Is Coincidental Correctness Less Prevalent in Unit Testing?

Wes Masri
American University of Beirut
Electrical and Computer Engineering Department

Outline

- **Definitions** – Weak CC vs. Strong CC
- **Causes of Coincidental Correctness**
- **Prevalence of CC** – *previous study*
- **Relation to Dependence Analysis**
- **Impact on Coverage-based Techniques** – CBFL and TSR
- **CC and Unit Testing – *Defects4J***
  - **Test Cases Breakdown** – True Passing, Failing, Weak CC, Strong CC
  - **Propagation Analysis**
  - **Bug Classification**

Definitions (1)

Coincidental Correctness **arises when the program produces the correct output, while:**

1) **Reachability** -- *is met* ← **Weak CC**

The defect is executed

2 definitions for a reason…

2) **Infection** -- *is met* ← **Strong CC**

The program has transitioned into an infectious state

3) **Propagation** -- *is not met*

The infection has propagated to the output

Definitions (II)

- CC might be perceived as a ***good thing***!
  - ***The program is working correctly… so why worry?***

- Two Problems:
  - ***Strong CC -*** results in overestimating the reliability of programs: *it hides defects that subsequently might surface following unrelated code modifications*

  - ***Weak CC & Strong CC*** - reduce the effectiveness of coverage-based techniques

Causes of Strong CC (1)

- Case when

The Infection fails to Propagate to the Output

- Consider x that takes on the values [1, 5], such that the program gets infected when x = 4

s_1 : y = x * 3;

- There is a clear one-to-one mapping between the *x* values and *y* values: {1→3, 2→6, 3→9, 4*→12*, 5→15}
- When *x* is infected, the corresponding *y* value, which is unique, will successfully propagate the infection past s_1
- That is, the infection *x*=4 leads to the infection *y*=12.

Causes of Strong CC (2)

```
s2 : if (x >= 3) {
            y = 1;
     } else {
            y = 0;
     }
```

- Here the mapping is {1→0, 2→0, 3→1, 4*→1, 5→1}
- There is no unique value of *y* that captures the infection
- *y*=1 is not an infection since it also results from *x*=3 and x=5
- The infection was nullified by the execution of s_2

- Constructs similar to s_2 are pervasive → prevalence of *strong CC*

Prevalence of CC

- From previous study:
  - **148** versions of ten Java programs (*NanoXML* and *Siemens*)
  - Test suite sizes ranged from 140 to 4130, with a total of 19,873

- ***Strong CC***: 3,120 tests (15.7%)
- ***Weak CC***: 11,208 tests (56.4%)

- 20 versions had more than 60% of their tests as *strong CC*
- 86 versions had more than 60% of their tests as *weak CC*.
- One version had 99.3% of its tests as *strong CC*

- *Failure Checkers: mostly trivial… seeded bugs*

Strong CC and Dependence Analysis (1)

Forms of Dependence Analysis: **Static →Dynamic →Strength-based**

- Basic Assumption of Dynamic Dependence Analysis:

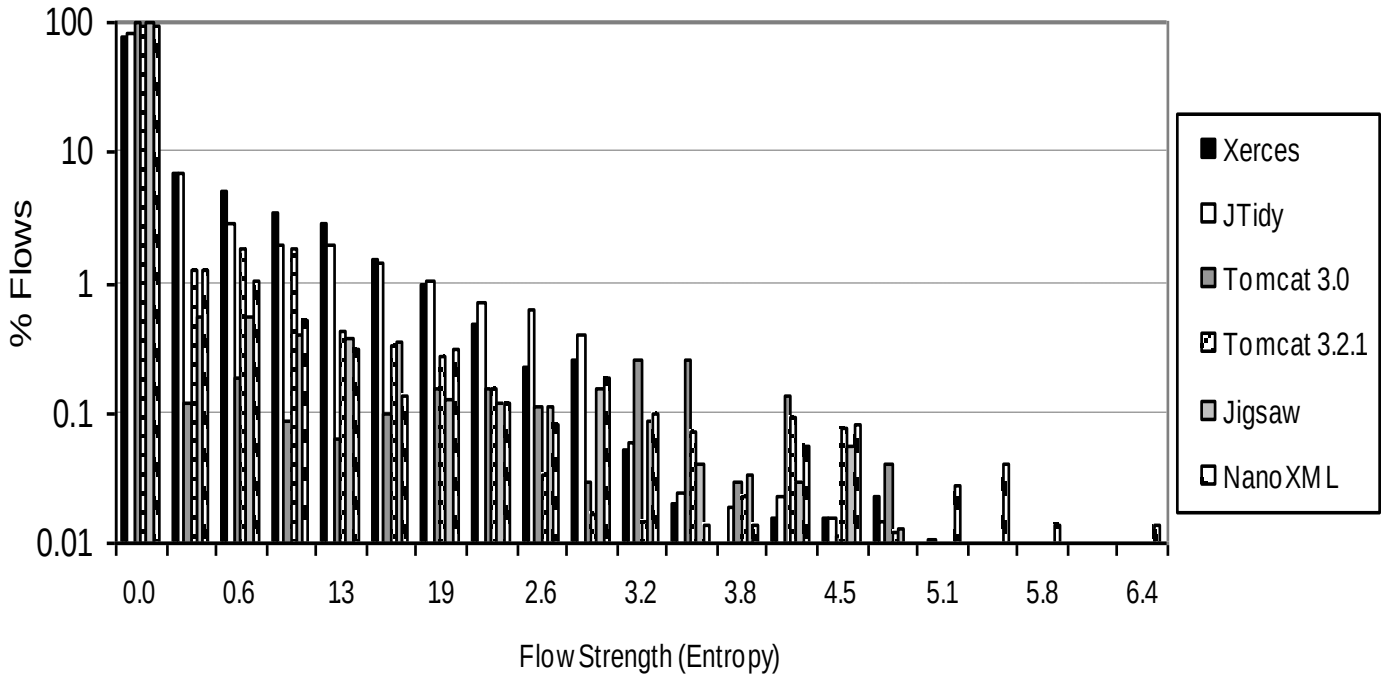
  If two variables are connected by a sequence of dynamic data and/or control dependences, then information actually flows between them

- To empirically validate this assumption, we used an *information theoretic measure* to answer the following questions:
  - ***Does dynamic program dependence always imply information flow?***
  - ***Is the Length of an Information Flow indicative of its Strength?***
  - ***Which Dependences are Stronger? Data or Control?***

Strong CC and Dependence Analysis (II)

- ***Does dynamic program dependence always imply information flow?***

In 90%+ of the cases, dynamic dependences did not channel any information!!! …Unexpected

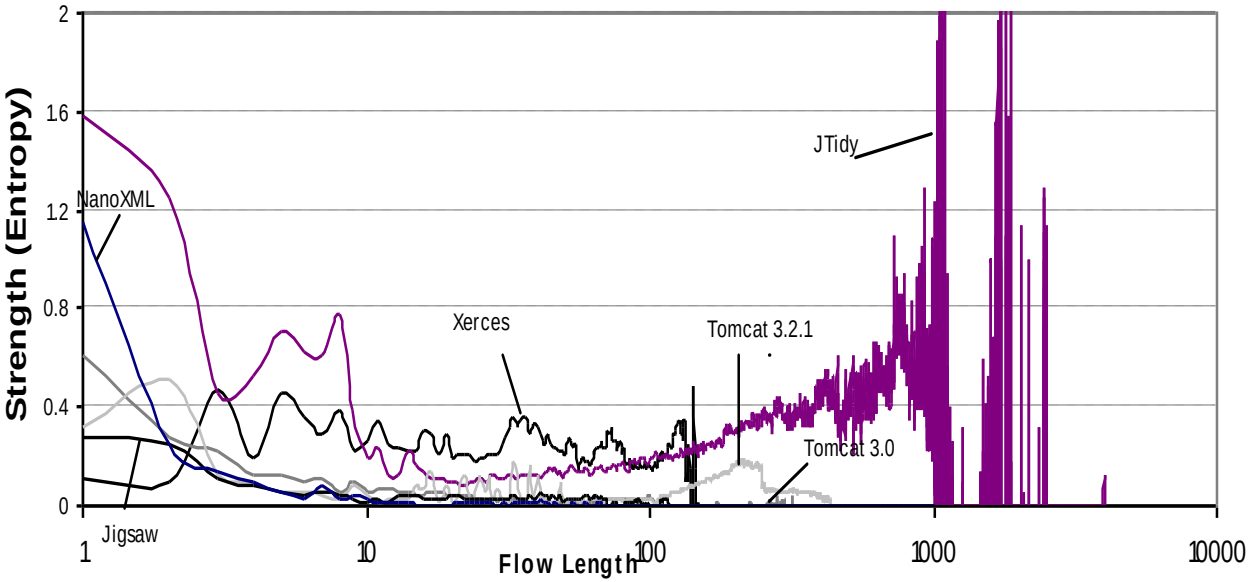


Strong CC and Dependence Analysis (III)

- ***Is the Length of an Information Flow indicative of its Strength?***

Many long flows were strong

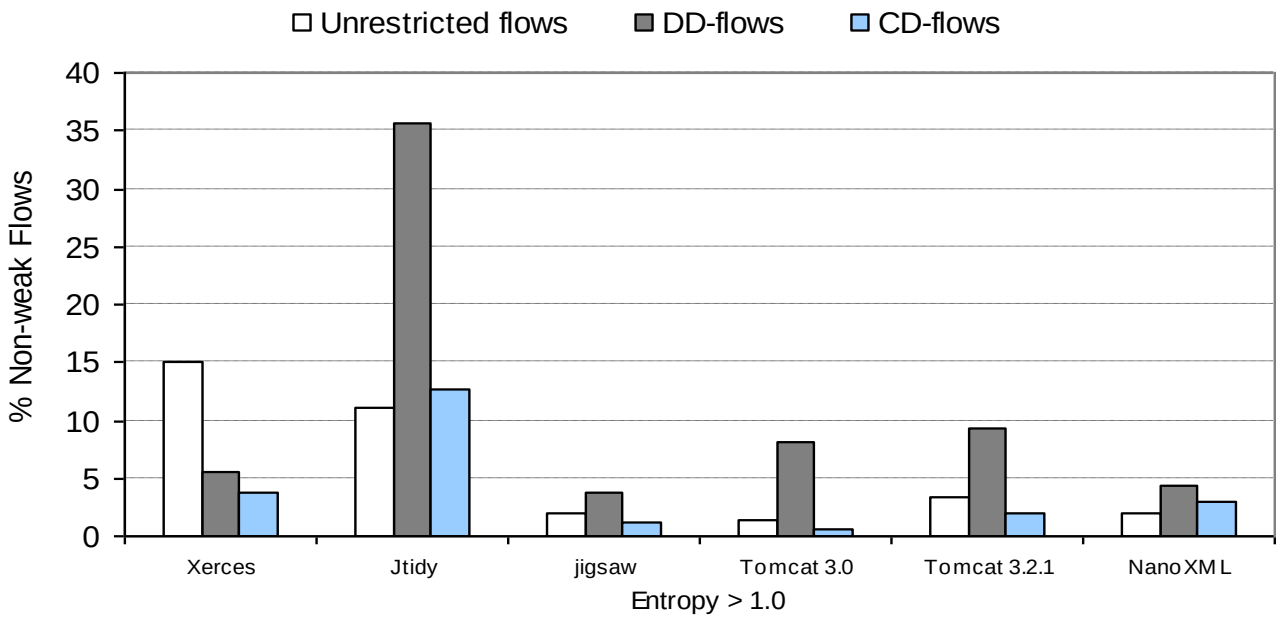
Many short flows were weak …Unexpected



Strong CC and Dependence Analysis (IV)

- ***Which Dependences are Stronger? Data or Control?***

Flows due to data dependences alone are stronger, on average, than flows due to control dependences alone … rather expected…



Strong CC and Dependence Analysis (V)

In 90%+ of the cases, dynamic dependences did not channel any information!!!

