Comparing User-Provided Tests to Developer-Provided Tests

René Just, Chris Parnin, Ian Drosos, Michael D. Ernst

ISSTA 2018

User-provided tests

Found in **bug reports**

One small test
Weak or no assertions
High code coverage

Used by **programmers**

Fault localization
5-14% worse
Automated program repair
54-100% worse

Developer-provided tests

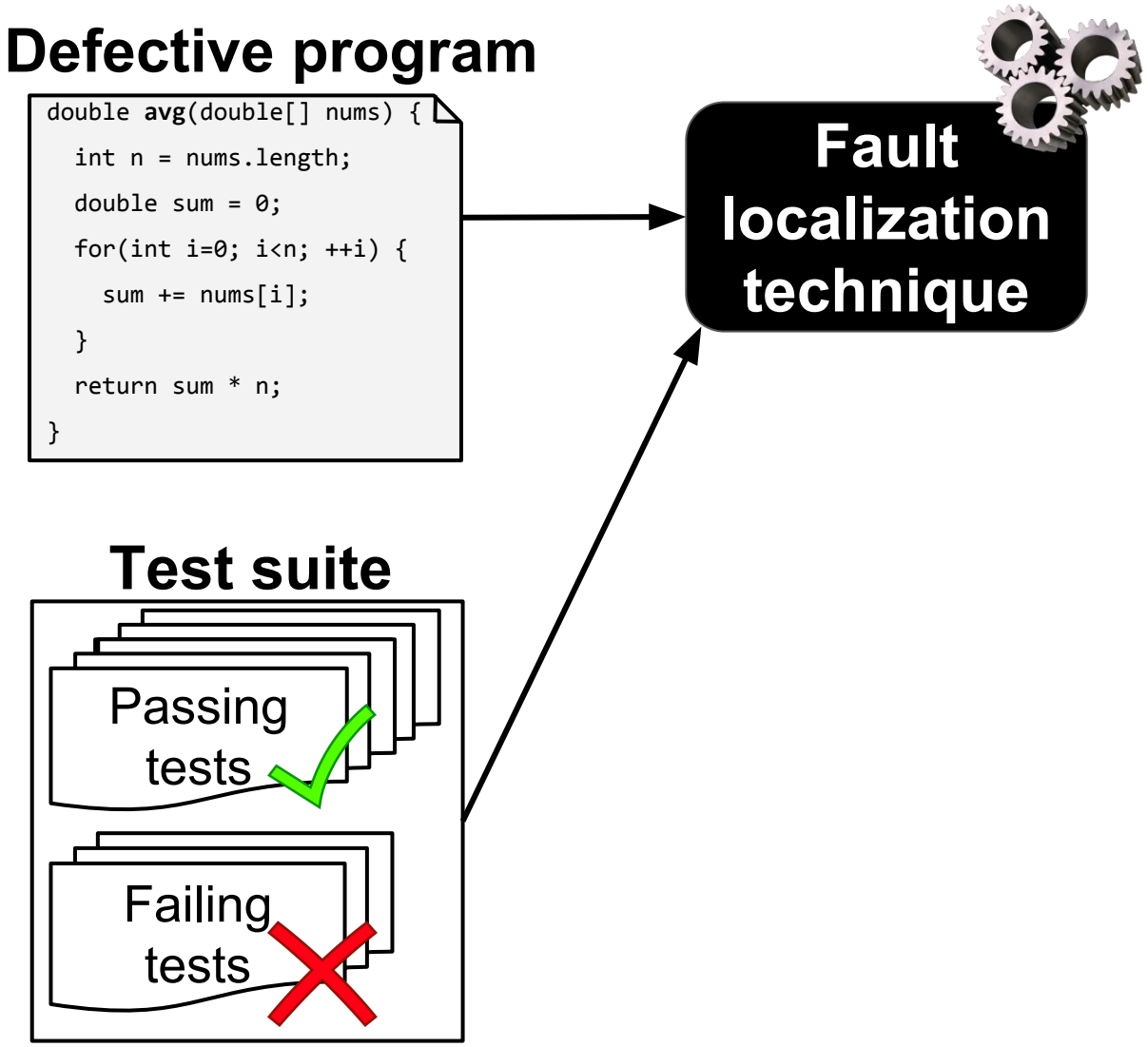
Committed to **repository**

More tests, more LOC
More, stronger assertions
Focused on the defect

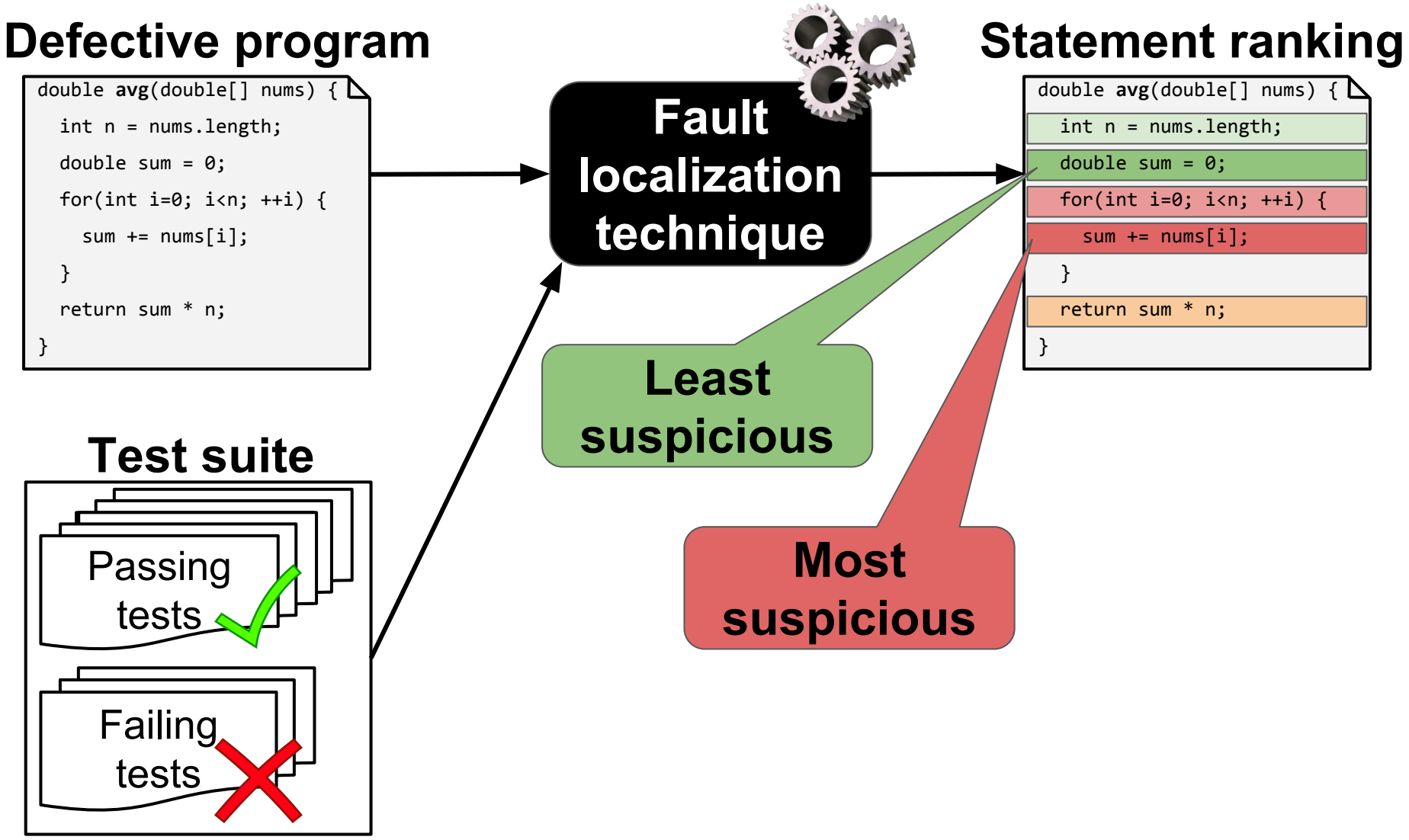
Used in **experiments**

User-provided tests should be used in experiments.

