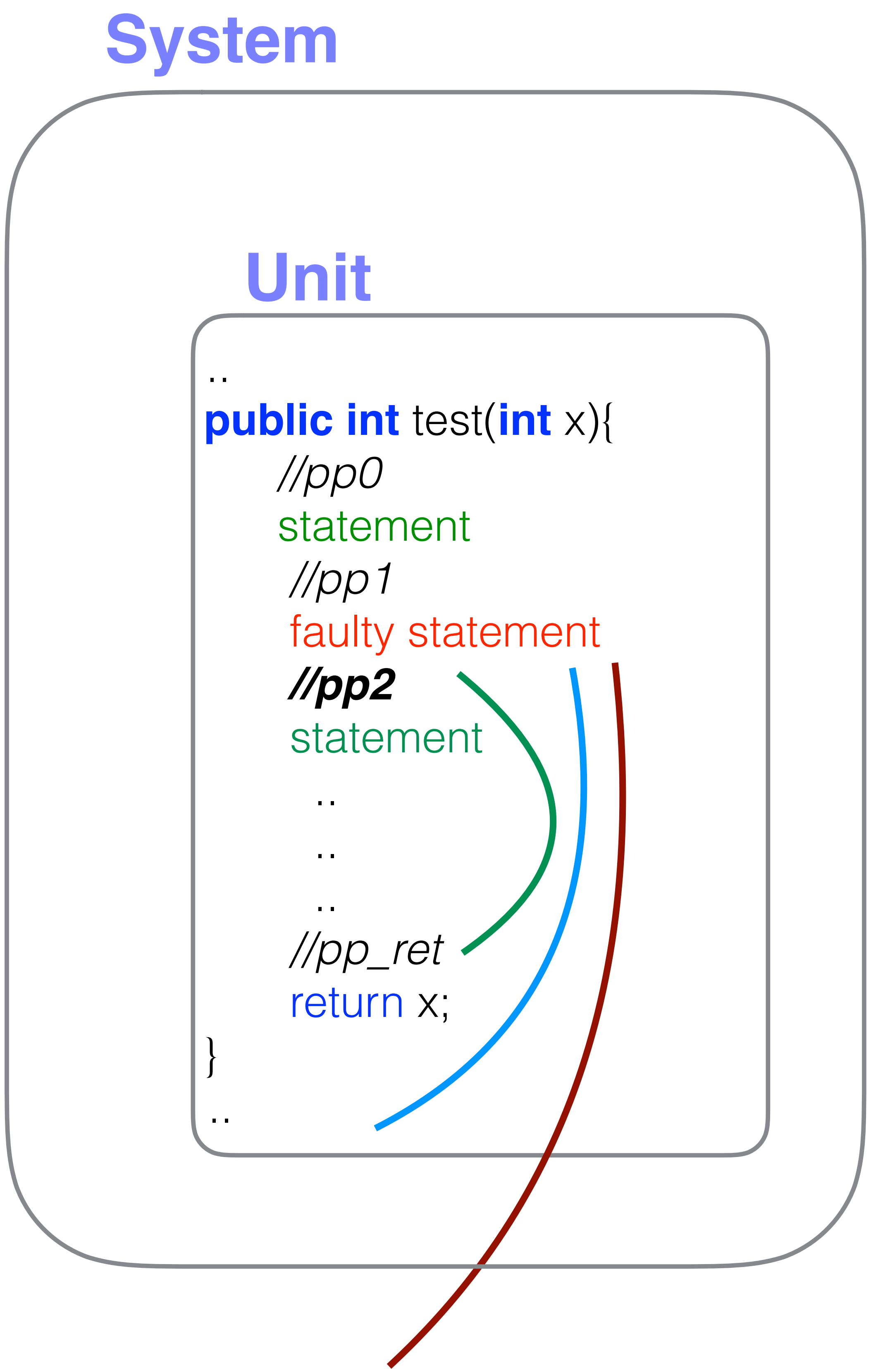
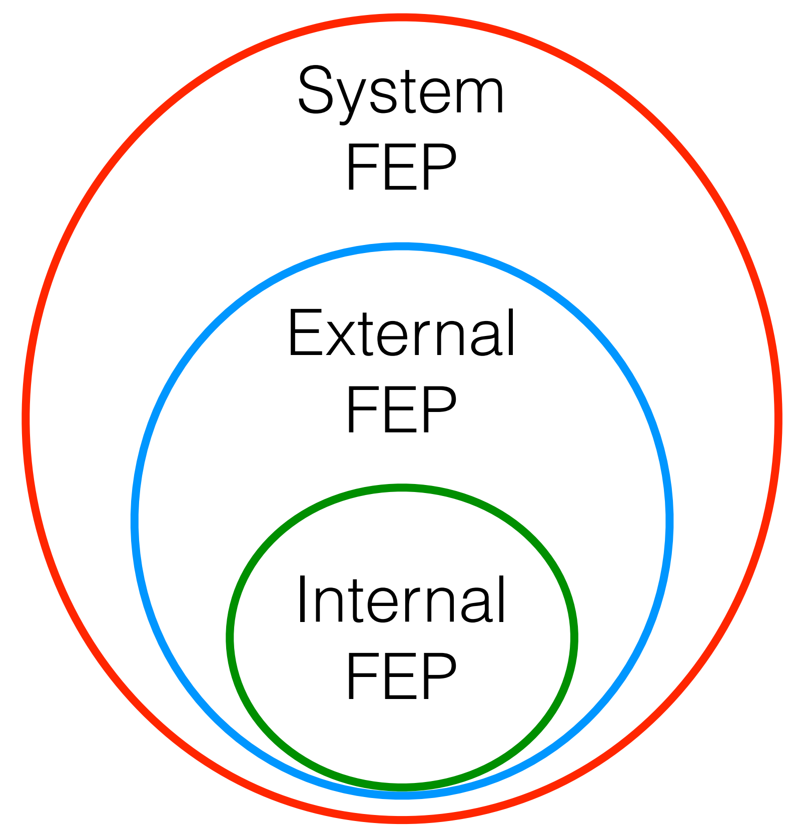
An Empirical Study on Failed Error Propagation in Java Programs with Real Faults

Gunel Jahangirova, David Clark, Mark Harman and Paolo Tonella

30th of January
57th CREST Open Workshop
UCL, London, UK

Failed Error Propagation





Experimental Procedure

Subjects
6 projects from Defects4J benchmark:
395 real bugs, 528 methods

Trace Alignment
Tree Edit Distance algorithm applied to ASTs of fixed and buggy methods

Test Case Generation
1000 executions that cover faulty statement

Measure FEP

Benchmark: Defects4J

Project	Number of Bugs	Bugs with fix in M/C	Number of M/C	M/C > 1 LOC
JFreeChart	26	26	39	39
Closure Compiler	133	131	172	172
Commons Lang	65	61	88	82
Commons Math	106	104	146	135
Mockito	38	37	76	50
Joda Time	27	26	51	50

384 bugs **572 M/C**

Test Case Generation

```
int test(int x) {
    x = 3 * x;
    if (x > 0) {
        x = x % 4;
    } else {
        x = x + 1;
    }
    return x;
}
```

```
int test(int x) {
    x = 3 * x;
    if (x > 0) {
        x = x % 3;
    } else {
        x = x + 1;
    }
    return x;
}
```

EVOSUITE

Line Coverage Criteria:

line_list

goals_multiply

Budget: 10000 seconds

Trace Alignment

```java
    public Complex reciprocal() {
        if (isNaN) {
            return NaN;
        }

        if (real == 0.0 && imaginary == 0.0) {
            return NaN;
        }

        if (isInfinite) {
            return ZERO;
        }

        if (FastMath.abs(real) < FastMath.abs(imaginary)) {
            double q = real / imaginary;
            double scale = 1. / (real * q + imaginary);
            return createComplex(scale * q, -scale);
        } else {
            double q = imaginary / real;
            double scale = 1. / (imaginary * q + real);
            return createComplex(scale, -scale * q);
        }
    }
```

```java
    public Complex reciprocal() {
        if (isNaN) {
            return NaN;
        }

        if (real == 0.0 && imaginary == 0.0) {
            return INF;
        }

        if (isInfinite) {
            return ZERO;
        }

        if (FastMath.abs(real) < FastMath.abs(imaginary)) {
            double q = real / imaginary;
            double scale = 1. / (real * q + imaginary);
            return createComplex(scale * q, -scale);
        } else {
            double q = imaginary / real;
            double scale = 1. / (imaginary * q + real);
            return createComplex(scale, -scale * q);
        }
    }
```