Suggests that many infectious states might get cancelled and not propagate to the output, thus, leading to a potentially high rate of Strong CC

Impact on Coverage-based Fault Localization

CC Underestimates the Suspiciousness of Faulty Program Elements

- Example: Tarantula suspiciousness metric

$$M(e) = F / (F + P)$$

e = faulty program element
F = % of failing runs that executed e
P = % of passing runs that executed e

Given *n* coincidentally correct tests, *n* should be taken out from *P* and added to *F* to arrive at :

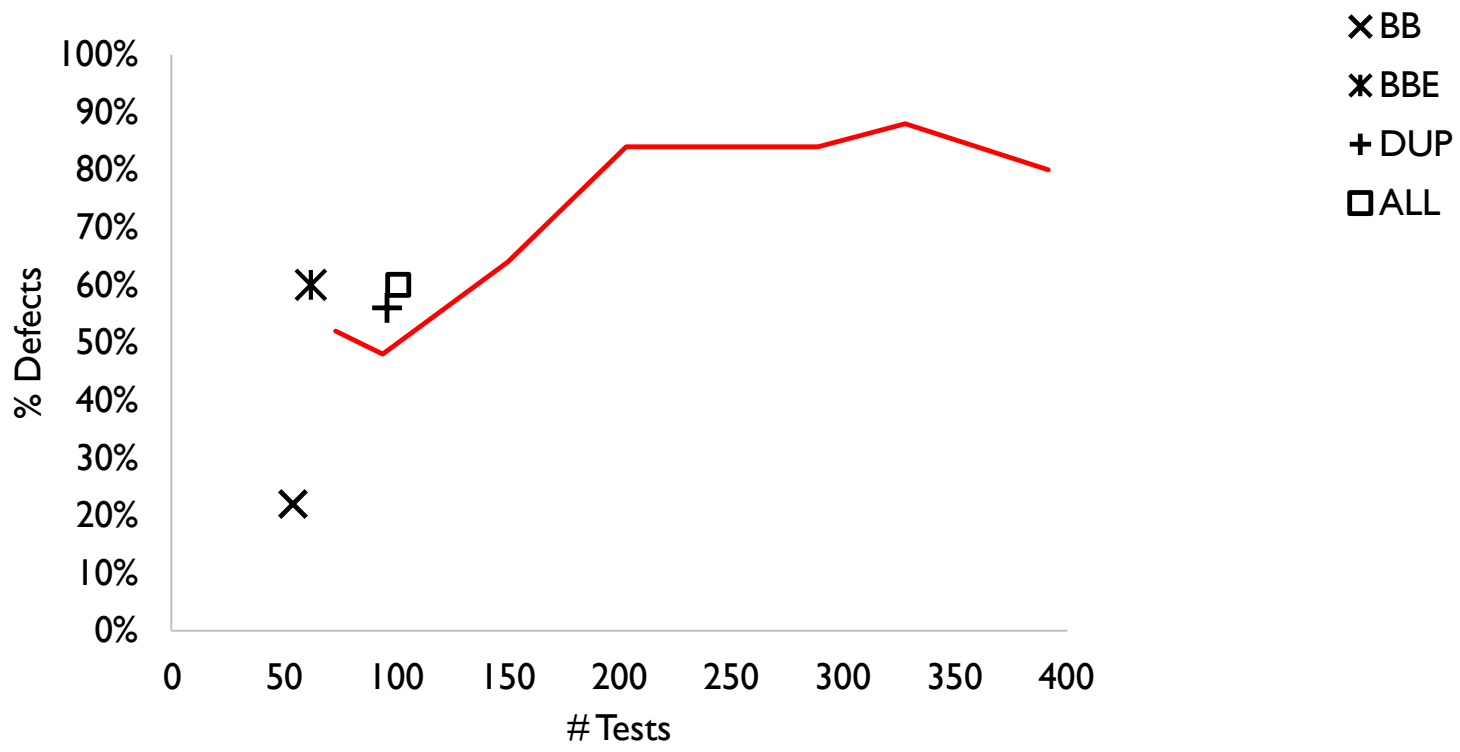
$$M'(e) = F' / (F' + P')$$

It could be easily shown that $M'(e) \geq M(e)$

That is, not accounting for *CC* would underestimate the suspiciousness of *the faulty program element*

CC is a Safety reducing factor in CBFL

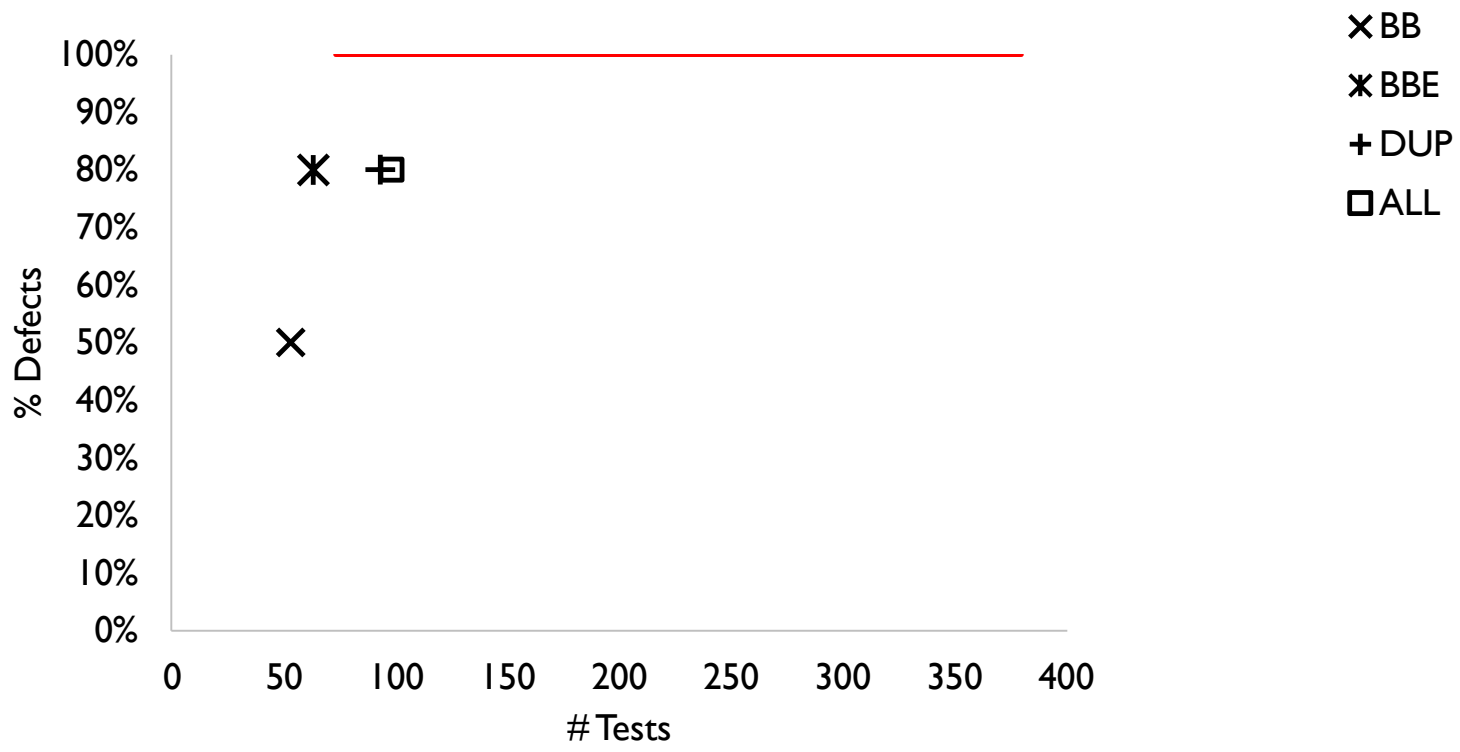
Impact on Test Suite Reduction (I)



JTidy, 1000 test cases, 5 defects, 24 failures

23 CC tests

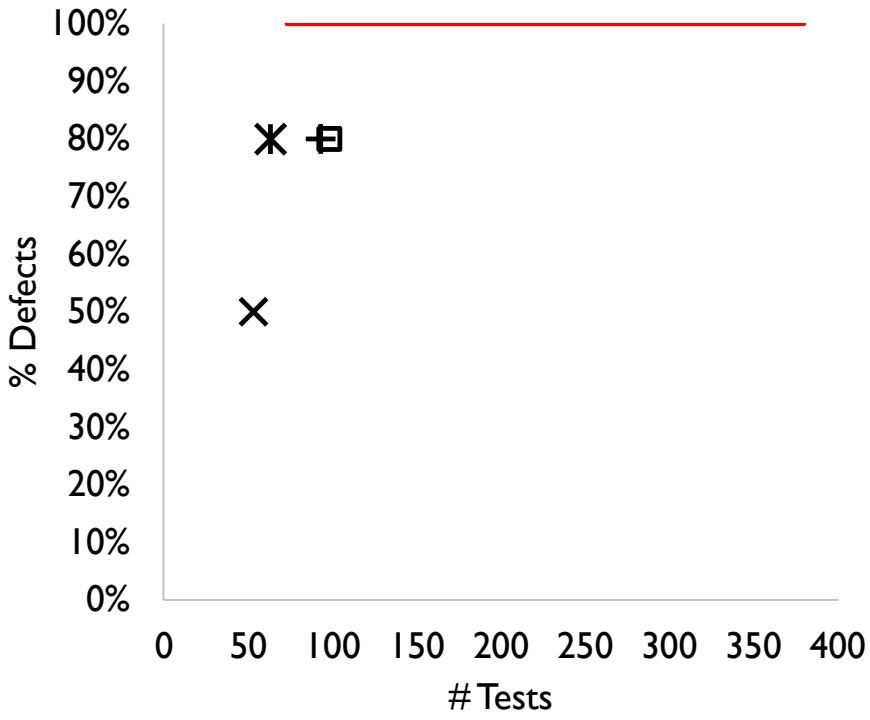
Impact on Test Suite Reduction (II)

