.hk

Coverage Is Not Strongly Correlated with Test Suite Effectiveness

Laura Inozemtseva and Reid Holmes
School of Computer Science
University of Waterloo
Waterloo, ON, Canada
{lminozem,rtholmes}@uwaterloo.ca

Code Coverage, Test Oracle and Fault-detection

- Code coverage is *essential* but *insufficient*
- Test oracles and fault-detection are *strongly correlated*



Assertions Are Strongly Correlated with Test Suite Effectiveness

Yucheng Zhang
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
yuchengz@ece.ubc.ca

Ali Mesbah
Electrical and Computer Engineering
University of British Columbia
Vancouver, BC, Canada
amesbah@ece.ubc.ca

ABSTRACT

Code coverage is a popular test adequacy criterion in practice. Code coverage, however, remains controversial as there is a lack of coherent empirical evidence for its relation with test suite effectiveness. More recently, test suite size has been shown to be highly correlated with effectiveness. However, previous studies treat test methods as the smallest unit of interest, and ignore potential factors influencing this relationship. We propose to go beyond test suite size, by investigating test assertions inside test methods. We empirically evaluate the relationship between a test suite's effectiveness and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions. We compose 6,700 test suites in total, using 24,000 assertions of five real-world Java projects. We find that the number of assertions in a test suite strongly correlates with its effectiveness, and this factor directly influences the relationship between test suite size and effectiveness. Our results also indicate that assertion coverage is strongly correlated with effectiveness and different types of assertions can influence the effectiveness of their containing test suites.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.8 [**Software Engineering**]: Metrics

General Terms

Experimentation, Measurement

Keywords

Test suite effectiveness; assertions; coverage;

1. INTRODUCTION

Software testing has become an integral part of software development. A software product cannot be confidently released unless it is adequately tested. Code coverage is the most popular test adequacy criterion in practice. However, coverage alone is not the goal of software testing, since coverage without checking for correctness is meaningless. A more meaningful adequacy metric is the fault detection ability of a test suite, also known as test suite effectiveness.

There have been numerous studies analyzing the relationship between test suite size, code coverage, and test suite effectiveness [13, 14, 16, 17, 18, 19, 22]. More recently, Inozemtseva and Holmes [20] found that there is a moderate to very strong correlation between the effectiveness of a test suite and the number of test methods, but only a low to moderate correlation between the effectiveness and code coverage when the test suite size is controlled for. These findings imply that (1) the more test cases there are, the more effective a test suite becomes, (2) the more test cases there are, the higher the coverage, and thus (3) test suite size plays a prominent role in the observed correlation between coverage and effectiveness.

All these studies treat test methods as the smallest unit of interest. However, we believe such coarse-grained studies are not sufficient to show the main factors influencing a test suite's effectiveness. In this paper, we propose to dissect test methods and investigate why test suite size correlates strongly with effectiveness. To that end, we focus on test assertions inside test methods. Test assertions are statements in test methods through which desired specifications are checked against actual program behaviour. As such, assertions are at the core of test methods.

We hypothesize that assertions[1] have a strong influence on test suite effectiveness, and this influence, in turn, is the underlying reason behind the strong correlation between test suite size, code coverage, and test suite effectiveness. To the best of our knowledge, we are the first to conduct a large-scale empirical study on the direct relationship between assertions and test suite effectiveness.

In this paper, we conduct a series of experiments to quantitatively study the relationship between test suite effectiveness and the (1) number of assertions, (2) assertion coverage, and (3) different types of assertions.

This paper makes the following main contributions:

- The first large-scale study analyzing the relation between test assertions and test suite effectiveness. Our study composes 6,700 test suites in total, from 5,892 test cases and 24,701 assertions of five real-world Java projects in different sizes and domains.

[1]We use the terms 'assertion' and 'test assertion' interchangeably in this paper.