Trace Alignment

```java
public Line revert() {
    final Line reverted = new Line(zero, zero.subtract(direction));
    return reverted;
}
```

```java
public Line revert() {
    final Line reverted = new Line(this);
    reverted.direction = reverted.direction.negate();
    return reverted;
}
```

```java
public Dfp multiply(final int x) {
        return multiplyFast(x);
}
```

```java
public Dfp multiply(final int x) {
    if (x >= 0 && x < RADIX) {
        return multiplyFast(x);
    } else {
        return multiply(newInstance(x));
    }
}
```

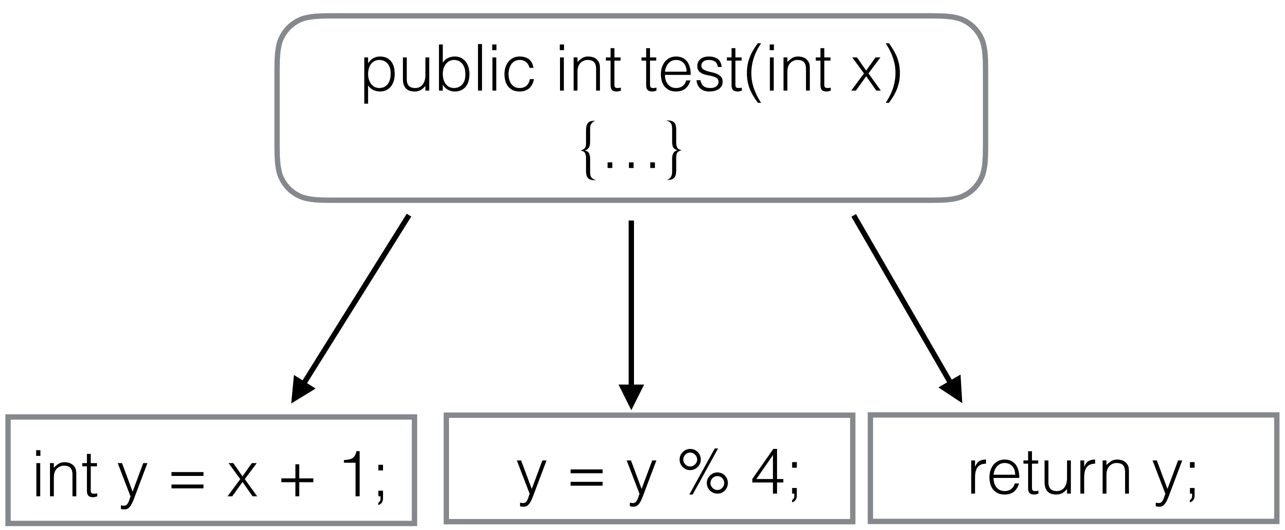
```java
public boolean isFeasible(final double[] x) {
    if (boundaries == null) {
        return true;
    }

    for (int i = 0; i < x.length; i++) {
        if (x[i] < 0) {
            return false;
        }
        if (x[i] > 1.0) {
            return false;
        }
    }
    return true;
}
```

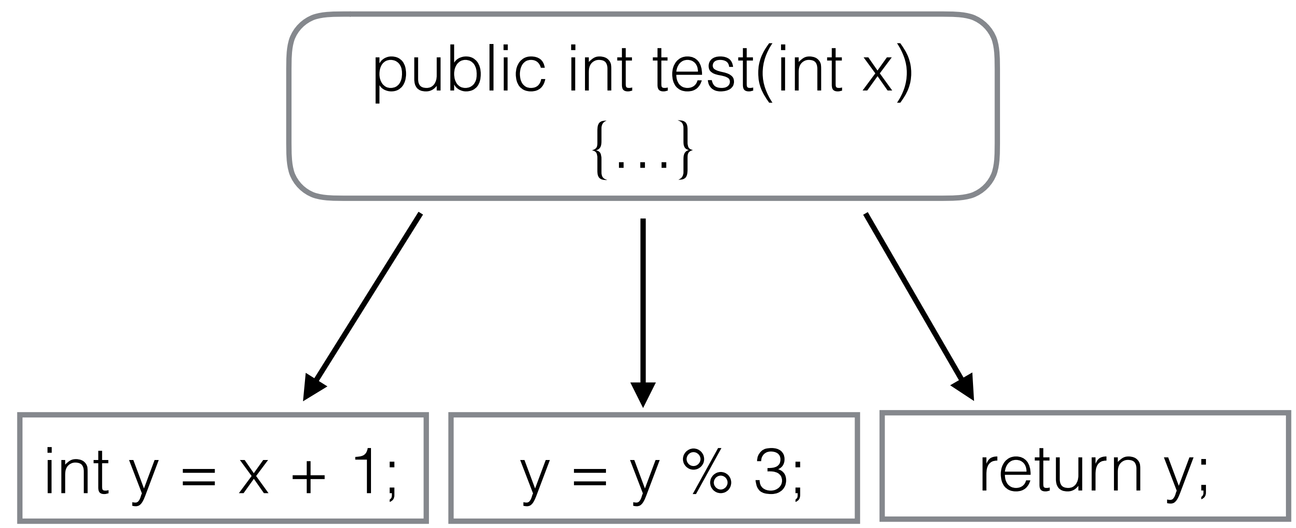
```java
public boolean isFeasible(final double[] x) {
    if (boundaries == null) {
        return true;
    }

    final double[] bLoEnc = encode(boundaries[0]);
    final double[] bHiEnc = encode(boundaries[1]);

    for (int i = 0; i < x.length; i++) {
        if (x[i] < bLoEnc[i]) {
            return false;
        }
        if (x[i] > bHiEnc[i]) {
            return false;
        }
    }
    return true;
}
```

Tree Edit Script

```
public int test(int x) {
    int y = x + 1;
    y = y % 4;
    return y;
}
```

```
public int test(int x) {
    int y = x + 1;
    y = y % 3;
    return y;
}
```





KEEP int y = x + 1;

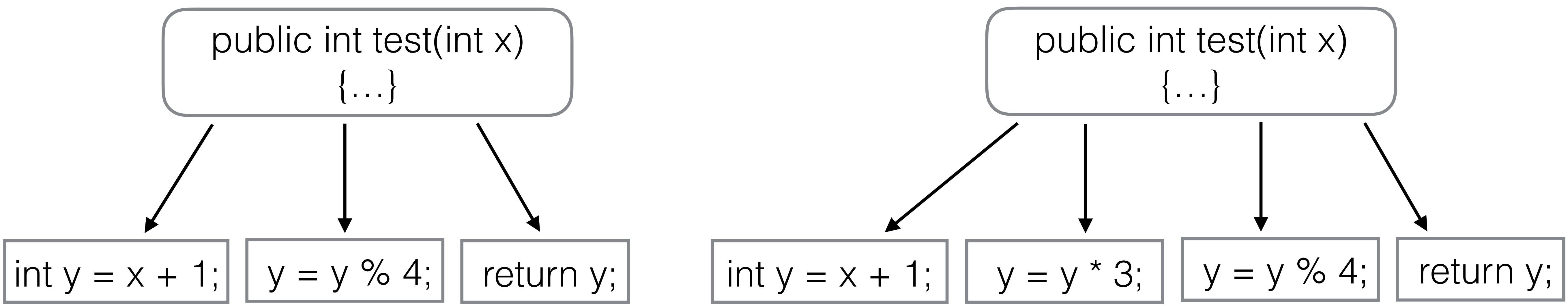
CHANGE y = y % 4; to y = y % 3;

KEEP return y;

Tree Edit Script

```
public int test(int x) {
    int y = x + 1;
    y = y % 4;
    return y;
}
```

```
public int test(int x) {
    int y = x + 1;
    y = y * 3;
    y = y % 4;
    return y;
}
```



KEEP int y = x + 1;

INSERT y = y * 3;