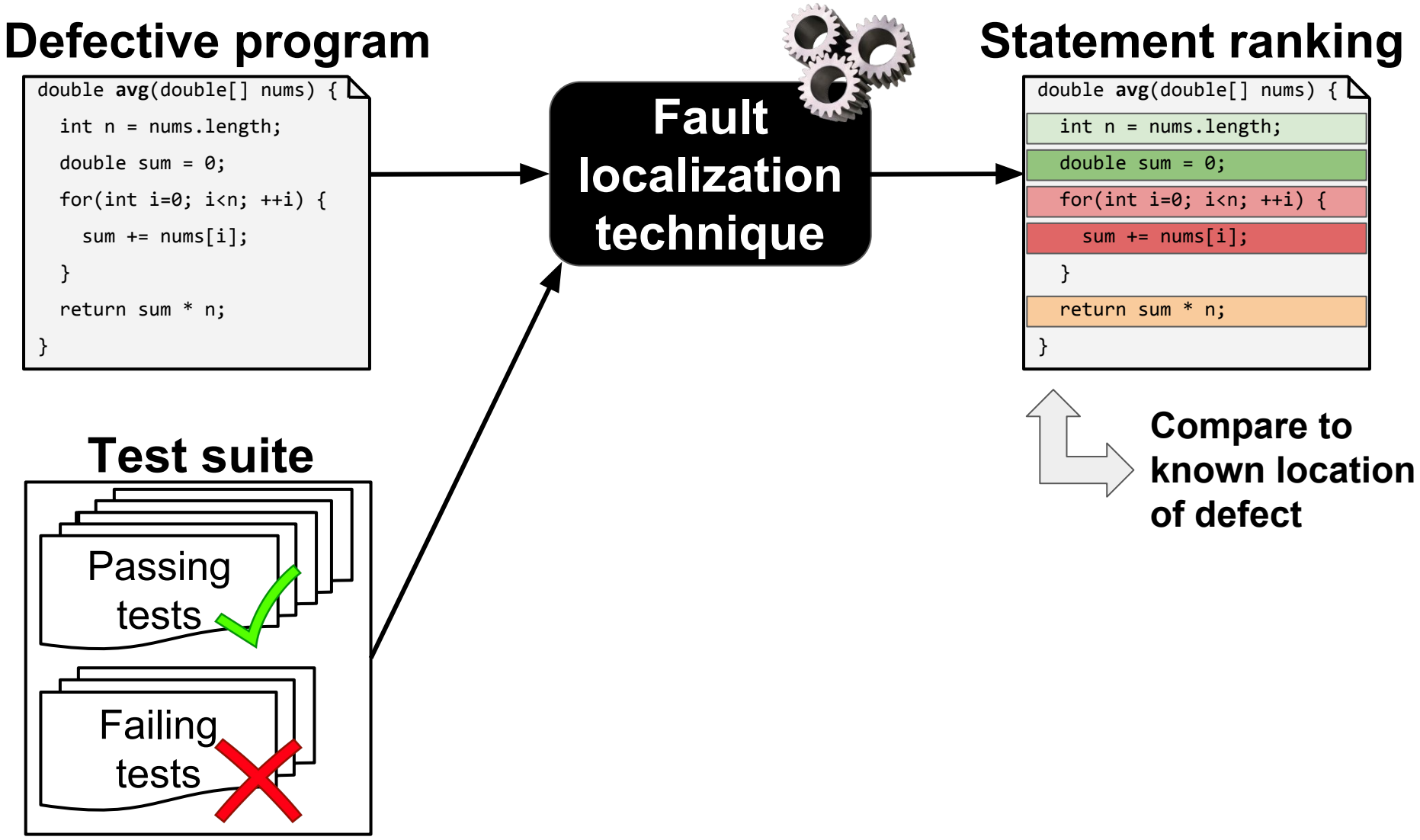
Fault localization: where is the defect?