ESEC/FSE'15, August 30 – September 4, 2015, Bergamo, Italy
ACM. 978-1-4503-3675-8/15/08...$15.00
http://dx.doi.org/10.1145/2786805.2786858

214

Coverage Based on Test Oracles

- Considers program execution and test oracles
  - Support statement criterion
  - Only *evaluate* test suites
- We build on Checked Coverage by Schuler and Zeller
  - We support stronger criterion
  - We introduce and study the concept of *Coverage Gap*

State Coverage: Software Validation Metrics beyond Code Coverage

Dries Vanoverberghe[1,*], Jonathan de Halleux[2], Nikolai Tillmann[2], and Frank Piessens[1]

[1] Katholieke Universiteit Leuven, Leuven, Belgium
(dries.vanoverberghe,frank.piessens)@cs.kuleuven.be

2011 Fourth IEEE International Conference on Software Testing, Verification and Validation

Assessing Oracle Quality with Checked Coverage

David Schuler
Saarland University – Computer Science
Saarbrücken, Germany
ds@cs.uni-saarland.de

Andreas Zeller
Saarland University – Computer Science
Saarbrücken, Germany
zeller@cs.uni-saarland.de

State Coverage: A Structural Test Adequacy Criterion for Behavior Checking

Ken Koster
kenk@agitar.com

David Kao
dvk@agitar.com

Agitar Software Laboratories
450 National Avenue
Mountain View, California 94043

ABSTRACT

We propose a new language-independent, structural test adequacy criterion called state coverage. State coverage measures whether unit-level tests check the outputs and side effects of a program.

State coverage differs in several respects from existing test adequacy criteria, such as code coverage and mutation adequacy. Unlike other coverage-based criteria, state coverage measures the extent of checks of program behavior. And unlike existing fault-based criteria such as mutation adequacy, state coverage has been designed to be readily automated and to present users with easily understood test inadequacy reports.

An experiment showed strong positive correlations between the number of behavior checks and both state coverage and mutation adequacy.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging–*testing tools*; D.2.8 [Software Engineering]:Metrics

General Terms: Measurement, Reliability

Keywords: Coverage, fault-based, mutation testing, state coverage, structural testing, test adequacy criteria, unit testing

1. INTRODUCTION

How thoroughly tested is a program? This is the main question that test adequacy criteria attempt to answer. Each criterion answers this question differently according to how it measures program "size" and how much of that size is evaluated by a test. For instance, branch coverage measures a program by the number of branches it contains and informs users which of those branches were not taken during test execution. *State coverage*, the test adequacy criterion we propose in this paper, evaluates tests according to behavior checking; it measures a program by the number of statements that define output and side effect variables and judges tests according to whether they check those variables.

The example JUnit[1] test below, `testAbeWithoutCheck` does not check program behavior automatically. Human intervention is required to examine the output of the test to verify the correctness of the behavior of the program. The implementation of `abs` could be replaced by any compiling

```
int abs(int n) {
  if (n < 0)
    return n * -1;
  else
    return n;
}

public void testAbeWithoutChe
  int x = abs(-4);
  System.out.println("abs(-4)
}
```

Copyright is held by the author/owner(s).
ESEC/FSE'07, September 3-7, 2007, Cavtat near Dubrovnik, Croatia.

Checked coverage: an indicator for oracle quality

David Schuler*,† and Andreas Zeller

Department of Computer Science, Saarland University, Saarbrücken, Germany

SUMMARY

A known problem of traditional coverage metrics is that they do not assess *oracle quality*—that is, whether the computation result is actually checked against expectations. In this paper, we introduce the concept of *checked coverage*—the dynamic slice of covered statements that actually influence an oracle. Our experiments on seven open-source projects show that checked coverage is a sure indicator for oracle quality and even more sensitive than mutation testing. Copyright © 2013 John Wiley & Sons, Ltd.

Received 11 September 2011; Revised 22 November 2012; Accepted 13 March 2013

KEY WORDS: test suite quality; coverage metrics; dynamic slicing; mutation testing

1. INTRODUCTION

How can I ensure that my programme is well-tested? The most widespread metric to assess test quality is *coverage*. Test coverage measures the percentage of code features such as statements or branches that are executed during a test. The rationale is that the higher the coverage, the higher the chances of catching a code feature that causes a failure—a rationale that is easy to explain, but which relies on an important assumption. This assumption is that we actually are able to *detect* the failure. It does not suffice to cover the error, we also need a means to detect it.

As it comes to detecting arbitrary errors, it does not make a difference whether an error is detected by the test (e.g. a mismatch between actual and expected result), by the programme itself (e.g. a failing assertion) or by the run-time system (e.g. a dereferenced null pointer). To validate computation results, however, we need checks in the test code and in the programme code—checks henceforth summarized as *oracles*. A high coverage does not tell anything about oracle quality. It is perfectly possible to achieve a 100% coverage and still not have any result checked by even a single oracle. The fact that the run-time system did not discover any errors indicates robustness, but does not tell anything about functional properties.