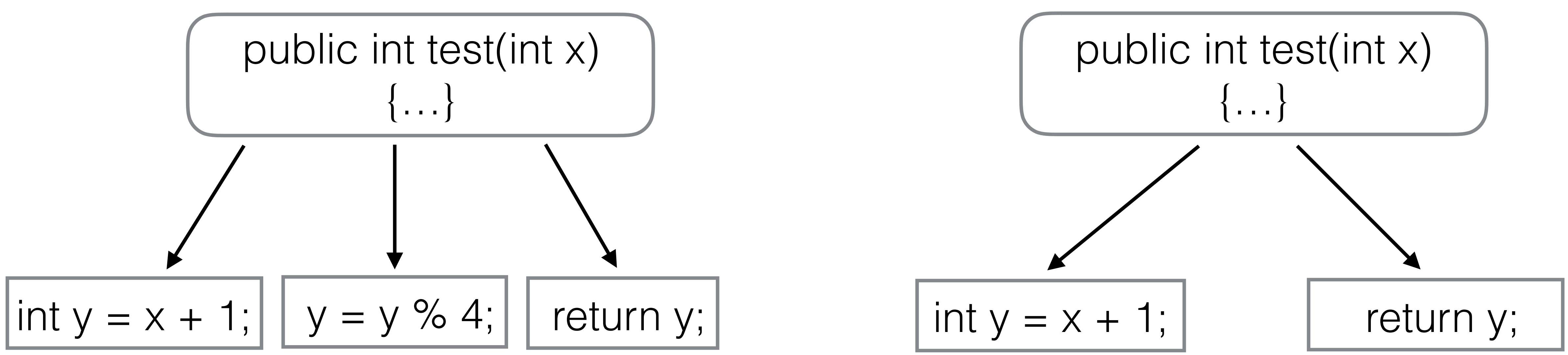
KEEP y = y % 4;

KEEP return y;

Tree Edit Script

```java
public int test(int x) {
    int y = x + 1;
    y = y % 4;
    return y;
}
```

```java
public int test(int x) {
    int y = x + 1;
    return y;
}
```



KEEP int y = x + 1;

DELETE y = y % 4;

KEEP return y;

Program Point Instrumentation

Algorithm 1: Program point instrumentation

Procedure `visit`(n, i)

    Input:
    n: AST node to be visited
    i: instrumentation index

    begin
        if *n is labeled as KEEP or CHANGE $\wedge$ type(n) is not (RETURN or THROW)* **then**
            while *next(n) is labeled as DELETE or INSERT* **do**
                $n :=$ next(n)
            $i := i + 1$
            instrumentAfter(n, pp_i)
        else
            visit(next(n), i)
        if *type(n) is not (FOR or WHILE)* **then**
            for *$m \in$ children(n)* **do**
                visit(m, i)

Program Point Instrumentation

```
public int test(int x) {
    //pp0
    int y = x + 1;
    //pp1
    y = y % 4;
    //pp2
    return y;
}
```

KEEP int y = x + 1;
CHANGE y = y % 4 to
y = y % 3;
KEEP return y;

```
public int test(int x) {
    //pp0
    int y = x + 1;
    //pp1
    y = y % 3;
    //pp2
    return y;
}
```

Program Point Instrumentation

```
public int test(int x) {
    //pp0
    int y = x + 1;
    //pp1
    y = y % 4;
    //pp2
    return y;
}
```

```
KEEP int y = x + 1;
INSERT y = y * 3;
KEEP y = y % 4;
KEEP return y;
```

```
public int test(int x) {
    //pp0
    int y = x + 1;
    y = y * 3;
    //pp1
    y = y % 4;
    //pp2
    return y;
}
```

Program Point Instrumentation

```
public int test(int x) {
    //pp0
    int y = x + 1;
    y = y % 4;
    //pp1
    return y;
}
```

```
KEEP int y = x + 1;
DELETE y = y % 4;
KEEP return y;
```

```
public int test(int x) {
    //pp0
    int y = x + 1;
    //pp1
    return y;
}
```

Measuring FEP

Input:

type = $\langle sys \mid unit \rangle$: type of analysis, system-level or unit-level

out, *out′*: output of the system, used only for system-level analysis

ext, *ext′*: externally observable state after buggy/fixed methods have been executed

$pp = \langle pp_0, \ldots, pp_n \rangle$: program points executed in fixed method

$pp' = \langle pp'_0, \ldots, pp'_k \rangle$: program points executed in buggy method

s, s': state by program point in buggy/fixed methods

Result:

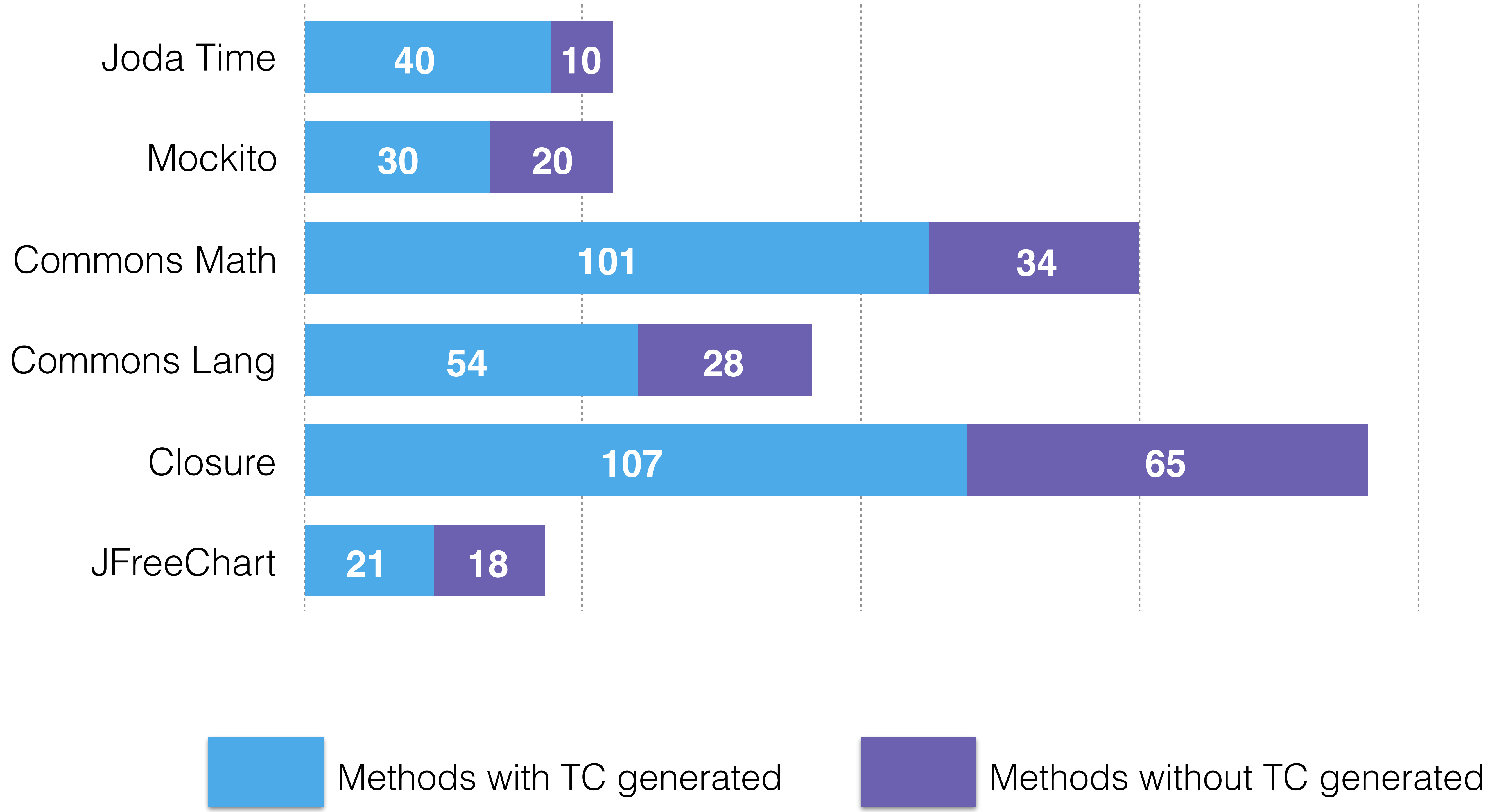
fepType: ⟨ sysFEP | intFEP | extFEP | noFEP ⟩

```
begin
    if type = unit && ext ≠ ext′ then
        return noFEP
    if type = sys then
        if out ≠ out′ then
            return noFEP
        if ext ≠ ext′ then
            return closure(sysFEP, type)
    if s[pp_n] ≠ s′[pp′_k] then
        return closure(extFEP, type)
    if pp ≠ pp′ then
        return closure(intFEP, type)
    for i ∈ [1 : n − 1] do
        if s[pp_i] ≠ s′[pp′_i] then
            return closure(intFEP, type)
    return noFEP
```