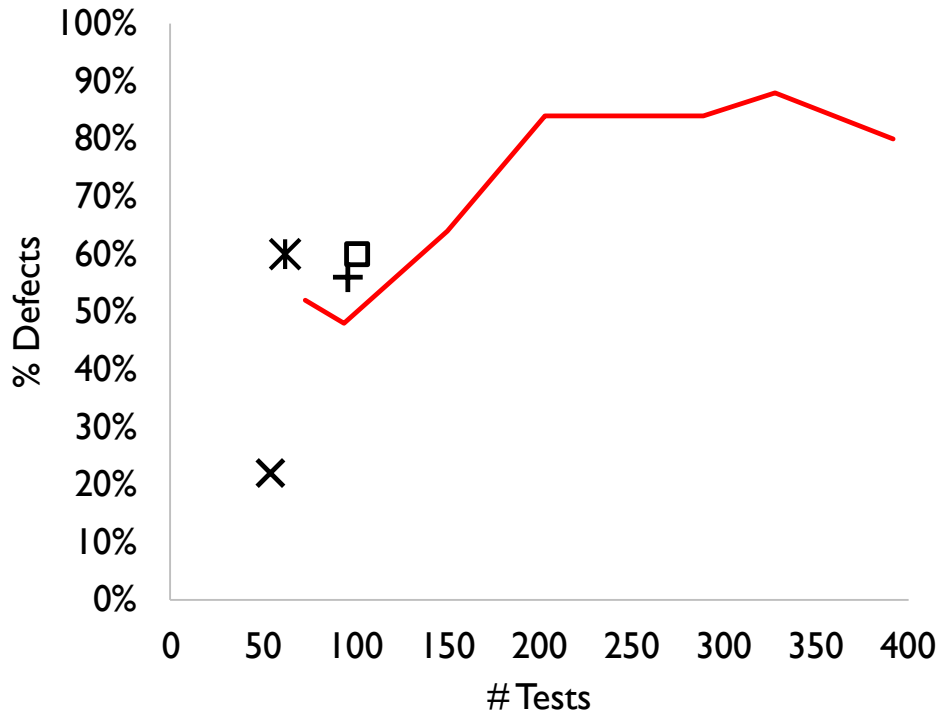


JTidy, 977 test cases, 5 defects, 24 failures

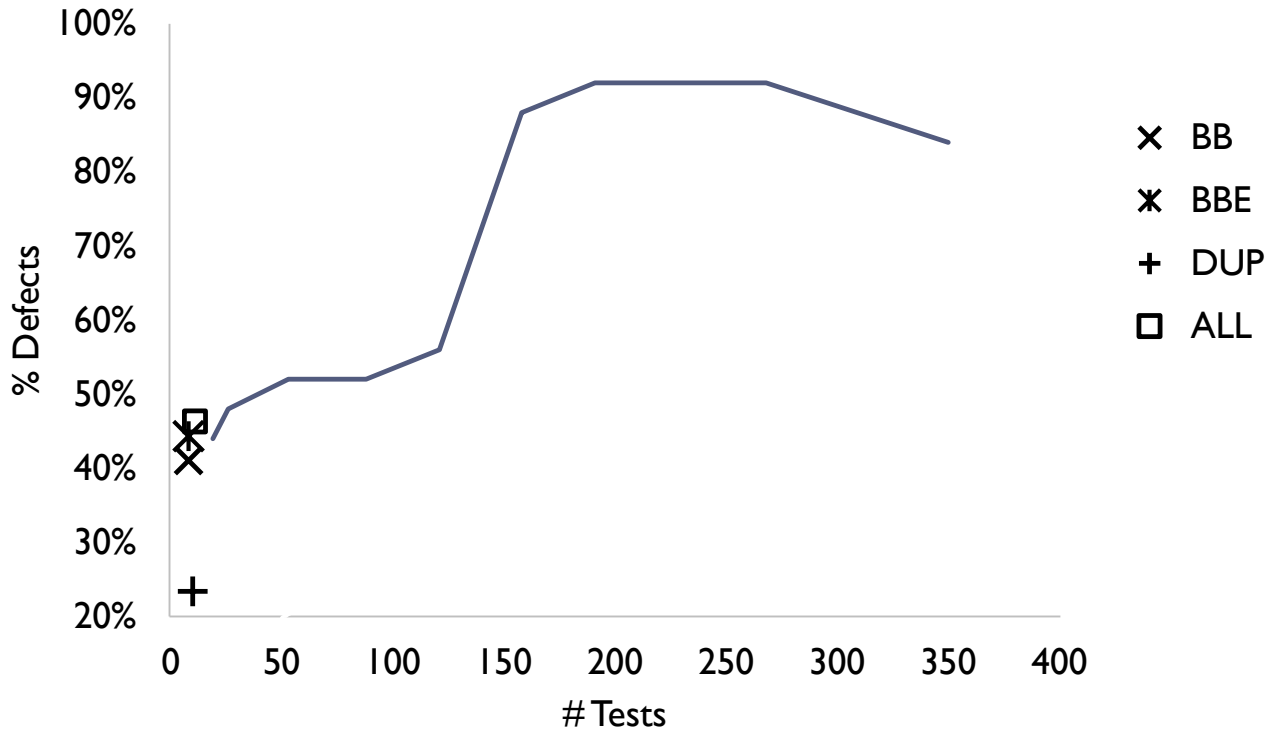
0 CC tests

Impact on Test Suite Reduction (III)





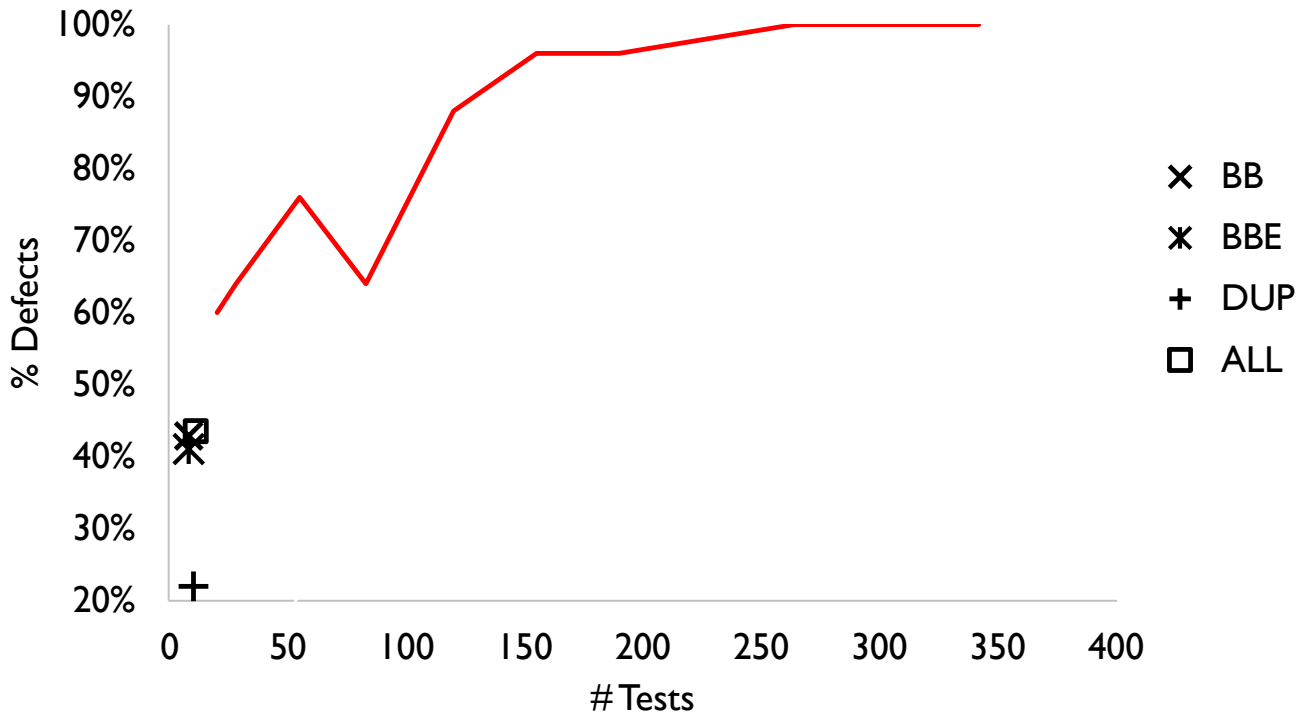
Impact on Test Suite Reduction (IV)



Math, 1857 test cases, 5 defects, 42 failures

57 CC tests

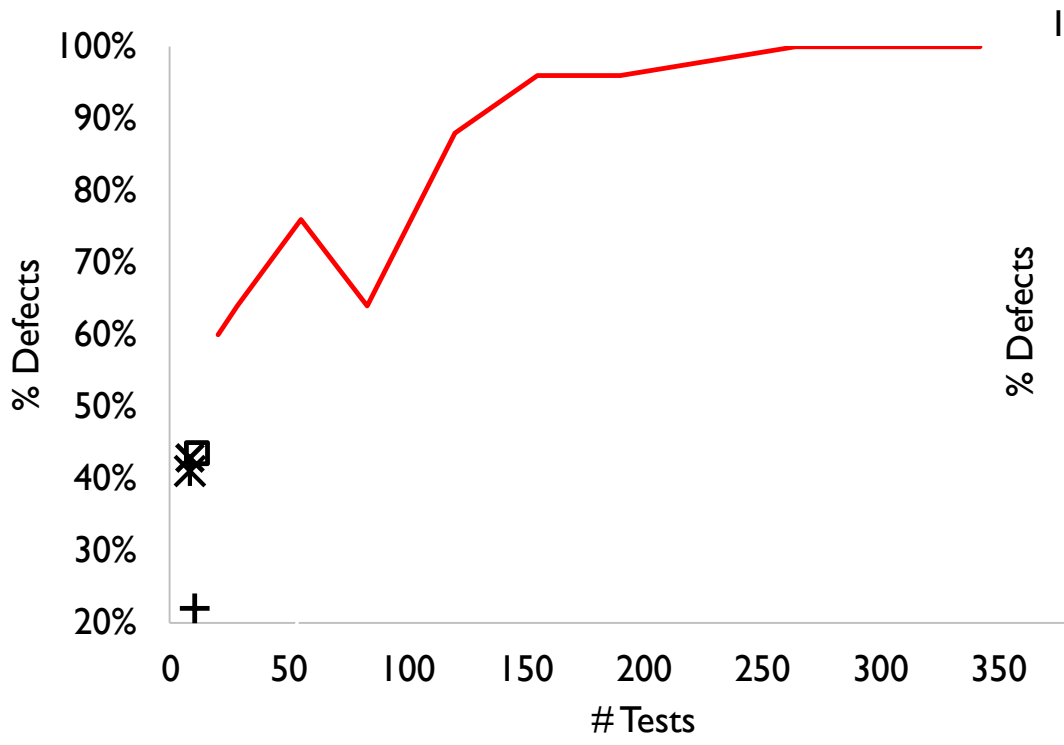
Impact on Test Suite Reduction (V)

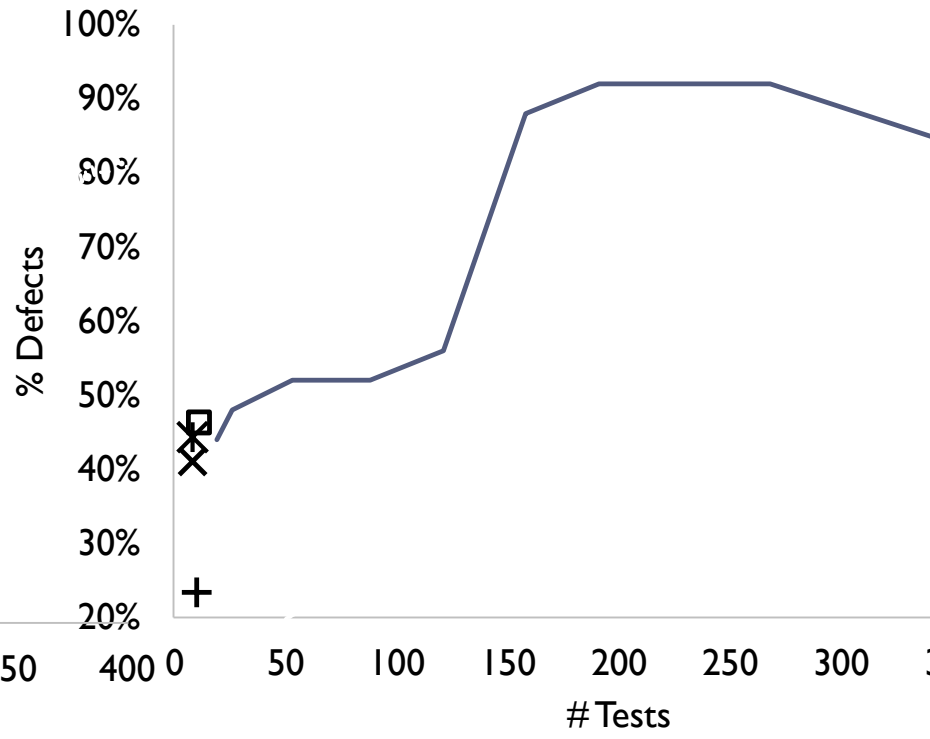


Math, 1800 test cases, 5 defects, 42 failures

0 CC tests

Impact on Test Suite Reduction (VI)





Defects4J

- De facto benchmark in program repair research and other
- Consists of 395 real bugs distributed over 6 libraries

Library	Number of bugs
Closure compiler	133
Apache Commons Math	106
Apache Commons Lang	65
Mockito	38
Joda Time	27
JFreeChart	26

Targeted in this presentation

Source: https://github.com/rjust/defects4j

[] René Just, Darioush Jalali, Michael D. Ernst. Defects4J: a database of existing faults to enable controlled testing studies for Java programs. ISSTA 2014: 437-440.

Identifying CC Tests within Defects4J: Why?