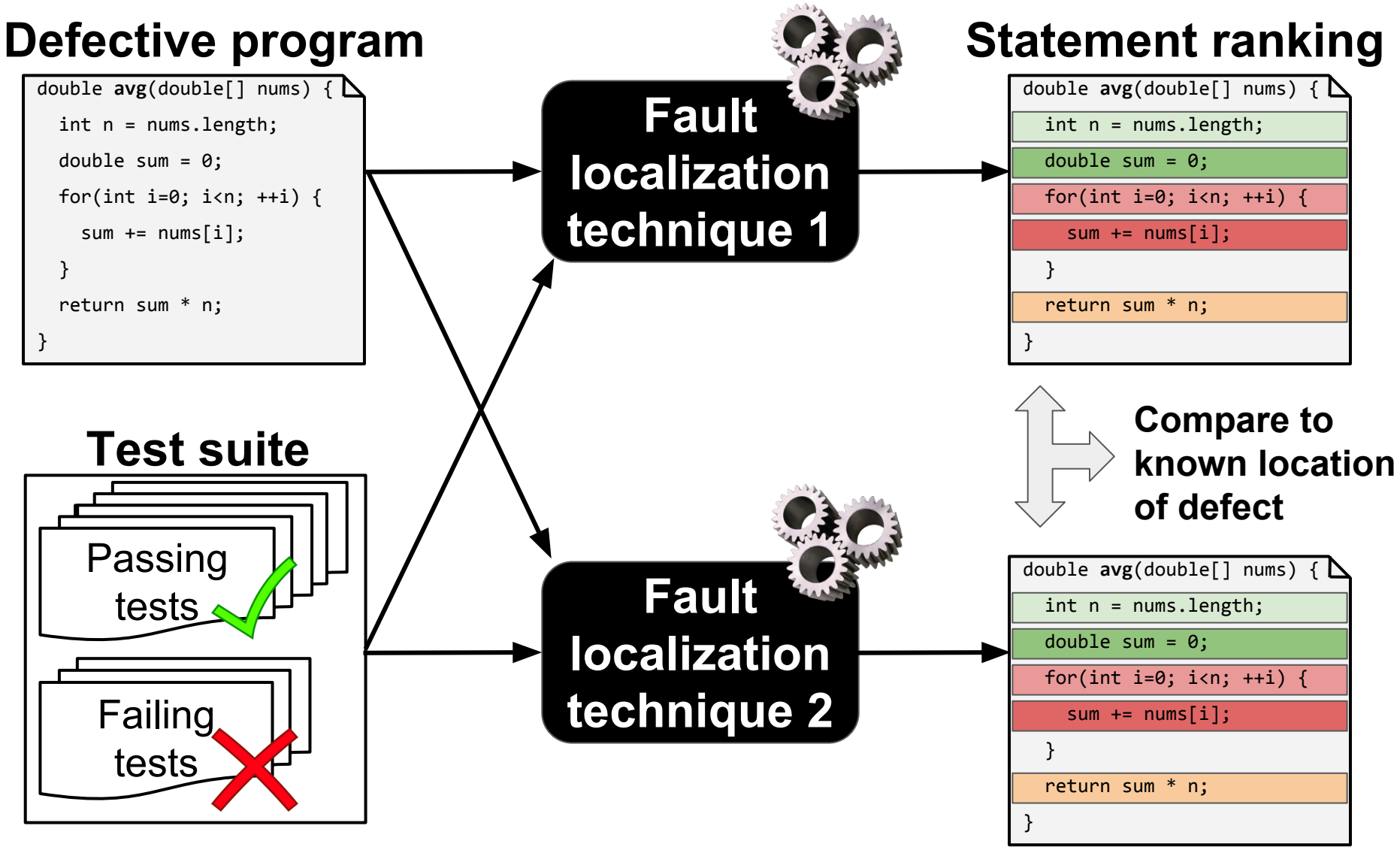
Fault localization: where is the defect?