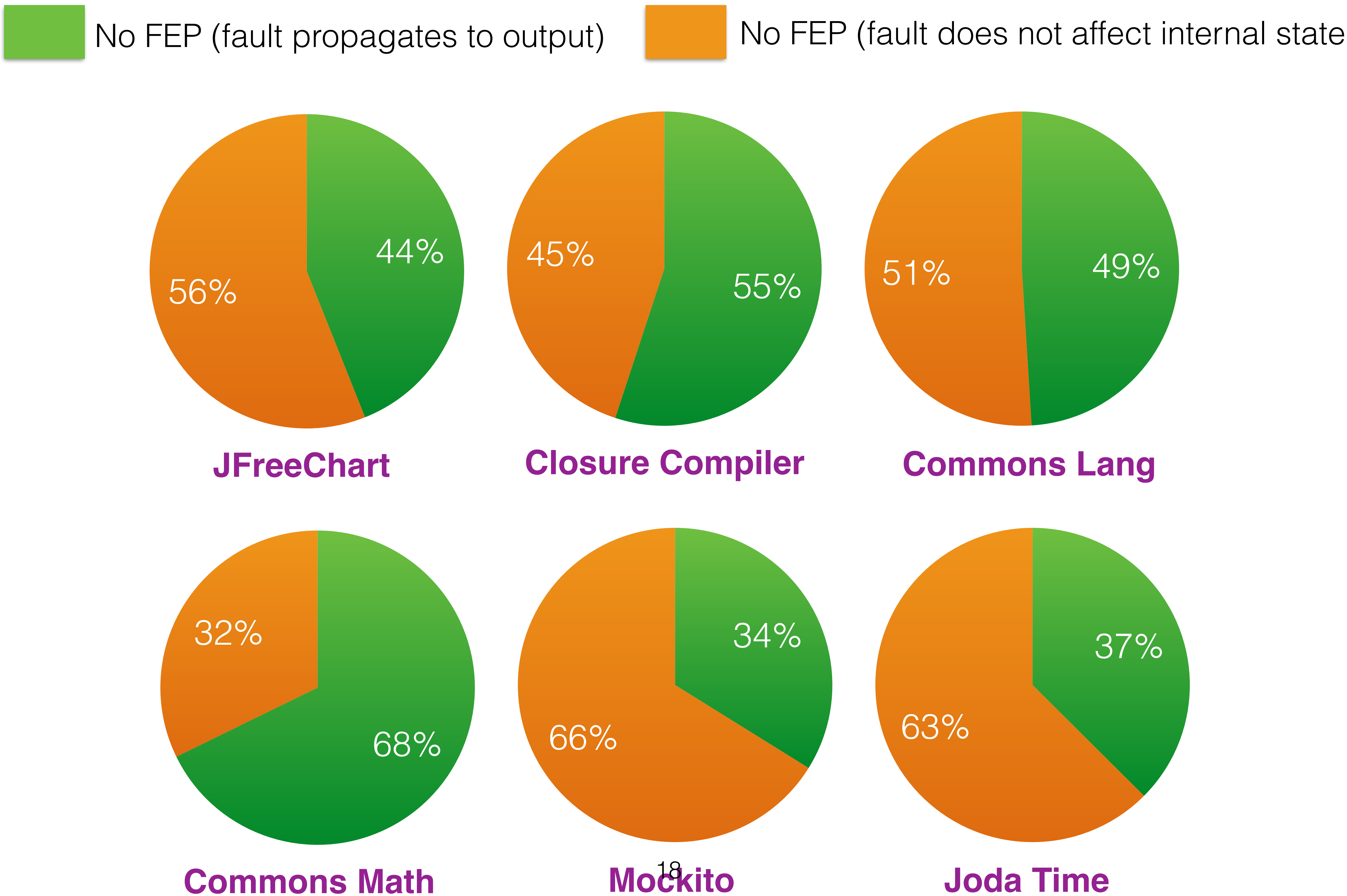
RQ1:

What is the prevalence of unit-level failed error propagation with real faults?