- CC is a confounding factor
  - When evaluating new techniques, researchers using *Defects4J* will be able to factor out the impact of Coincidental Correctness *(by discarding CC tests or treating them as failing)*

- Determining whether CC is as prevalent at the unit testing level *(than at higher levels of testing)*
  - If less prevalent
    - An argument for conducting CBFL and other coverage-based techniques at the unit testing level
    - An additional argument in favor of *Test-Driven Development*

Lang Library

- Provides helper utilities for the java.lang API
  - String manipulation methods
  - Basic numerical methods
  - Object reflection
  - Concurrency
  - …
- Number of defects: 65

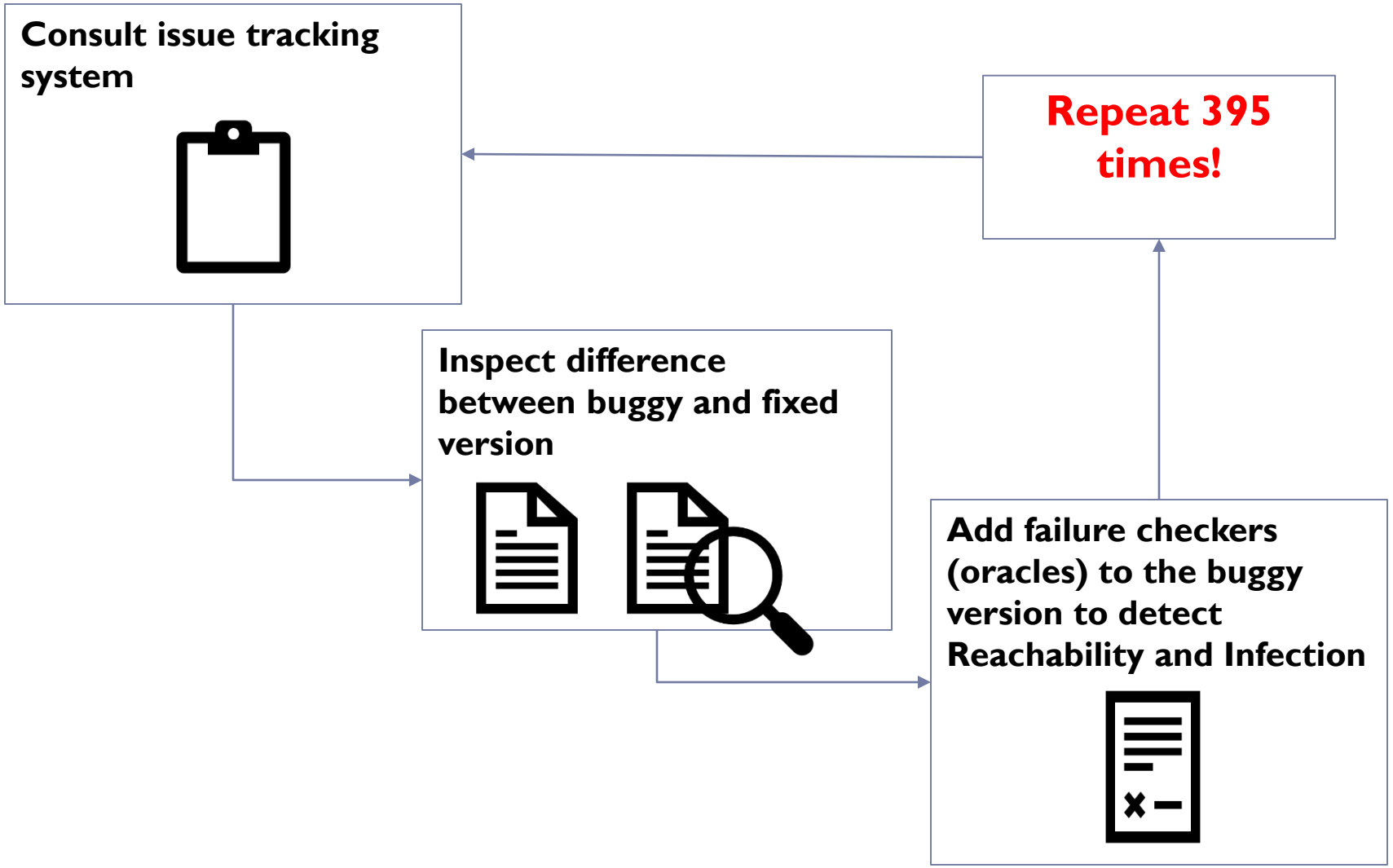
Source: https://commons.apache.org/proper/commons-lang/

Commons Math Library

- Provides mathematical and statistical components:
  - Complex numbers
  - Matrices
  - …
- Number of defects: 106

Source: http://commons.apache.org/proper/commons-math/

How to identify the CCs in Defect4J



Augmenting buggy versions with oracles to identify CCs (trivial)

Math library, bug #10: DSCompiler.java

Buggy Version:with oracles:

```
...else {
  subtract(tmp1, 0, x, xOffset, tmp2, 0);
  divide(y, yOffset, tmp2, 0, tmp1, 0);
  atan(tmp1, 0, tmp2, 0);
  result[resultOffset] = ((tmp2[0] <= 0) ? -FastMath.PI :
                          FastMath.PI) - 2 * tmp2[0];
  for (int i = 1; i < tmp2.length; ++i)     {
     result[resultOffset + i] = -2 * tmp2[i];
   }

System.out.println("\nWeak Oracle 10");
if (result[resultOffset] != FastMath.atan2(y[yOffset], x[xOffset])) {
        System.out.println("\nStrong Oracle 10");
}

  }
}
}
```

Fixed Version:

```
...else {
  subtract(tmp1, 0, x, xOffset, tmp2, 0);
  divide(y, yOffset, tmp2, 0, tmp1, 0);
  atan(tmp1, 0, tmp2, 0);
  result[resultOffset] = ((tmp2[0] <= 0) ? -FastMath.PI :
                          FastMath.PI) - 2 * tmp2[0];
  for (int i = 1; i < tmp2.length; ++i)     {
     result[resultOffset + i] = -2 * tmp2[i];
   }

result[resultOffset] = FastMath.atan2(y[yOffset], x[xOffset]);


  }
}
```

Augmenting buggy versions with oracles to identify CCs (non-trivial)

Lang library, bug #40: StringUtils.java

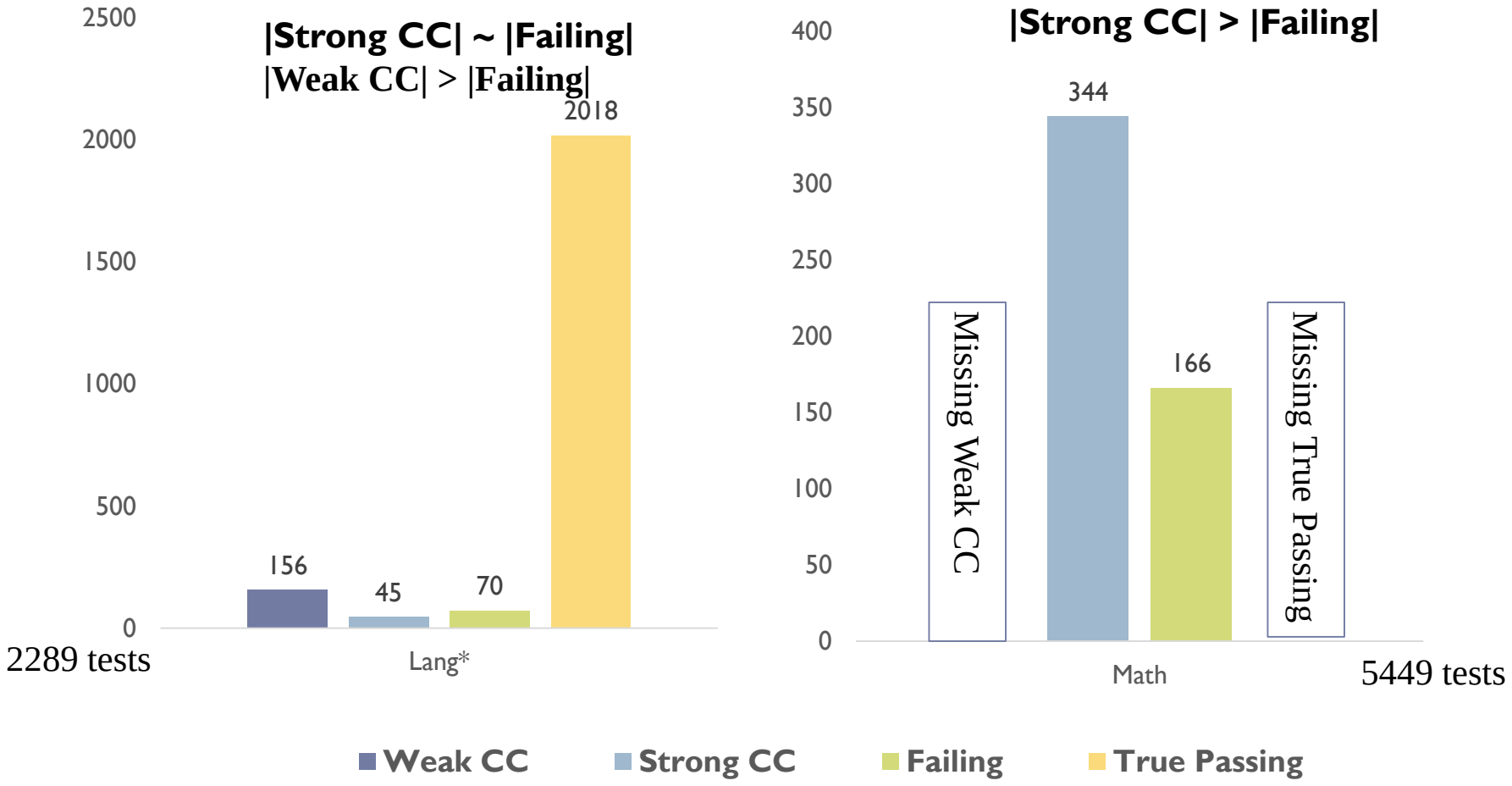
Buggy Version:with oracles:

```
if (str == null || searchStr == null) {
    return false;
}
boolean result = contains(str.toUpperCase(),
searchStr.toUpperCase());
return result;
System.out.println("\nWeak Oracle 40");
boolean fixedResult = false;
int len = searchStr.length();
int max = str.length() - len;
for (int i = 0; i <= max; i++) {
    if (str.regionMatches(true, i, searchStr, 0, len)) {
        fixedResult = true;
        break;
    }
}
if (result != fixedResult) {
    System.out.println("\nStrong Oracle 40");
}
return result;
```

Fixed Version:

```
if (str == null || searchStr == null) {
    return false;
}
int len = searchStr.length();
int max = str.length() - len;
for (int i = 0; i <= max; i++) {
    if (str.regionMatches(true, i, searchStr, 0, len)) {
        return true;
    }
}
return false;
```

Test Cases Breakdown

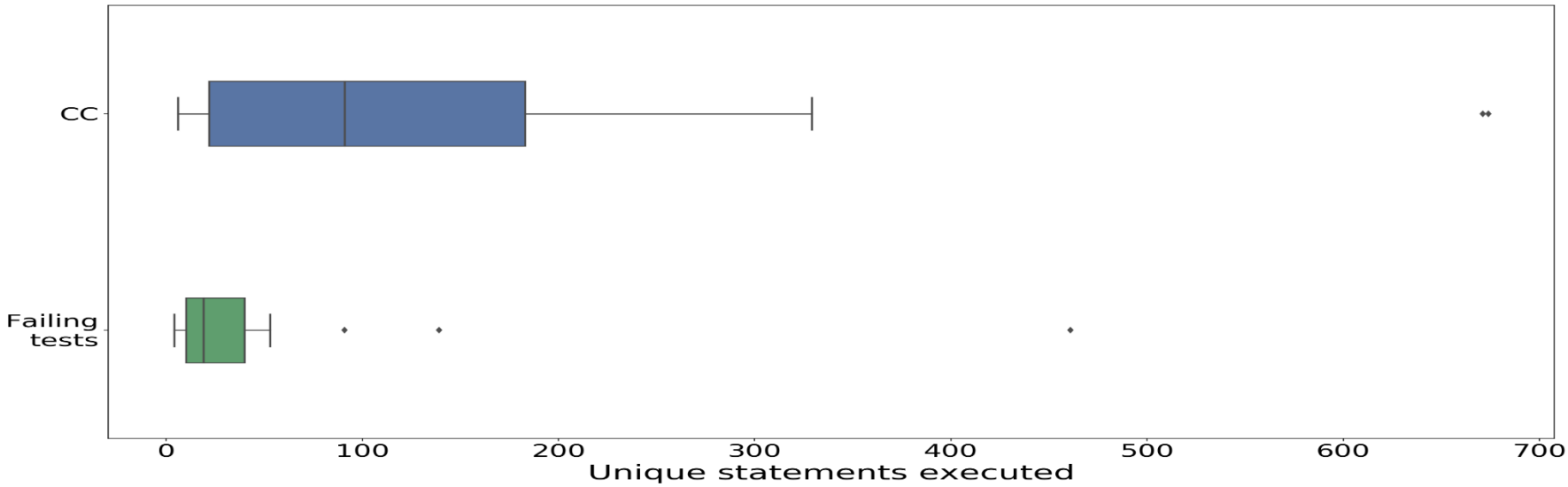


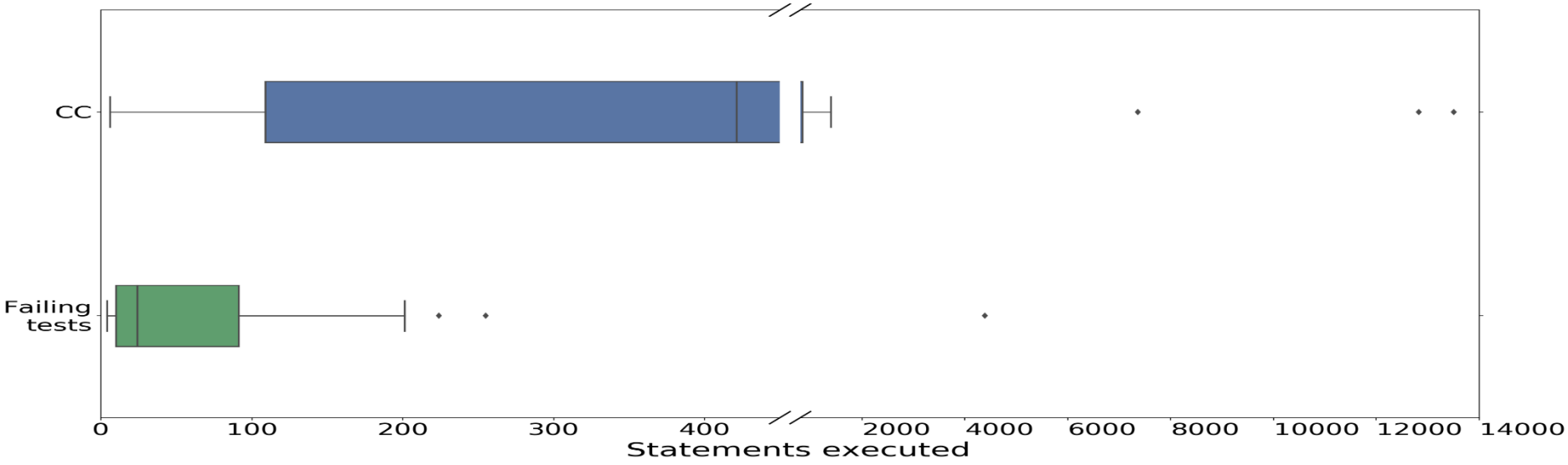
- Lang analysis includes version 34 to 65 only