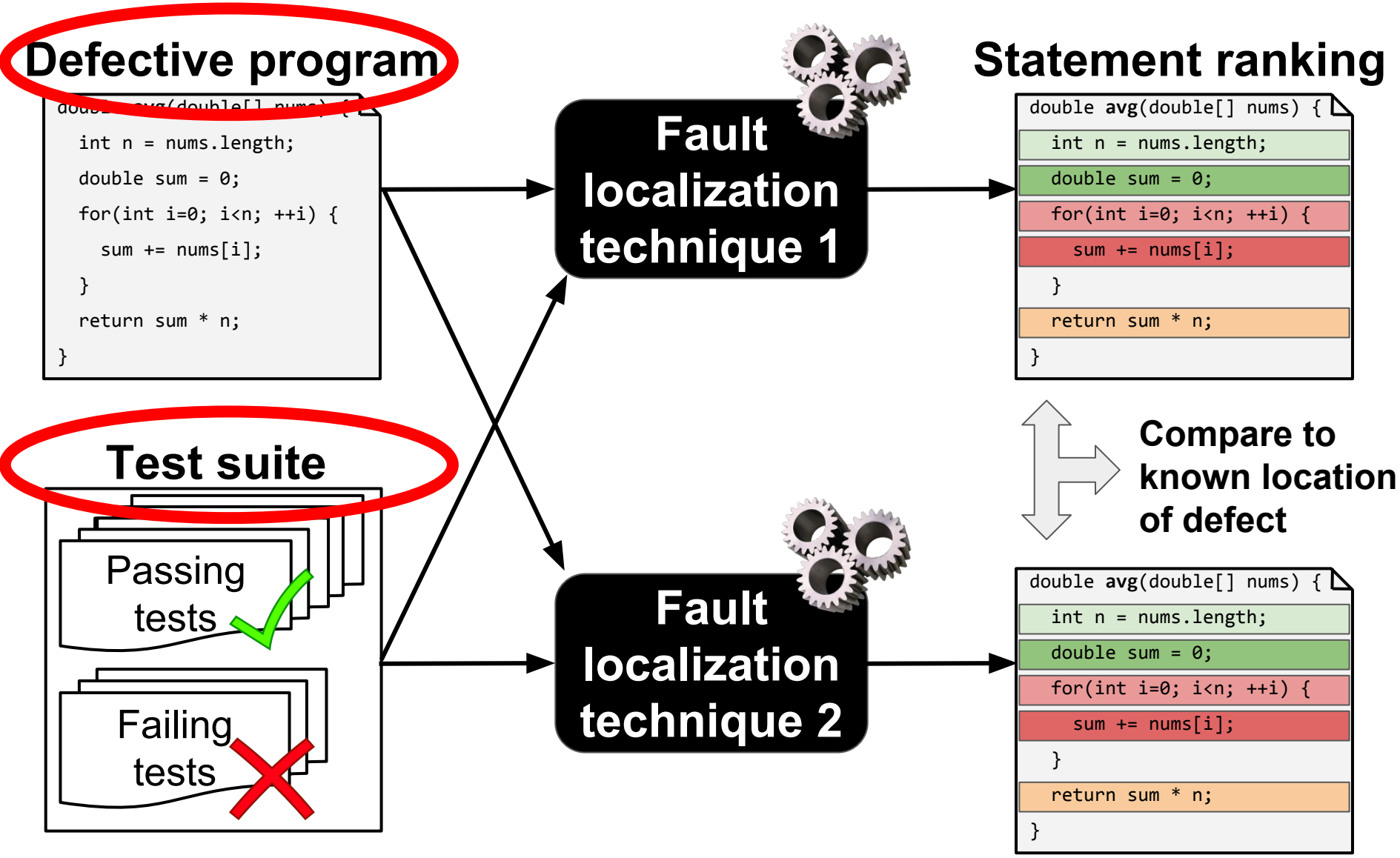
Evaluating fault localization



Evaluating fault localization



Evaluating fault localization

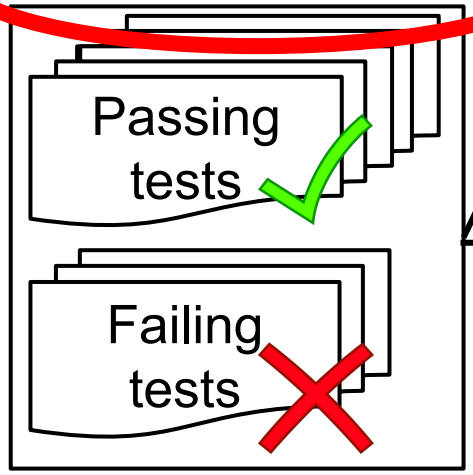


Evaluating fault localization

Defective program

```
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Test suite



Early work

- Artificial defects ("mutants")
  - Easy to create lots of them
  - Known fault locations

Pearson et al. [ICSE 2017]

- 310 real defects (Defects4J)
- 2995 artificial defects

Fault localization technique 2

Compare to known location of defect

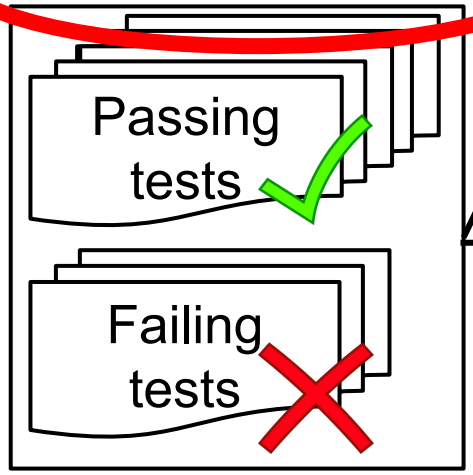
```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Evaluating fault localization