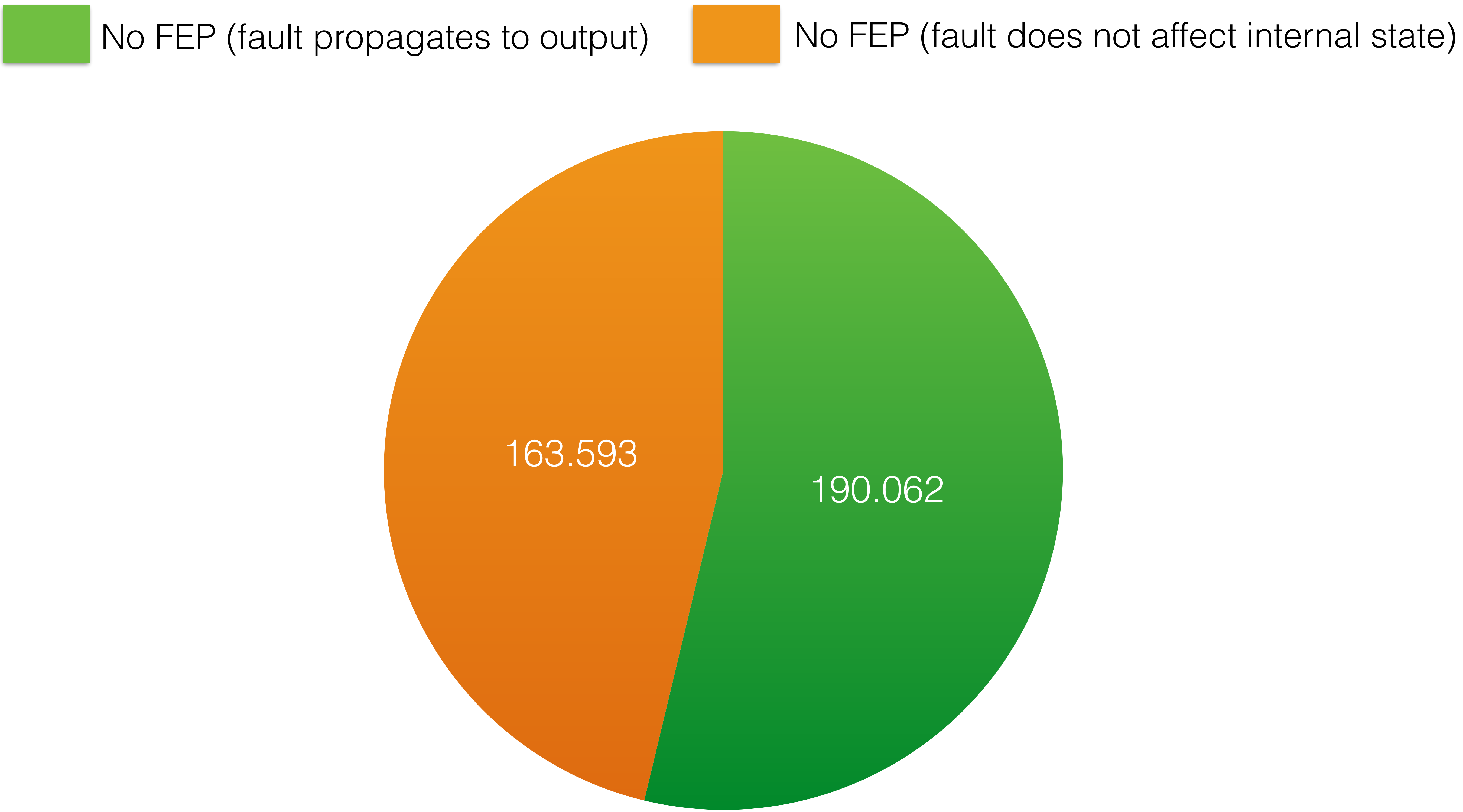
RQ1: TS generation



RQ1: Executions



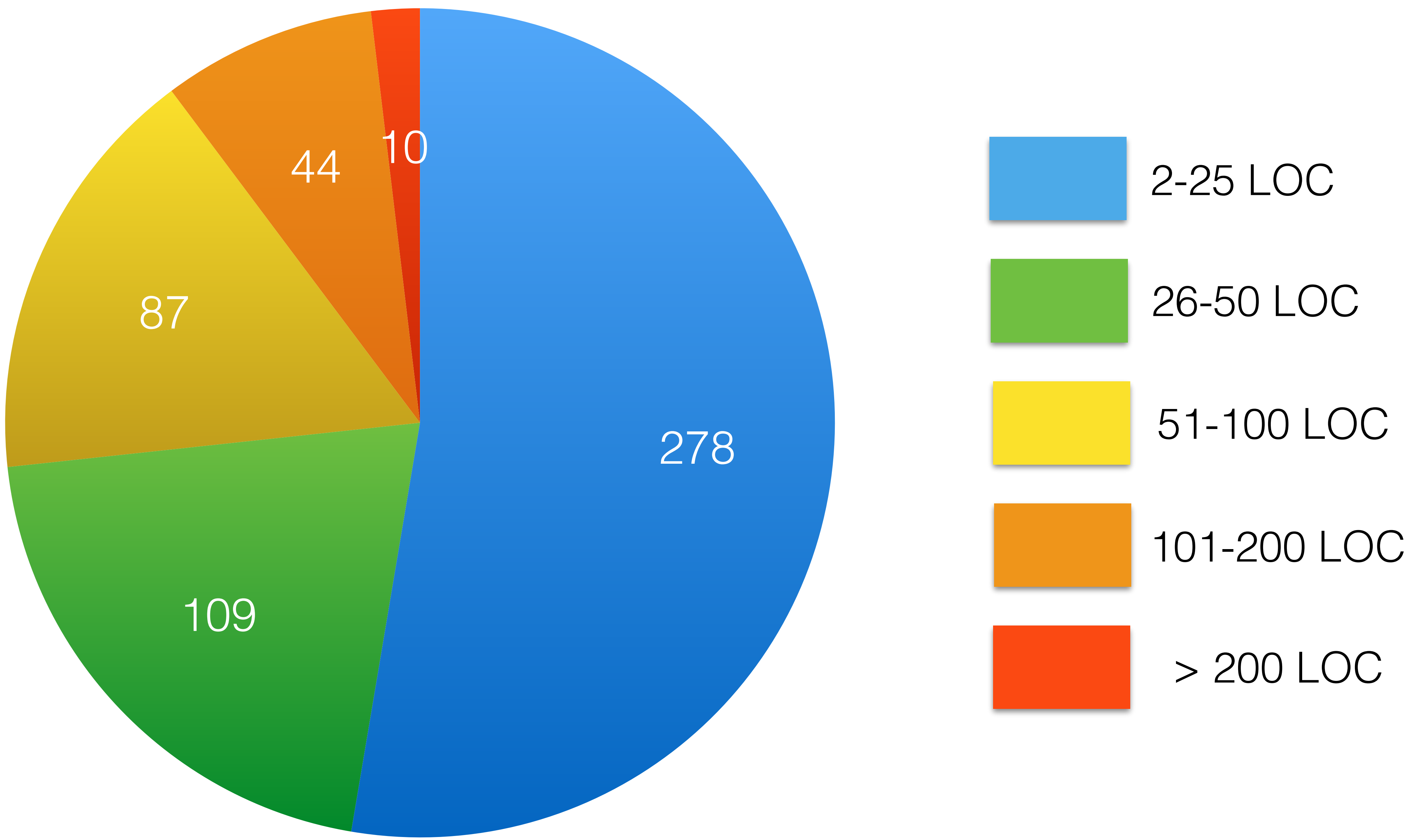
RQ1: Executions



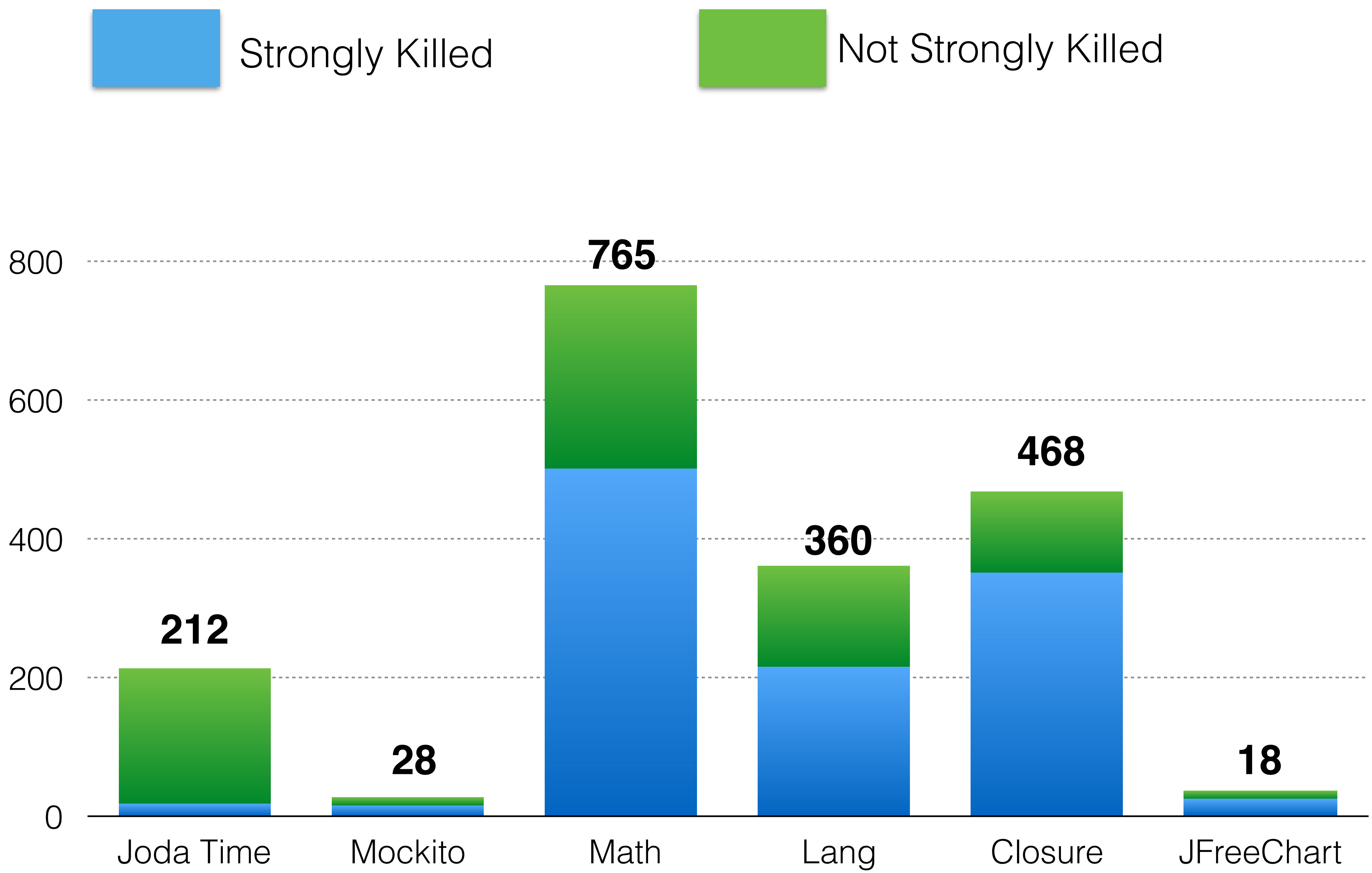
RQ2:

Does the prevalence of unit-level failed error propagation change if real faults are replaced by mutants?