CC propagation analysis

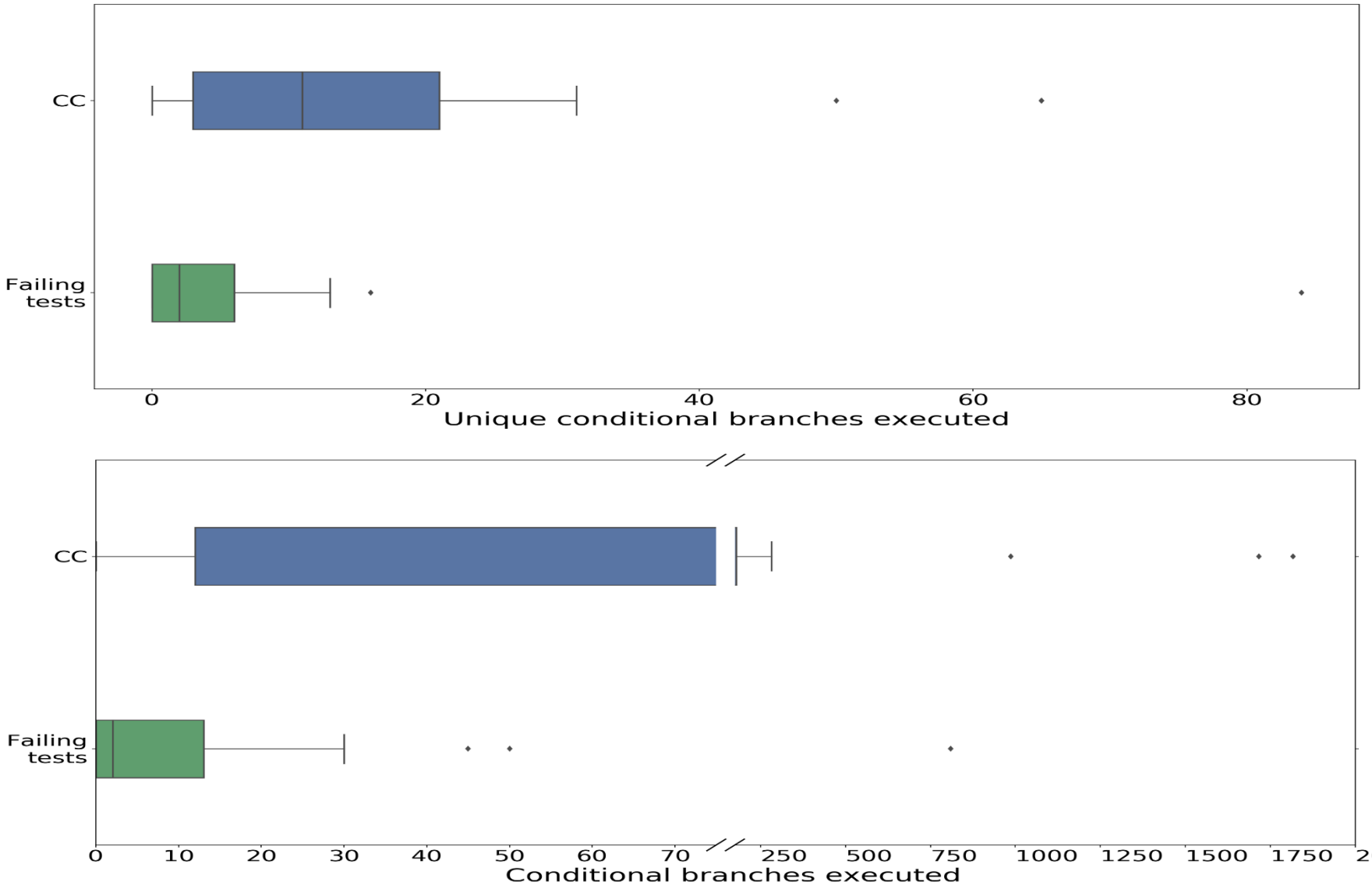
- Following metrics gathered from the moment the oracle is reached (i.e., infection happens) till the test exits to get a sense of the propagation:
  - Statements executed
  - Conditionals executed
  - Method calls executed
  - Modulo operation executed
  - Multiply operation executed
  - Divide operation executed

Lang Library CC analysis: Statements executed

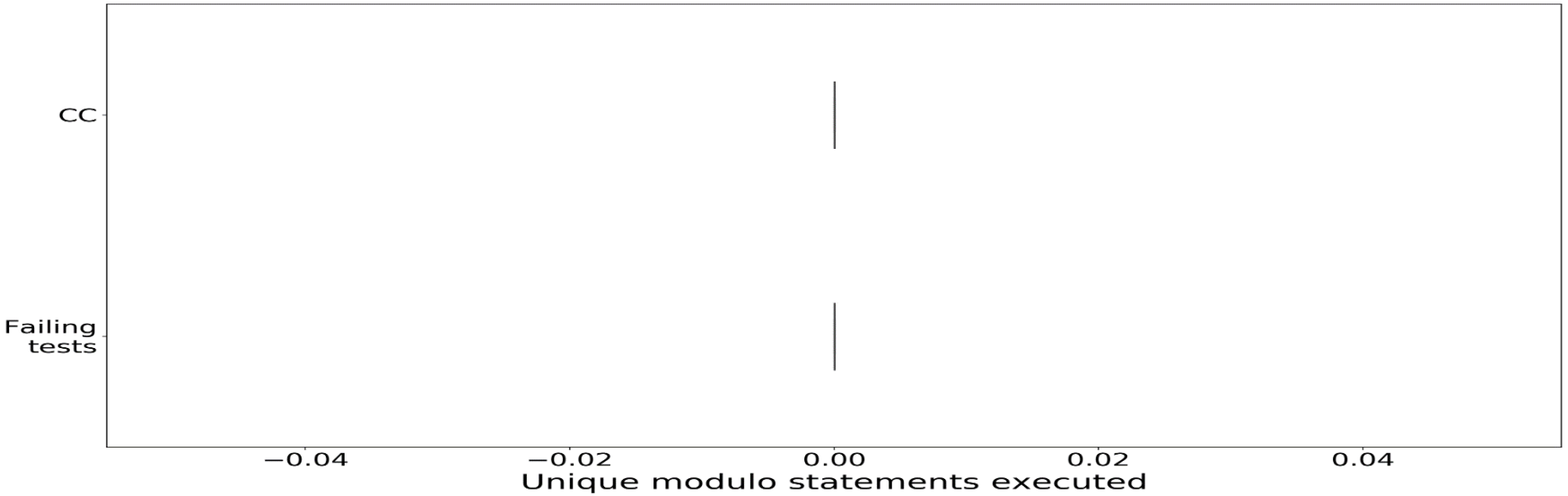


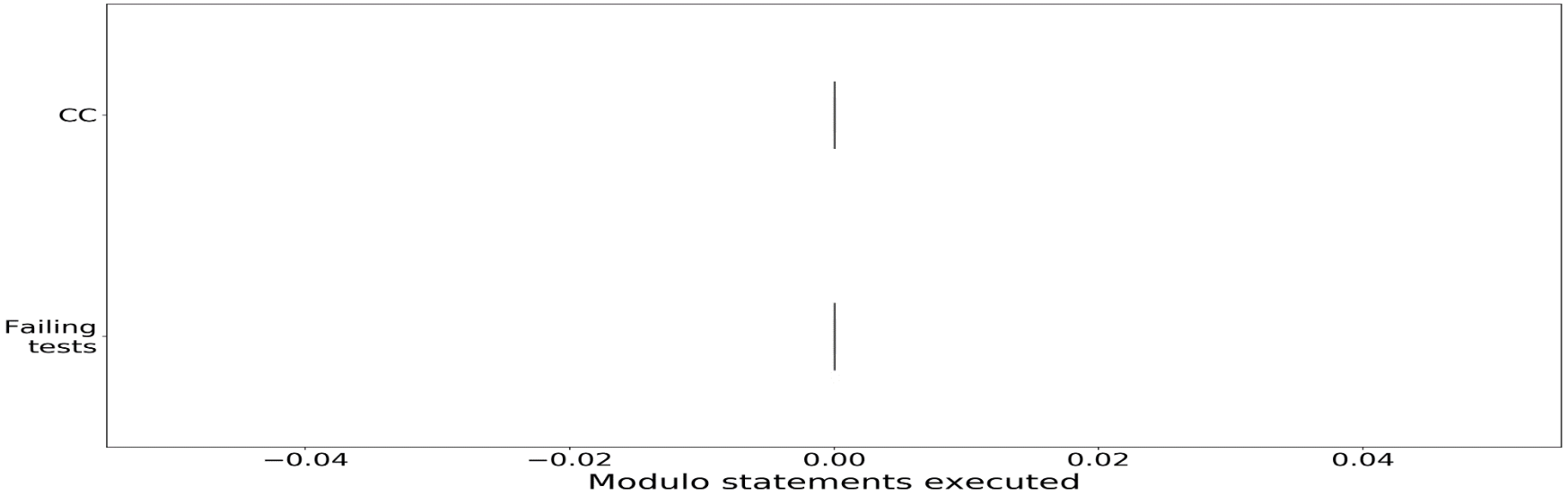


Lang Library CC analysis: Conditional branches executed



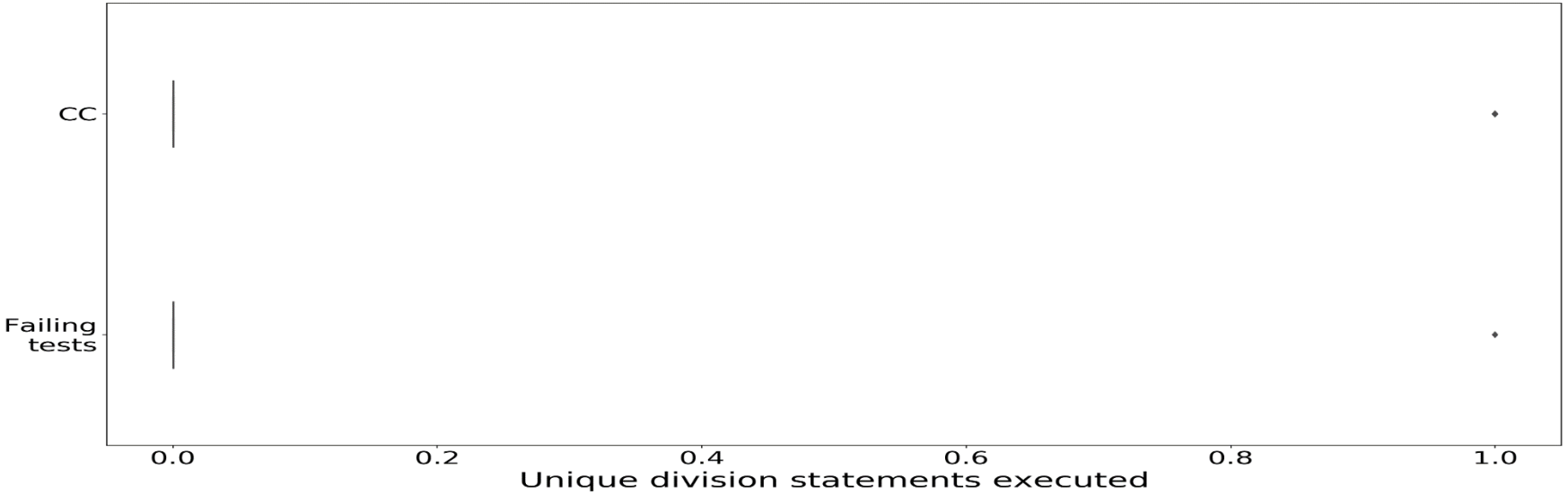
Lang Library CC analysis: Modulo operations executed