As an example of such a mismatch between coverage and oracle quality, consider the test for the `PatternParser` class from the JAXEN XPATH library, shown in Figure 1. This test case passes a set of paths through the parser. Interestingly enough, it never checks any of the parser results. The parser may return complete nonsense, and this test would never notice it.

Why are the parser results unchecked? Actually, the purpose of this test is not to test the parser, but the individual paths in the `paths` array; if any of these contain a syntactical error, the parser would throw an exception. If the test passes, all paths are fine. Running this test, however, also achieves a statement coverage of 83% in the parser, and one may thus assume the parser is well-tested.

Focus of Our Paper

- Measuring the gap between code that is executed and code that is checked by test oracles – we call this the coverage gap
- Evaluating the impact of the coverage gap on fault-detection
- Mitigating coverage gaps by enhancing test suites to achieve better fault detection

Measuring Gaps

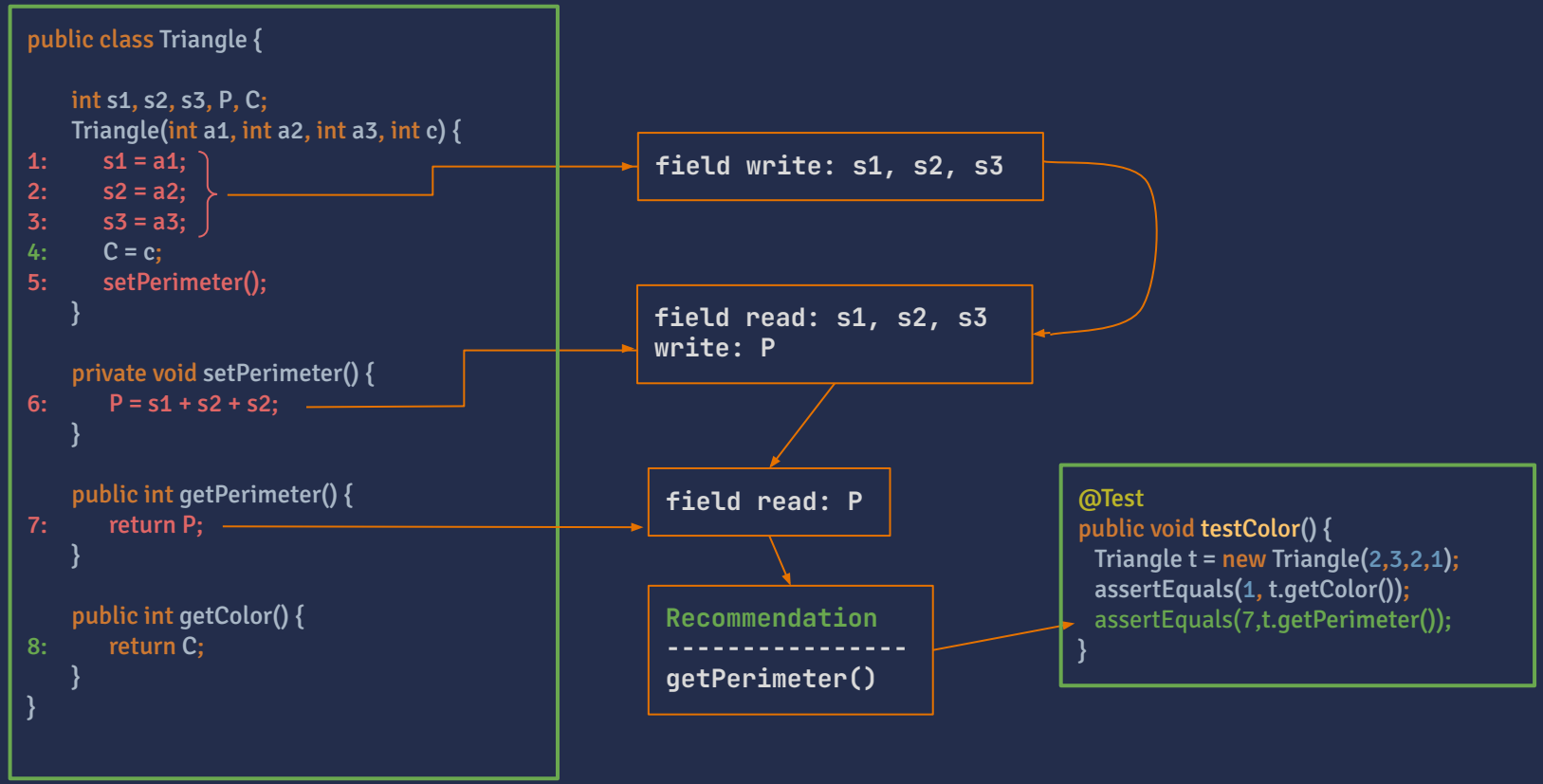
	E	E∩C	G
1:	✓		✓
2:	✓		✓
3:	✓		✓
4:	✓	✓	
5:	✓		✓
6:	✓		✓
7:	✓		✓
8:	✓	✓	

```
public class Triangle {

    int s1, s2, s3, P, C;
    Triangle(int a1, int a2, int a3, int c) {
1:      s1 = a1;
2:      s2 = a2;
3:      s3 = a3;
4:      C = c;
5:      setPerimeter();
    }

    private void setPerimeter() {
6:      P = s1 + s2 + s2;
    }

    public int getPerimeter() {
7:      return P;
    }

    public int getColor() {
8:      return C;
    }
}
```

```
@Test
public void testColor() {
  Triangle t = new Triangle(2,3,2,1);
  t.getPerimeter();
  assertEquals(1, t.getColor());
}
```