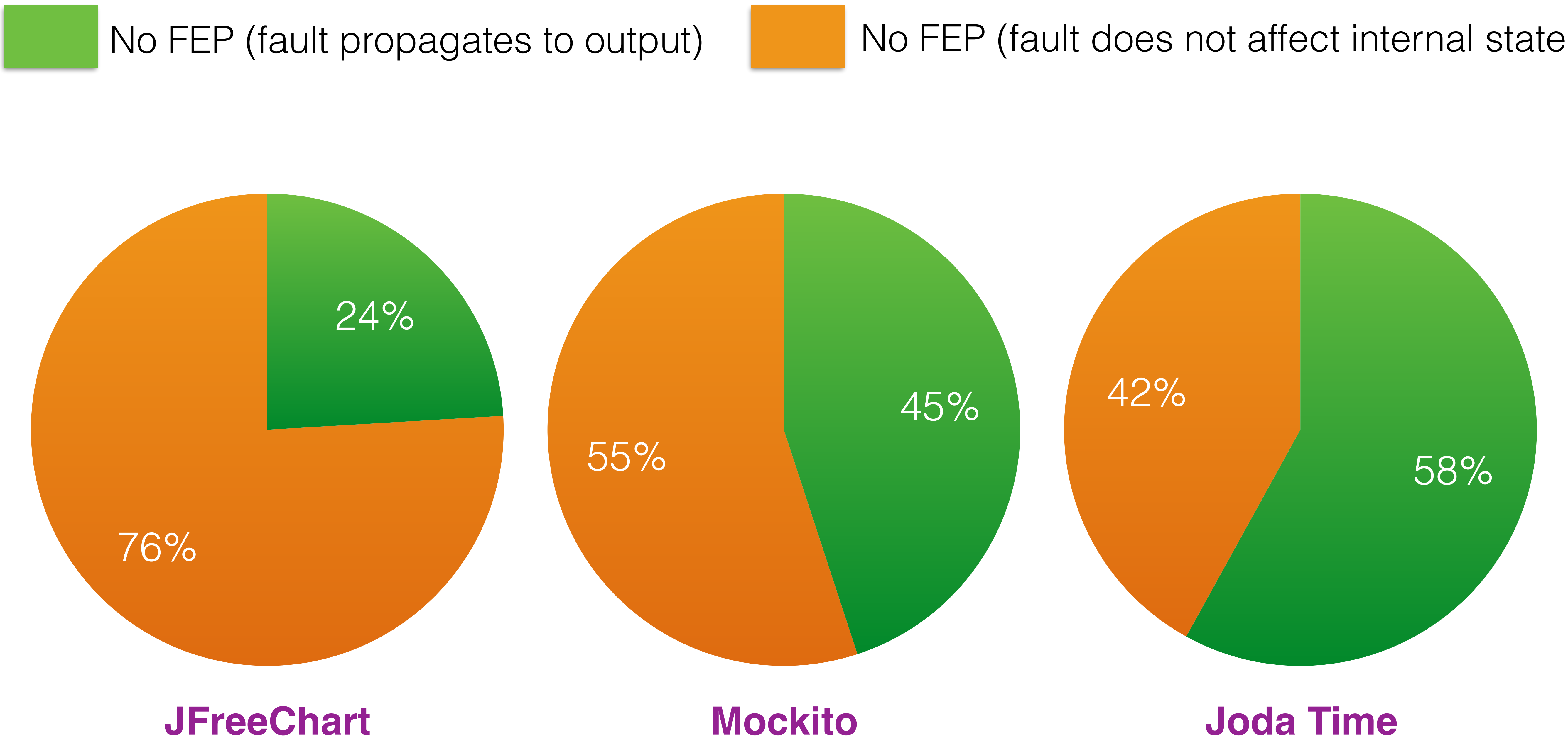
RQ2: Method Groups



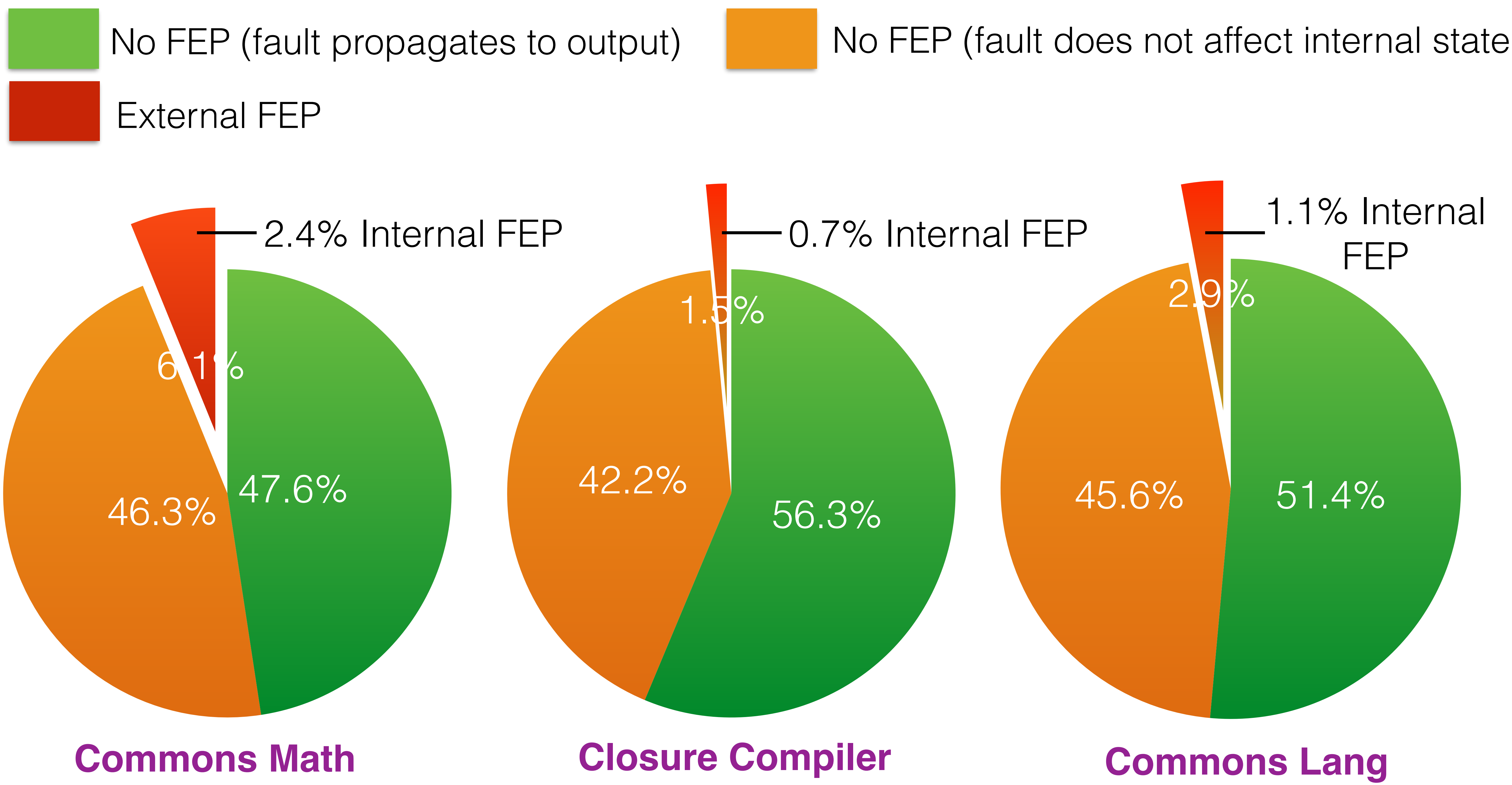
RQ2: Mutations



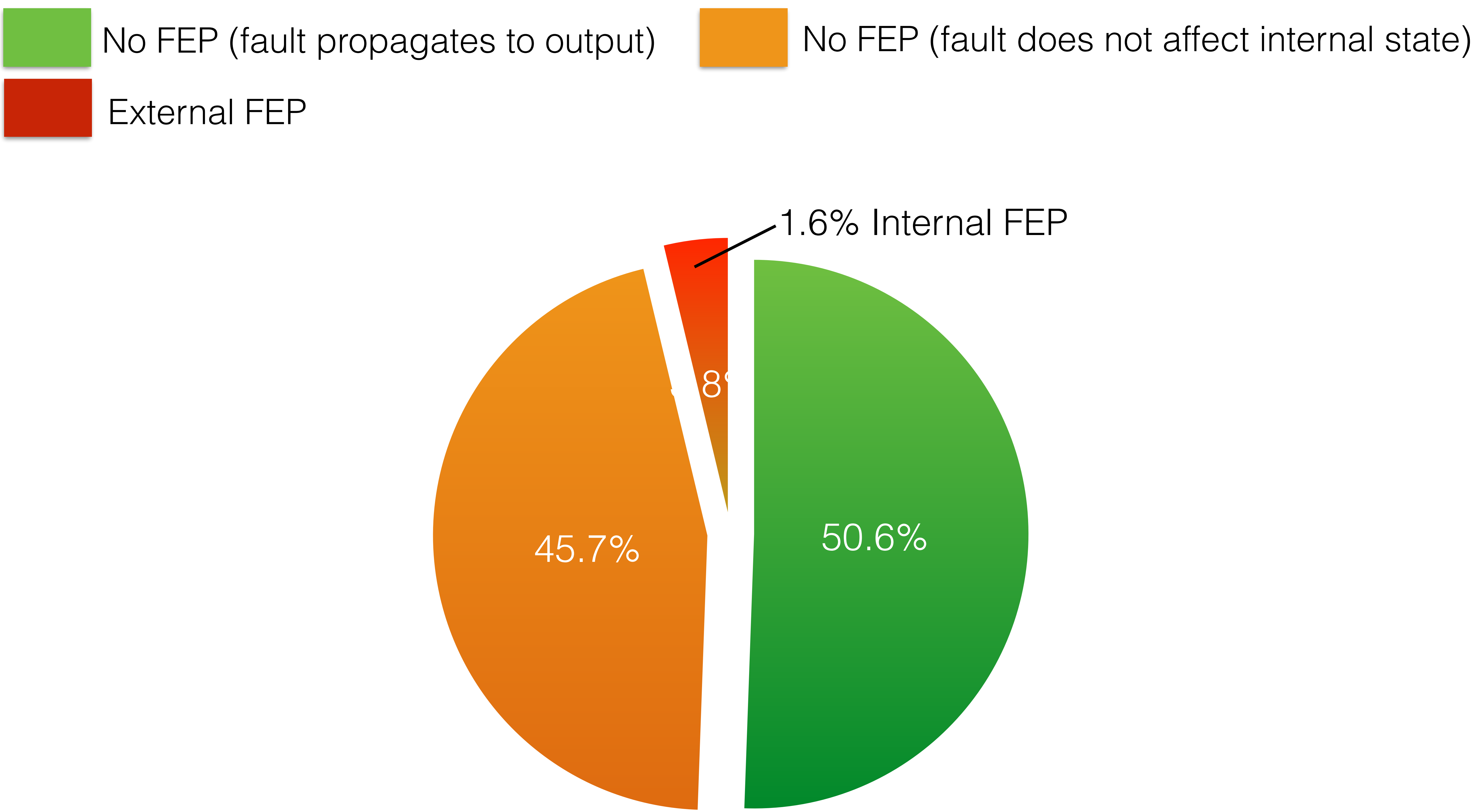
RQ2: Executions



RQ2: Executions



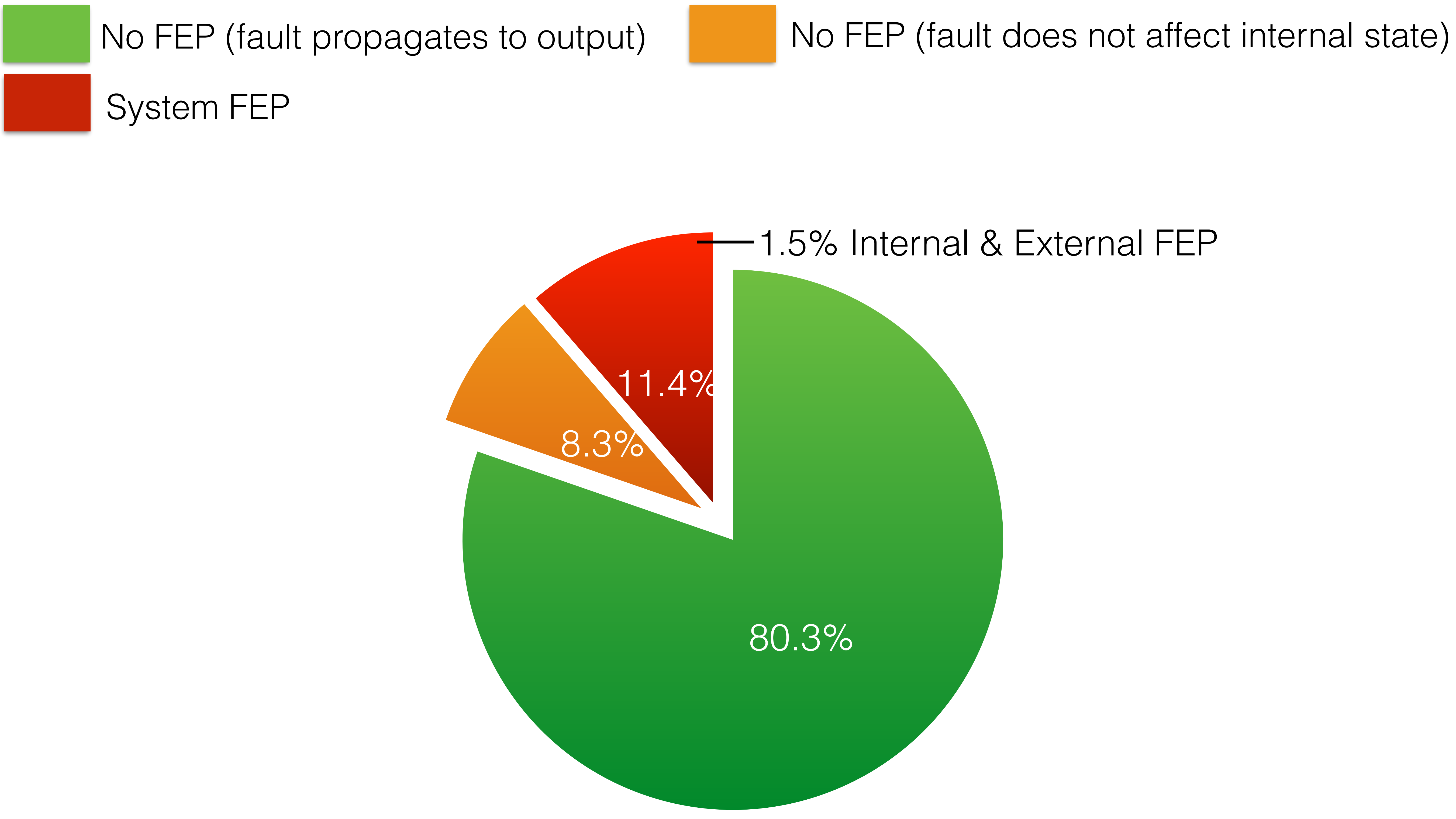
RQ2: Executions



RQ3:

Does the prevalence of failed error propagation with real faults change if it is measured at the system level instead of unit level?

RQ3: Executions



Qualitative Analysis

Manual analysis on 384 bugs from Defects4J and generated mutants

Bug fix affects output directly

State change resulting from the fix is such that it always propagates to output

Bug fix affects output directly

```
public Complex divide(double divisor) {
  if (isNaN || Double.isNaN(divisor)) {
      return NaN;
   }