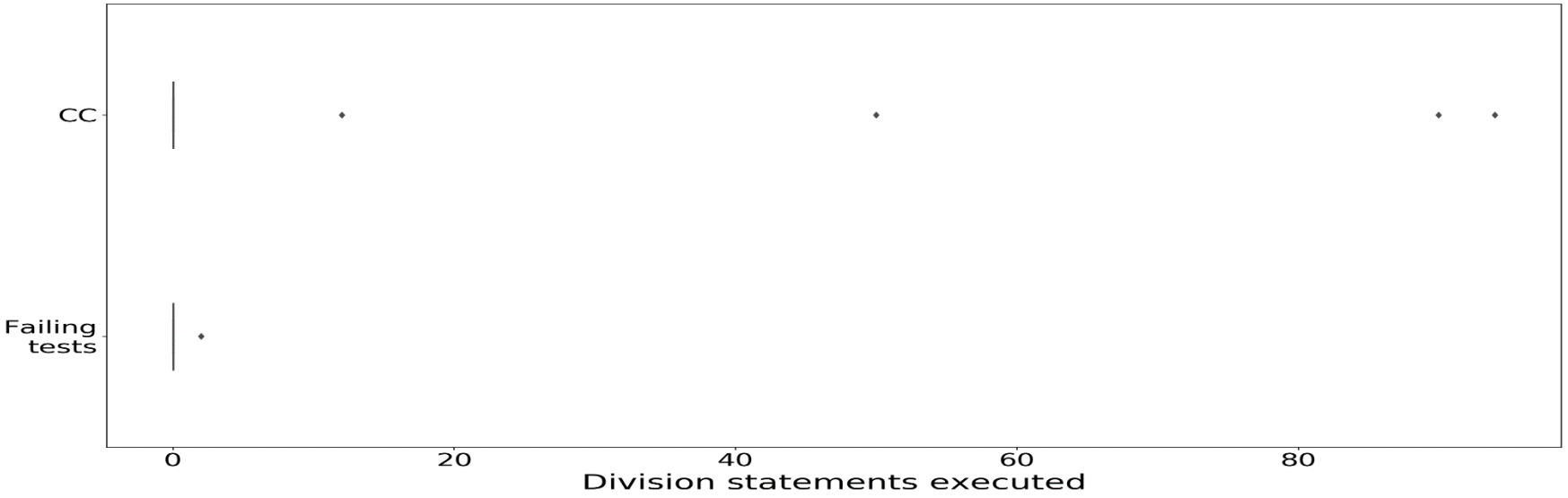




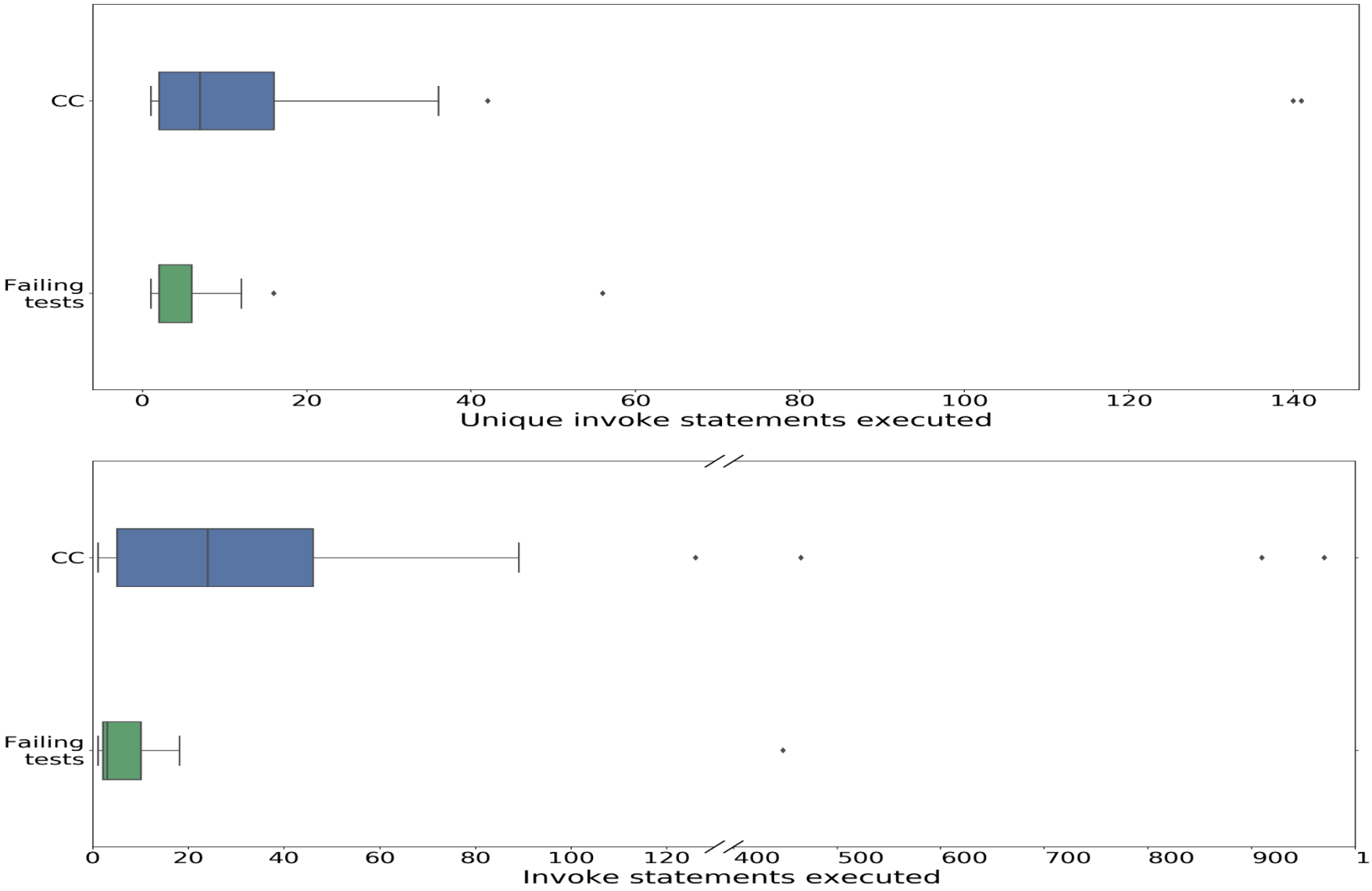
Lang Library CC analysis: Multiplication operations executed