Defective program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Test suite



Early work

- Artificial defects (“mutants”)
  - Easy to create lots of them
  - Known fault locations

Pearson et al. [ICSE 2017]

- 310 real defects (Defects4J)
- 2995 artificial defects

Early work

- Artificial tests
  - Written by researchers
  - Unrealistically strong

Pearson et al. [ICSE 2017]

- Real tests (Defects4J)
  - Written by developers
  - Committed with the fix

Pearson [ICSE 2017]

Comparison of fault localization techniques

	Prior studies (winner > loser)
SBFL vs. SBFL	Ochiai > Tarantula
	Barinel > Ochiai
	Barinel > Tarantula
	Op2 > Ochiai
	Op2 > Tarantula
	DStar > Ochiai
	DStar > Tarantula
MBFL vs. SBFL	Metallaxis > Ochiai
	MUSE > Op2
	MUSE > Tarantula

Pearson [ICSE 2017]

Comparison of fault localization techniques

	Prior studies (winner > loser)	Ours (artificial faults) Replicated	Effect
SBFL vs. SBFL	Ochiai > Tarantula	**yes**	small
	Barinel > Ochiai	**no**	small
	Barinel > Tarantula	**yes**	*negligible*
	Op2 > Ochiai	**yes**	*negligible*
	Op2 > Tarantula	**yes**	small
	DStar > Ochiai	**yes**	*negligible*
	DStar > Tarantula	**yes**	small
MBFL vs. SBFL	Metallaxis > Ochiai	yes	*negligible*
	MUSE > Op2	**no**	*negligible*
	MUSE > Tarantula	**no**	*negligible*

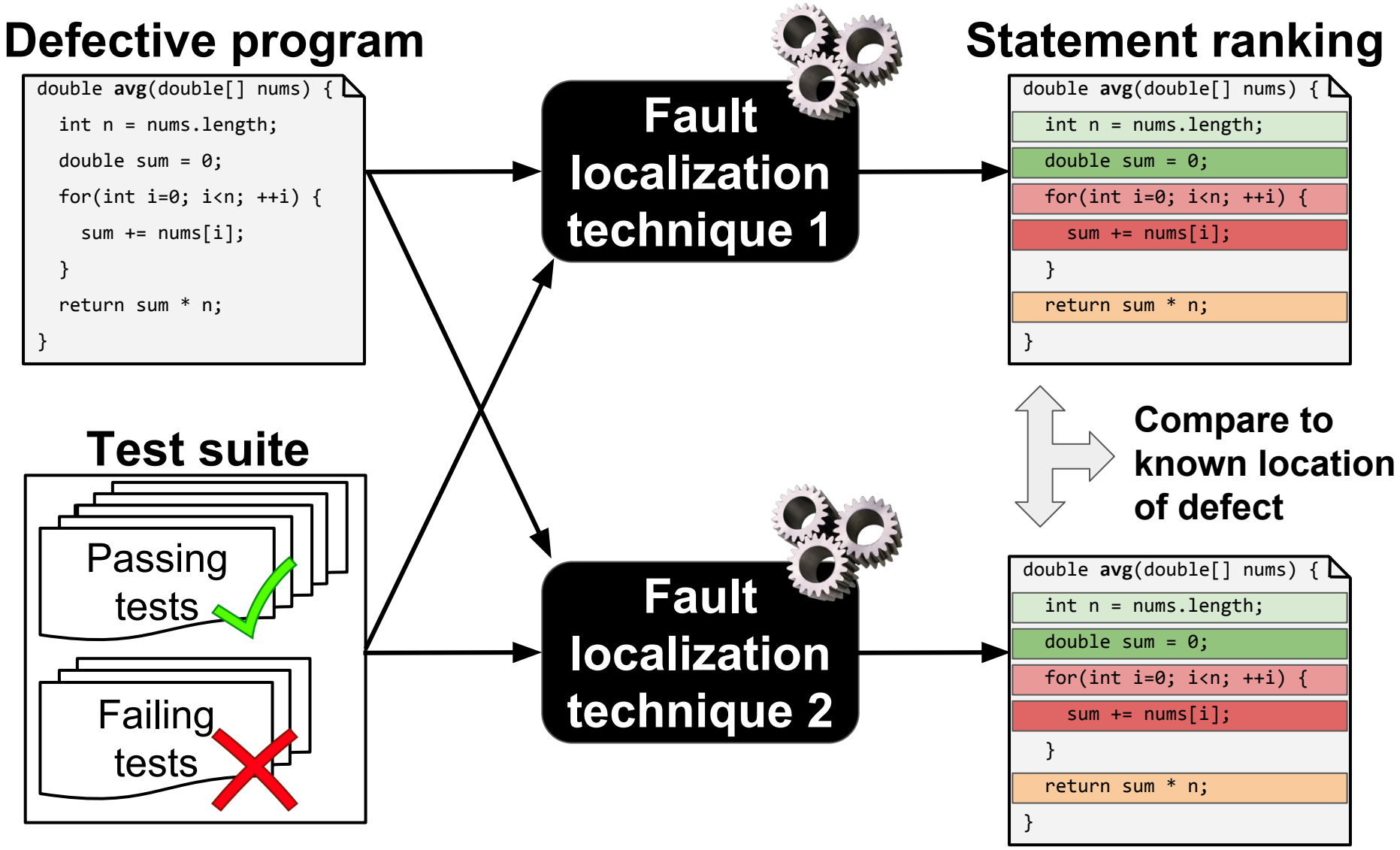
Results agree with most prior studies **on artificial faults** but **only 3 effect sizes** are **not negligible.**