   if (divisor == 0d) {
      return NaN;
      //return isZero ? NaN : INF;
   }
   if (Double.isInfinite(divisor)) {
       return !isInfinite() ? ZERO : NaN;
    }
   return createComplex(real / divisor,
                              imaginary / divisor);
}
```

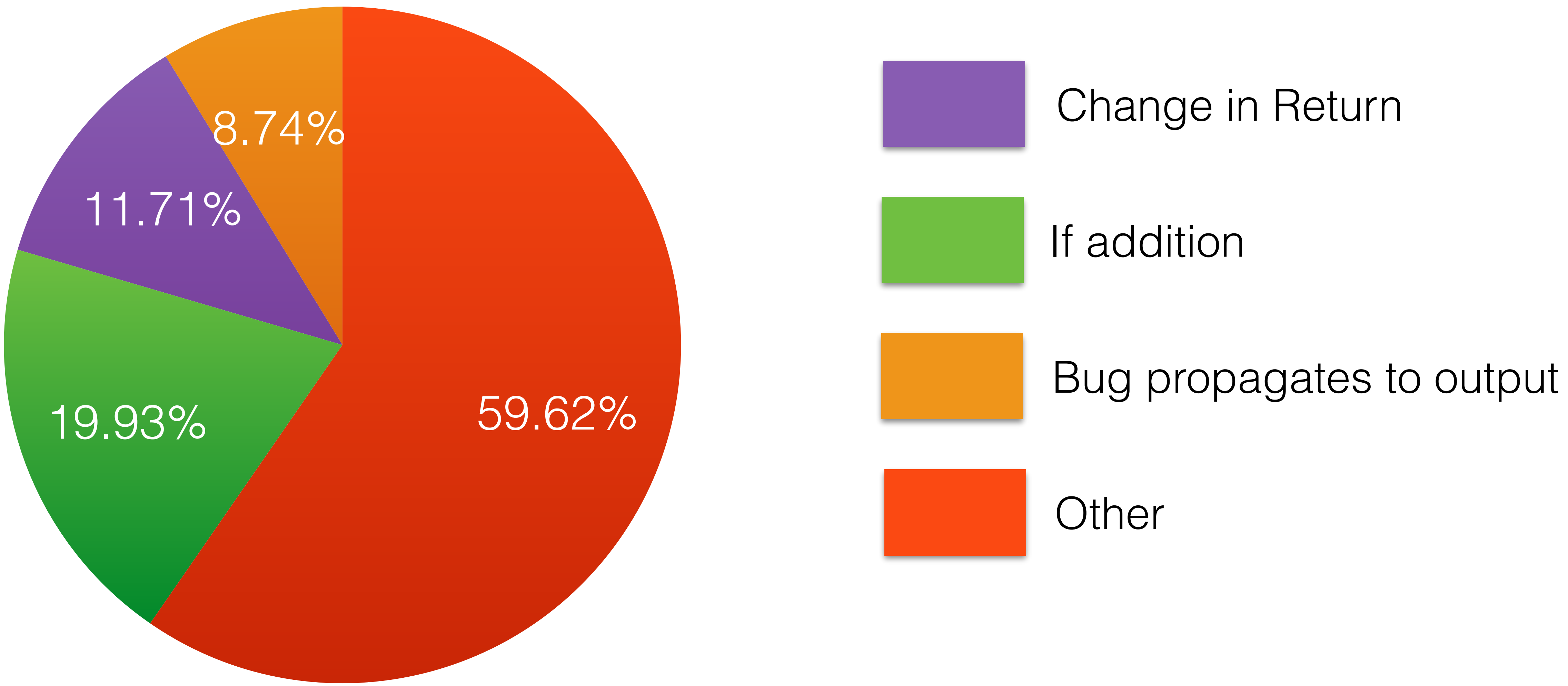
Bug fix affects output directly

```java
public static LocalDate fromDateFields(Date date) {
  if (date == null) {
      throw new IllegalArgumentException
                ("The date must not be null");
   }

   //if (date.getTime() < 0) {
   //    GregorianCal cal = new GregorianCal();
   //    cal.setTime(date);
   //    return fromCalendarFields(cal);
   // }

   return new LocalDate(
      date.getYear() + 1900,
      date.getMonth() + 1,
      date.getDate());
 }
```