Covered: 100%
Checked: 25%
In Gap: 75%

Mitigating Gaps With Recommender



Evaluation: Artifacts

- 13 Java Applications
- 16K tests
- 51.6K test assertions

TABLE I
DESCRIPTION OF ARTIFACTS

Artifact (version)	Description	Program Size(SLOC)[1]	Test Size(SLOC)[1]	Tests(#)[2]	Assertions(#)[2]
Commons-Cli (1.4) [14]	Command line option parsing	2,699	3,932	372	573
Commons-Codec (1.2) [15]	Common encodings	8,352	12,182	887	1,793
Commons-Csv (1.5) [16]	CSV utilities	1,615	4,467	296	934
Commons-Lang (3.6) [17]	Java helper utilities	27,265	48,172	2,908	15,424
Commons-Validator (1.6) [18]	Data validation	7,409	8,352	536	2,486
Gson (2.8.0) [22]	JSON support	7,815	13,762	1,014	1,780
Jackson-Dataformat-Xml (2.9.10) [27]	XML processing	4,945	5,728	185	556
Jaxen (1.2.0) [30]	XPath engine	10,760	8,042	716	587
JFreeChart (1.5.0) [31]	2D Charts	97,350	39,348	2,174	5,506
Joda-Time (2.10.11) [32]	Date and time library	28,811	55,849	4,238	17,973
Jsoup (1.10.1) [33]	HTML parsing	10,785	5,499	510	1,645
Plexus-Utils (3.1.0) [10]	Utility classes	18,496	6,337	304	799
XStream (1.14.15) [11]	XML serialization	21,741	25,518	1,830	1,554
	Total:	248K	237K	16K	51.6K

[1] Source lines of code (SLOC) are non-comment, non-blank lines reported by the IntelliJ statistic plugin.
[2] Tests are JUnit test cases annotated with @Test, and assertions are JUnit assertions.