Lang Library CC analysis: Division operations executed

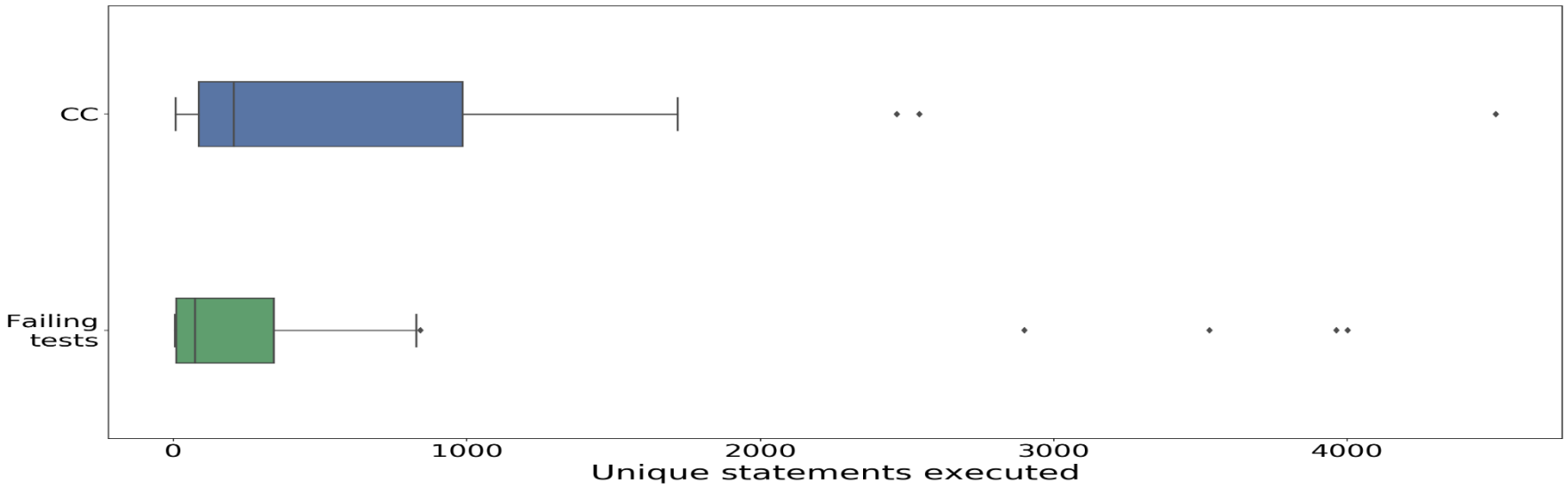


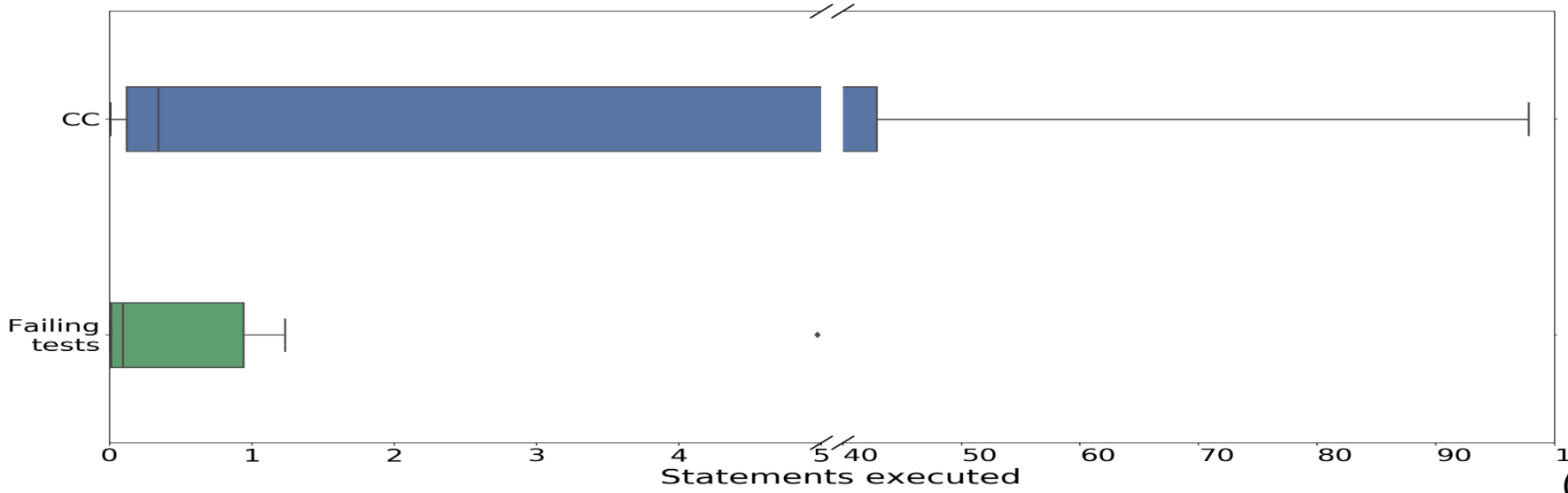


Lang Library CC analysis: method calls



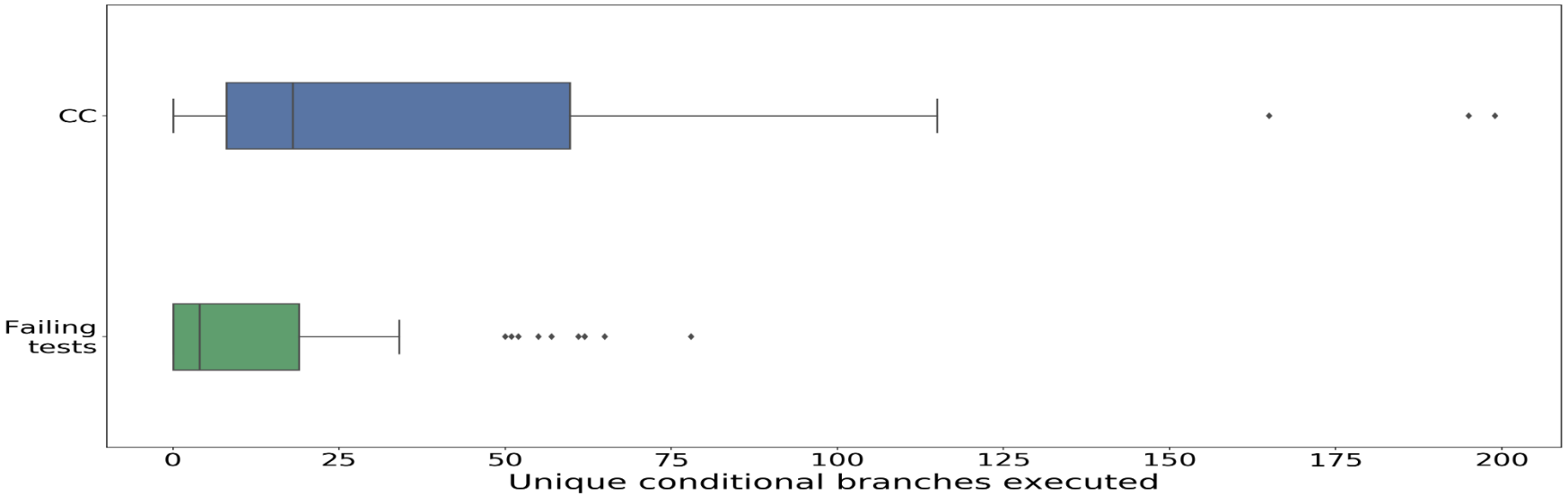
Math Library CC analysis: Statements executed

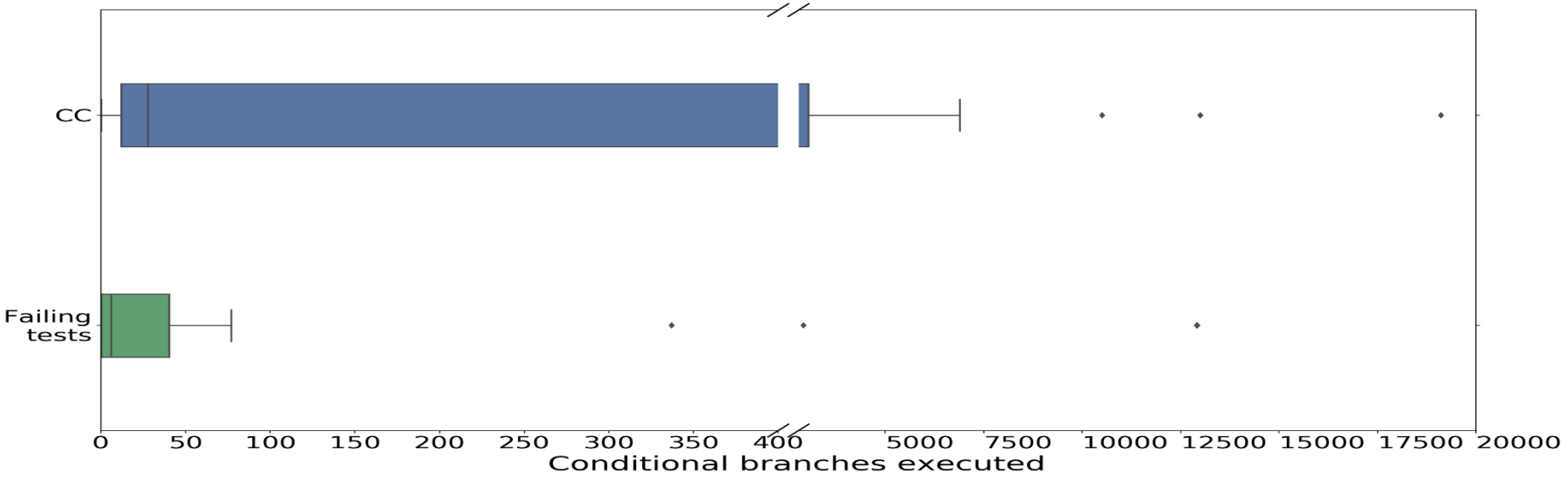




Note: some outliers have been omitted from the bottom graph for visualization purposes

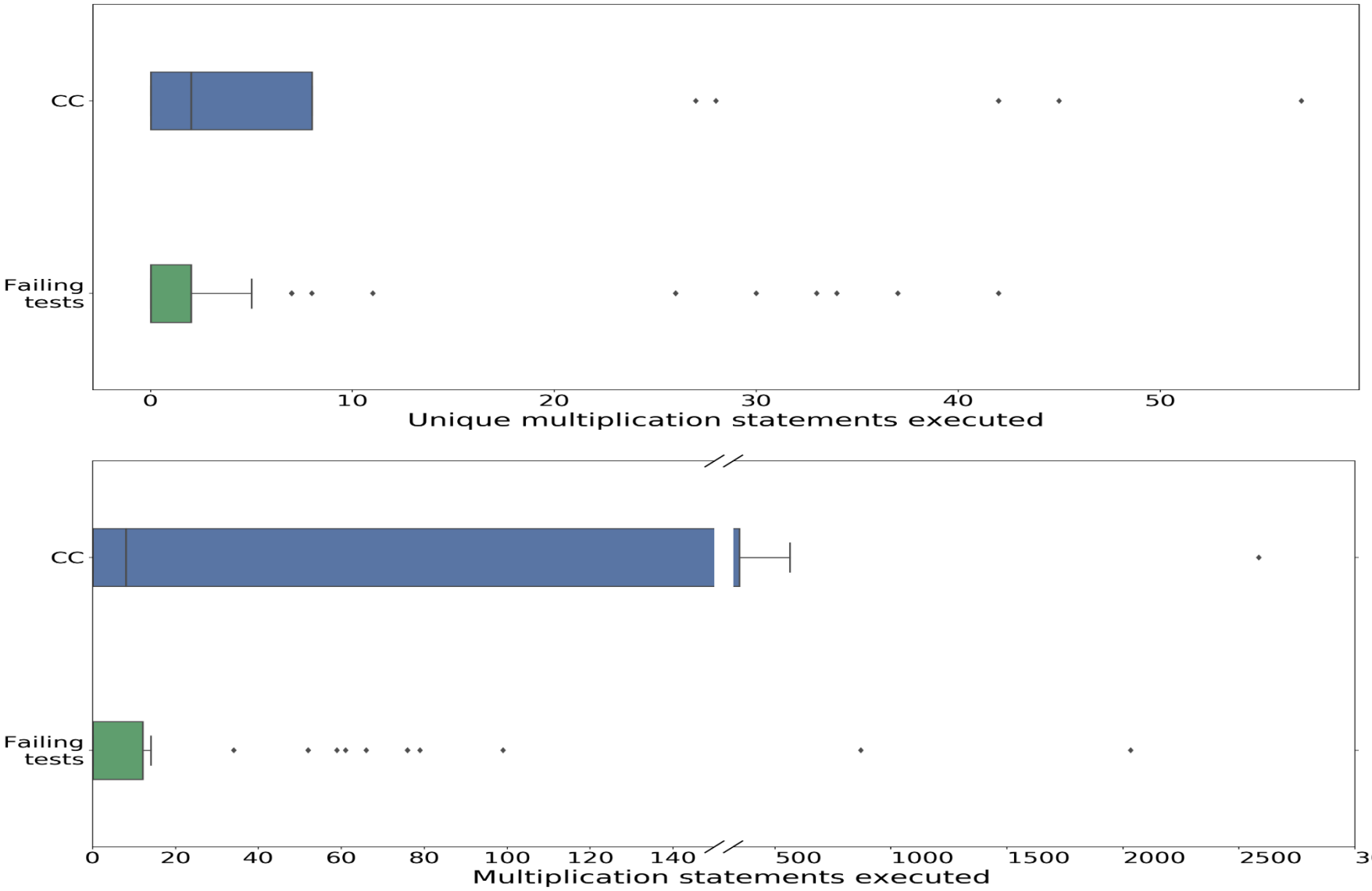
Math Library CC analysis: Conditional branches executed





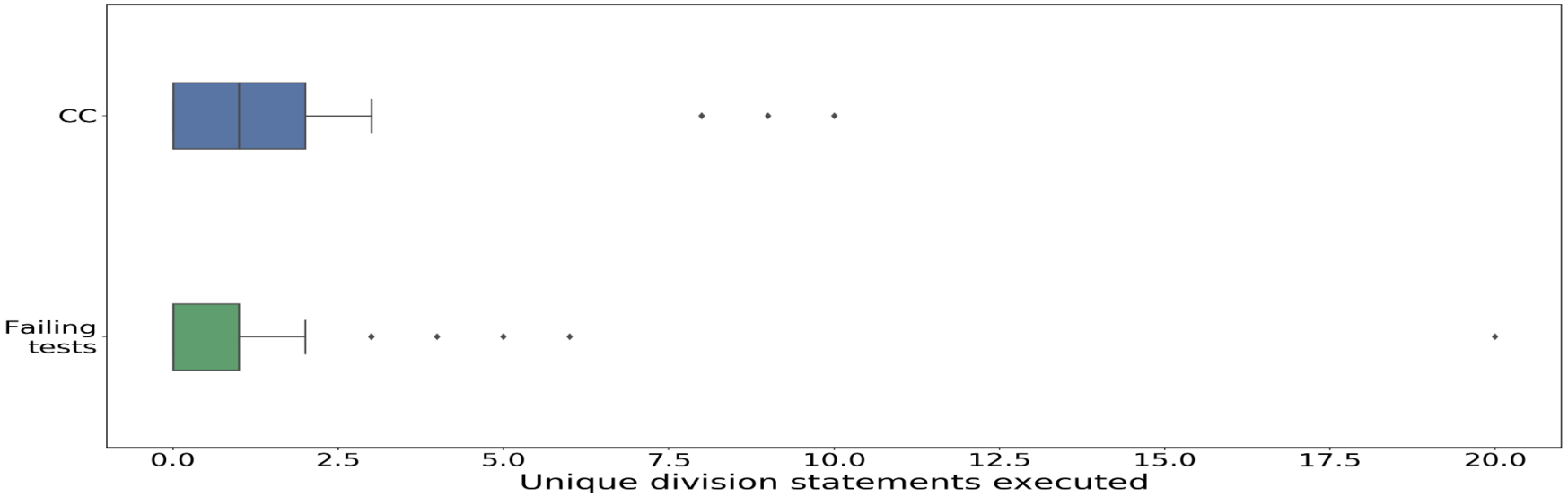
Note: some outliers have been omitted from the bottom graph for visualization purposes

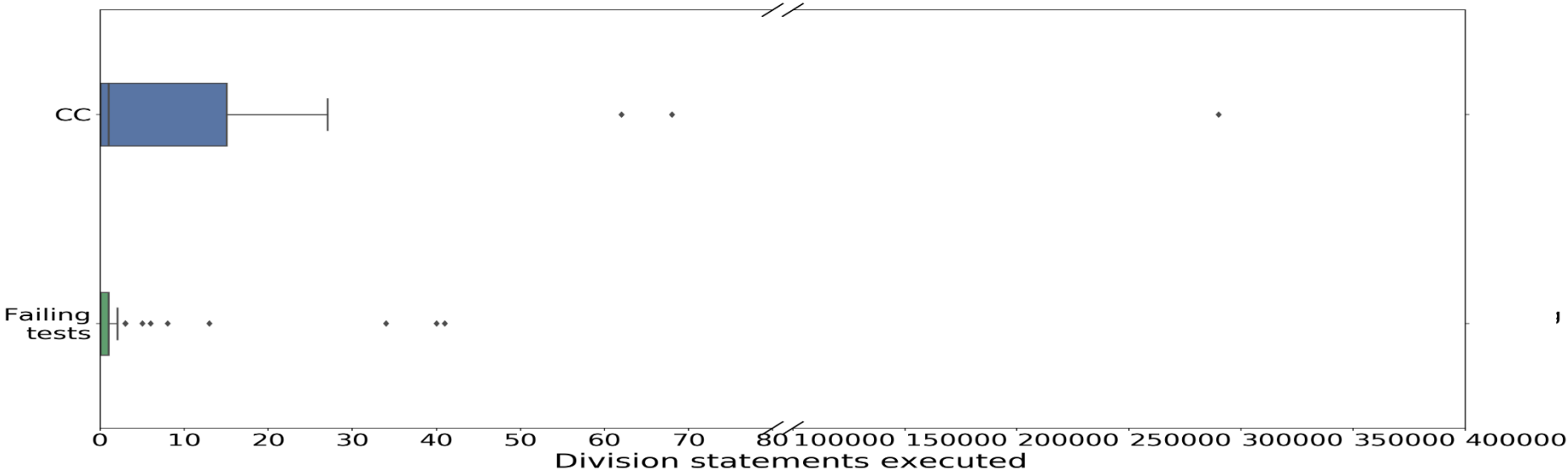
Math Library CC analysis: Multiplication operations executed