Comparison of fault localization techniques

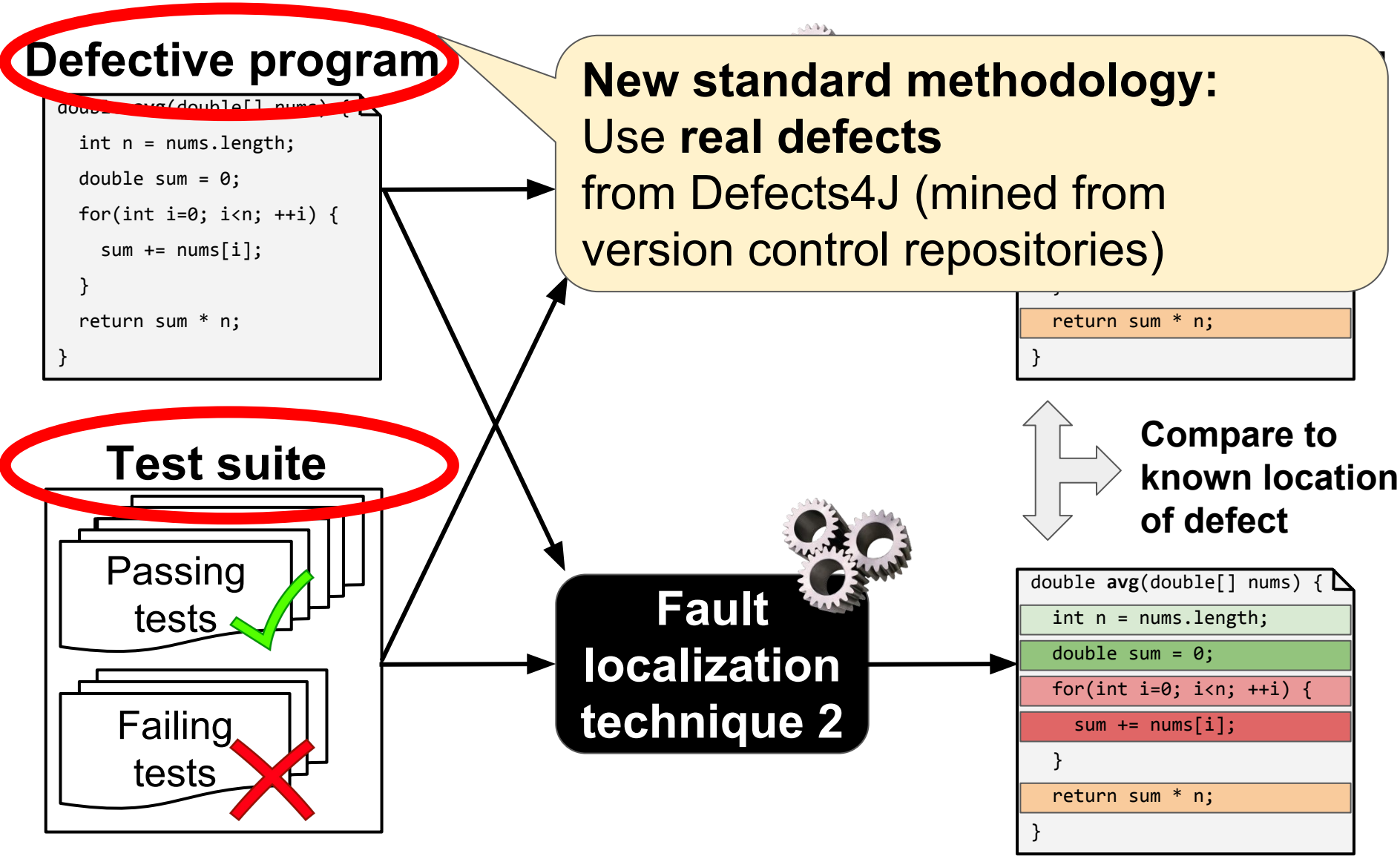
	Prior studies (winner > loser)	Ours (artificial faults) Replicated	Ours (artificial faults) Effect	Ours (real faults) Replicated	Ours (real faults) Effect
SBFL vs. SBFL	Ochiai > Tarantula	**yes**	small	*insignificant*	*negligible*
	Barinel > Ochiai	**no**	small	*insignificant*	*negligible*
	Barinel > Tarantula	**yes**	*negligible*	*insignificant*	*negligible*
	Op2 > Ochiai	**yes**	*negligible*	no	*negligible*
	Op2 > Tarantula	**yes**	small	*insignificant*	*negligible*
	DStar > Ochiai	**yes**	*negligible*	*insignificant*	*negligible*
	DStar > Tarantula	**yes**	small	*insignificant*	*negligible*
MBFL vs. SBFL	Metallaxis > Ochiai	yes	*negligible*	**no**	small
	MUSE > Op2	**no**	*negligible*	**no**	**large**
	MUSE > Tarantula	**no**	*negligible*	**no**	**large**