Fix propagates to output

```
public double getChiSquare() {
      double chiSquare = 0;
      for (int i = 0; i < rows; ++i) {
          final double residual = residuals[i];
          chiSquare += residual * residual *
                             residualsWeights[i];
          //chiSquare += residual * residual /
          //                 residualsWeights[i];
      }
     return chiSquare;
  }
```

Reasons for the absence of FEP



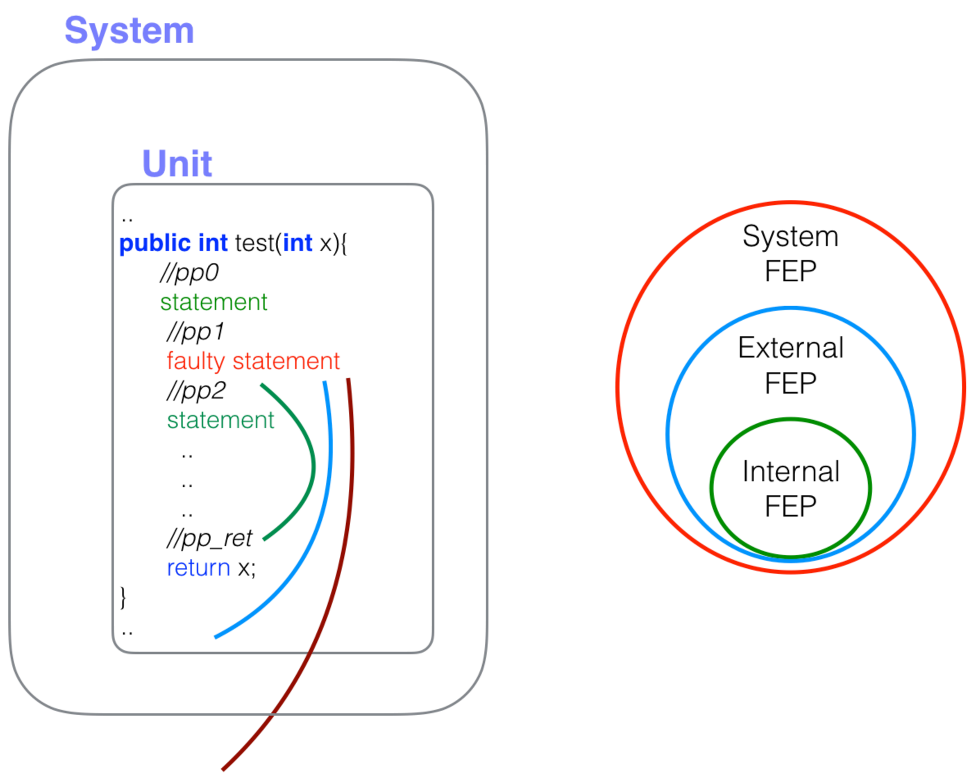
FEP in mutants

```
protected double getInitialDomain(double p) {
    double ret = 0.0;
    //mut0: double ret = 1.0;
    //pp1
    double d = getDenominatorDegreesOfFreedom();

    if (d > 2.0) {
        ret = d / (d - 2.0);
    }
   //pp_ret
    return ret;
}
```

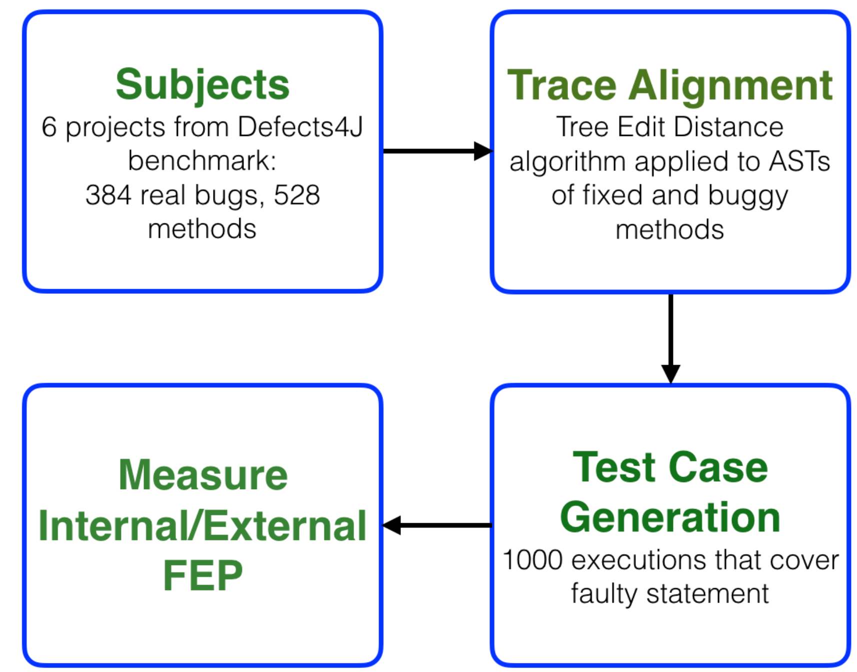
Implications

- Internal Oracles
- Post-conditions
- Subsystem Testing
- Mutants vs. Real Faults

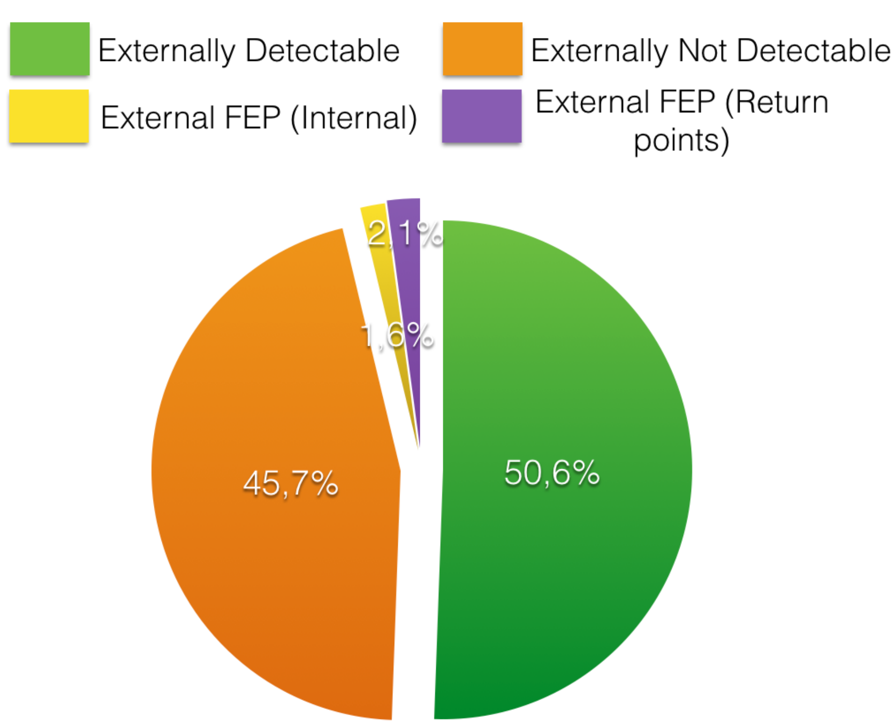
Failed Error Propagation



Experimental Procedure



RQ2: Executions



Reasons for the absence of FEP