Research Questions

- RQ1: Gaps in studied artifacts

  Finding: Gaps range from 19-51 percentage points (pp), with an average of 35pp

- RQ2: Impact of gaps on fault detection

- RQ3: Recommender performance

- RQ4: Recommended assertions and fault detection effectiveness

  Finding: Fault detection improved as much as 57*pp* and on avg. 13*pp*

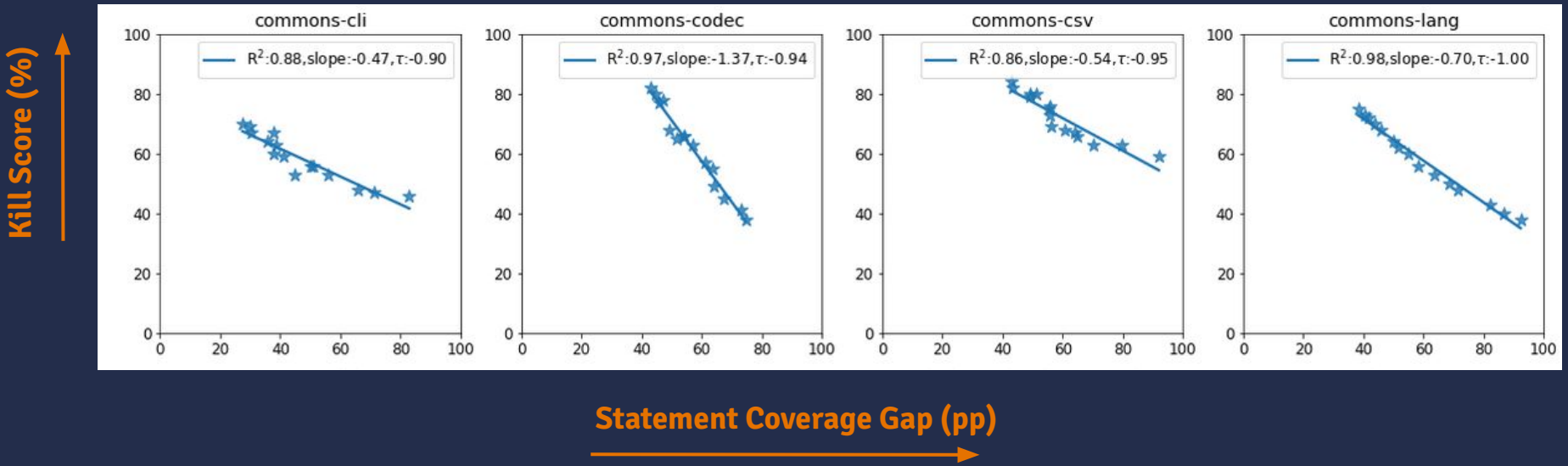
RQ2: Impact of Gaps on Fault Detection

Study Design:

- Generate 180 test suites by manipulating the gap size
- Generated 96K mutants to evaluate fault detection effectiveness
- Measure the correlation between gaps and kill scores

RQ2: Impact of Gaps on Fault Detection

Granularity: Application, Package
Criteria: Statement, Object branch



RQ2: Impact of Gaps on Fault Detection

Findings: Faults can hide in the coverage gap and there is a *strong negative* and *statistically-significant* correlation between gap size and fault-detection effectiveness.

RQ3: Recommender Performance

Study design:

- Remove developer written assertions from test suites
- Compute the resulting gap
- Analyze the SUT and the gap to recommend focus methods
- Compare recommended focus methods to focus methods in removed assertions

RQ3: Recommender Performance

TABLE III

PERCENTAGE OF ASSERTION FOCUS METHODS RECOMMENDED WITHIN THE TOP-K RECOMMENDATIONS ACROSS ALL 13 ARTIFACTS

Artifacts	Assert(#)	Top 1(%)	Top 5(%)	Top 10(%)
Commons-Cli	332	16	51	70
Commons-Codec	532	84	96	97
Commons-Csv	602	69	84	90
Commons-Lang	9843	80	96	98
Commons-Validator	1441	50	77	89
Jackson-Dataformat-Xml	83	33	43	63
Jaxen	134	11	30	37
JFreeChart	3240	82	93	97
Joda-Time	15775	17	43	53
Jsoup	1098	21	31	38
Gson	871	53	82	87
Plexus-Utils	365	55	75	78
XStream	578	38	56	59
Summary	**Total 34894**	**Average 46**	**Average 67**	**Average 73**

RQ3: Recommender Performance

TABLE III
PERCENTAGE OF ASSERTION FOCUS METHODS RECOMMENDED WITHIN THE TOP-K RECOMMENDATIONS ACROSS ALL 13 ARTIFACTS

Commons-Codec	532	84	96	97
Commons-Lang	9843	80	96	98
[illegible]	1441	50	77	89
[illegible]	83	33	43	63
Jaxen	134	11	30	37
JFreeChart	3240	82	93	97
Joda-Time	15775	17	43	53
Jsoup	1098	21	31	38
Gson	871	53	82	87
Plexus-Utils	365	55	75	78
XStream	578	38	56	59
Summary	Total	Average	Average	Average
	34894	46	67	73

Finding: On average, 67% of the focus methods in the original test suites are suggested within the top-5 recommendations. Restricting to the top-1 recommendation, nearly half of the developer-written focus methods are present.

What next?

- Reduce the cost of gap analysis
- Automate the insertion of recommended assertions

Artifact: https://github.com/soneyahossain/hcc-gap-recommender

Acknowledgement: DARPA ARCOS FA8750-20-C-0507, Air Force Office of Scientific Research FA9550-21-0164, and Lockheed Martin Advanced Technology Laboratories