Note: some outliers have been omitted from the bottom graph for visualization purposes

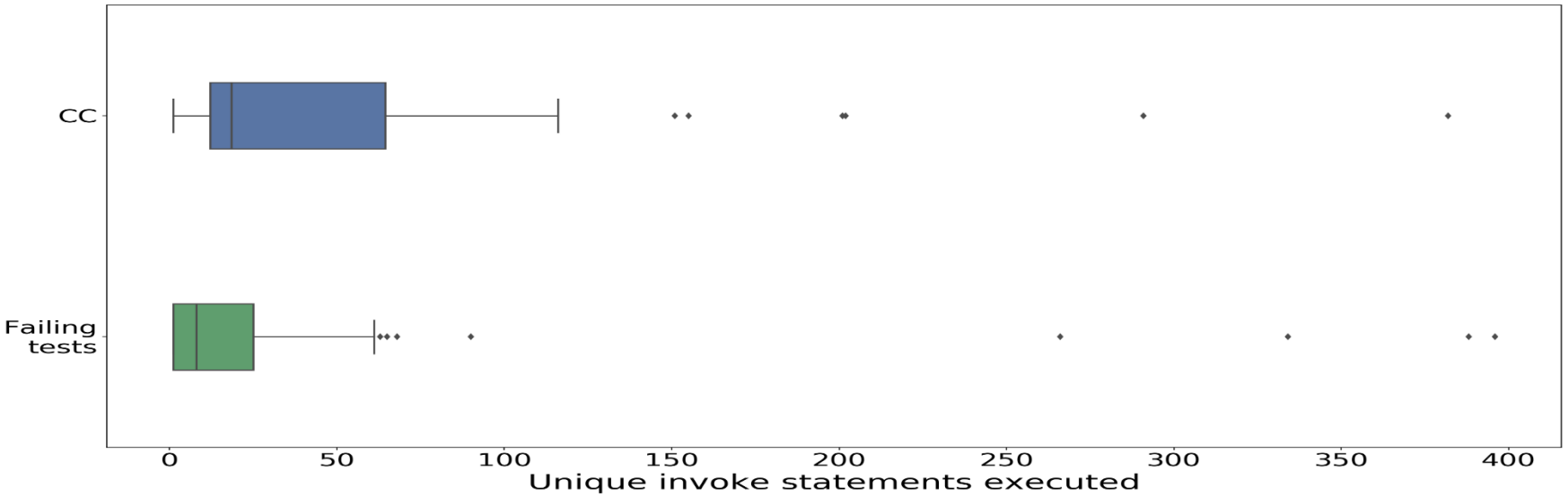
Math Library CC analysis: Division operations executed

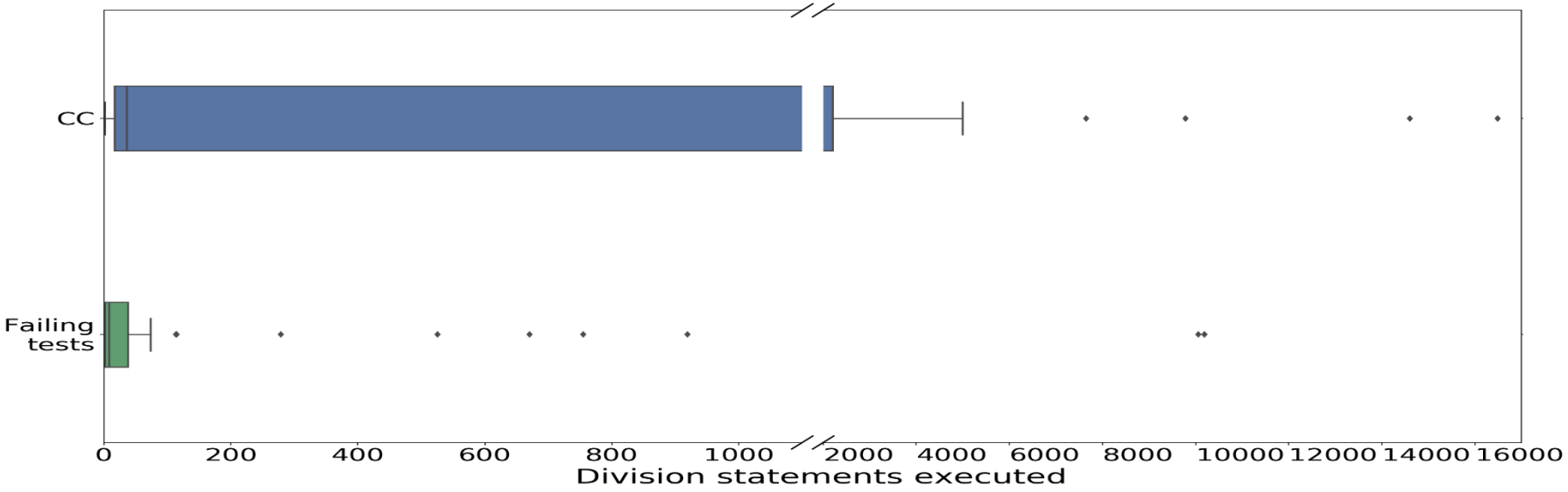




Note: some outliers have been omitted from the bottom graph for visualization purposes

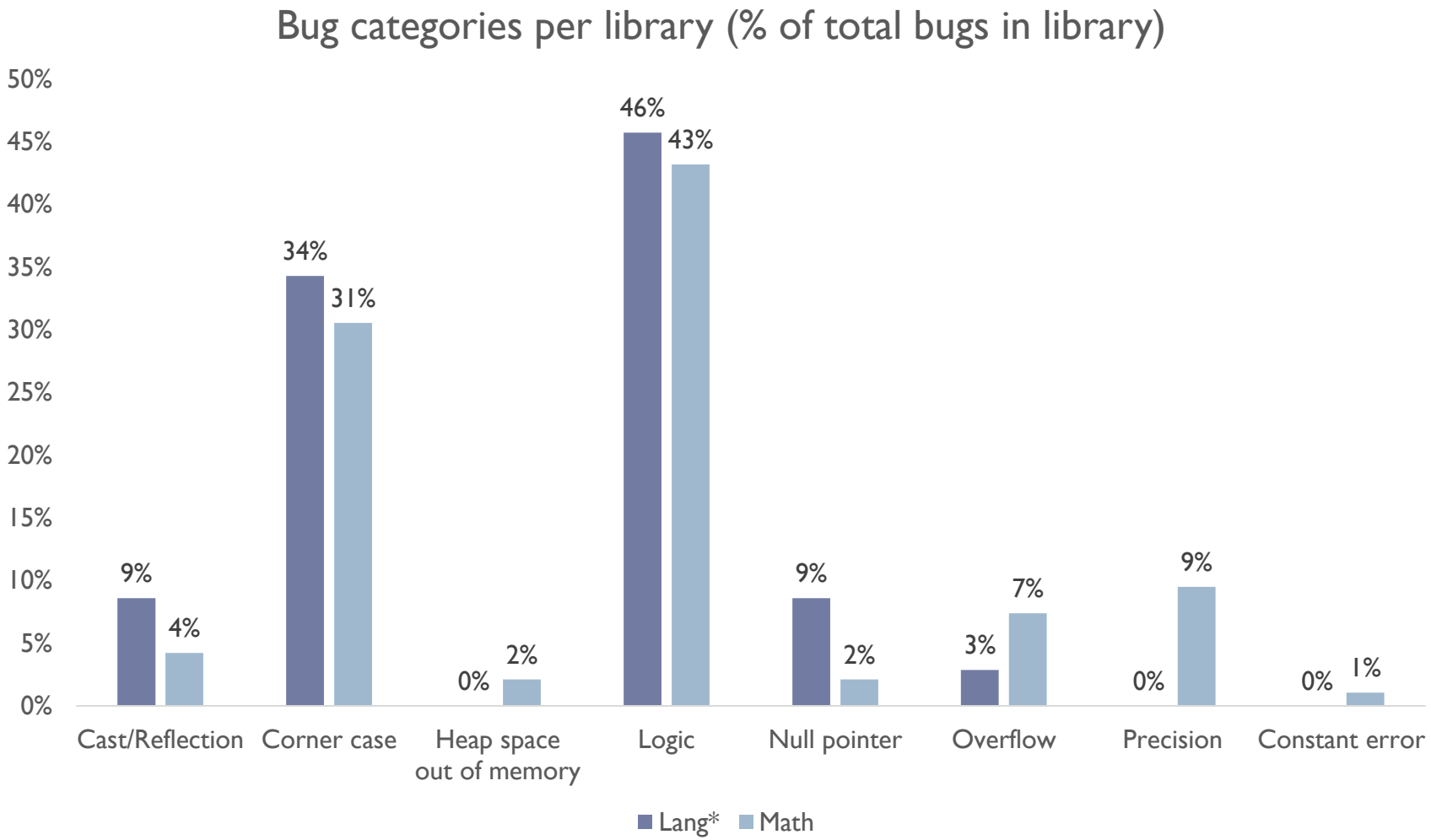
Math Library CC analysis: method calls





Note: some outliers have been omitted from the bottom graph for visualization purposes

Bug Classification



- Lang analysis includes bugs 34 to 65 only

Logic Error Example (40%+)

```
double sumWts = 0;     // Buggy
for (int i = 0; i < weights.length; i++) {
      sumWts += weights[i];
}

double sumWts = 0;    //  Fixed
for (int i = begin; i < begin + length; i++) {
       sumWts += weights[i];
}
```

```
double sumWts = 0;     // Added Oracles
double oracleSumWts = 0;
for (int i = 0; i < weights.length; i++) {
    sumWts += weights[i];
    if (i >= begin && i < (begin+length)) {
                oracleSumWts += weights[i];
   }
}
 System.out.println("\nWeak Oracle 41");
 if (Double.compare(sumWts, oracleSumWts) != 0)
{
     System.out.println("\nStrong Oracle 41");
}
```

Corner Case Error Example (30%+)

```
double foo(double[] a, double[] b) { // Buggy
  final int len = a.length;
  final double[] prodHigh = new double[len];
```

```
double foo(double[] a, double[] b) { // Fixed
  final int len = a.length;
  if (len == 1) {
     // Revert to scalar multiplication.
     return a[0] * b[0];
  }
  final double[] prodHigh = new double[len];
```

```
double foo(double[] a, double[] b) { // Added Oracles
  final int len = a.length;
  System.out.println("\nWeak Oracle 3");
  if (len == 1) {
   System.out.println("\nStrong Oracle 3");
  }

  final double[] prodHigh = new double[len];
```

Null Pointer Check Example (10%+)

```
for (int i = 0; i < sList.length; i++) { // Buggy

   greater = rList[i].length() - sList[i].length();
   …
}
```

```
for (int i = 0; i < sList.length; i++) { // Fixed
   if (sList[i] == null || rList[i] == null) {
       continue;
   }
   greater = rList[i].length() - sList[i].length();
   …
}
```

```
for (int i = 0; i < sList.length; i++) { // Added Oracles
   System.out.println("\nWeak Oracle 39");
   if (sList[i] == null || rList[i] == null) {
       System.out.println("\nStrong Oracle 39");
   }
   greater = rList[i].length() - sList[i].length();
   …
}
```

Is Coincidental Correctness Less Prevalent in Unit Testing?

Prevalent?

YES

Less Prevalent than in other Higher Levels of Testing?

Don't Know Yet