Results disagree with all prior studies **on real faults**.
Design decisions don't matter: techniques **indistinguishable**.

Evaluating fault localization



Evaluating fault localization

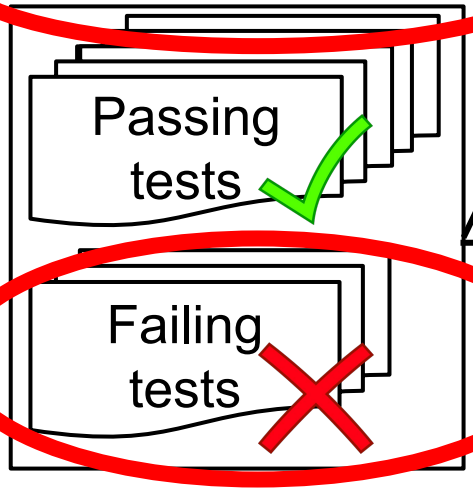


Evaluating fault localization

Defective program

```
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Test suite



New standard methodology:
Use **real defects**
from Defects4J (mined from version control repositories)

```
  return sum * n;
```

Defects4J: **real triggering tests**

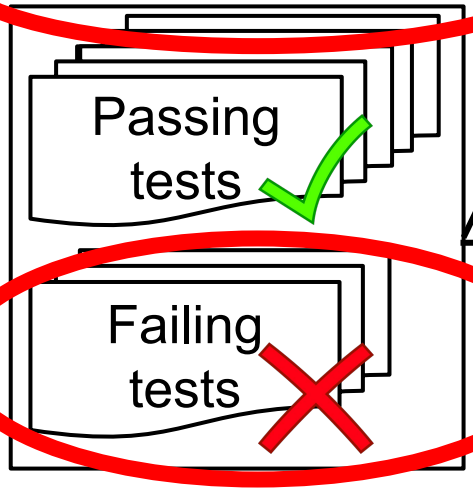
- Written by developers
- Committed with the fix

Evaluating fault localization

Defective program

```
double avg(double[] nums) {
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Test suite



New standard methodology:
Use **real defects**
from Defects4J (mined from version control repositories)

Defects4J: **real triggering tests**

- Written by developers
- Committed with the fix

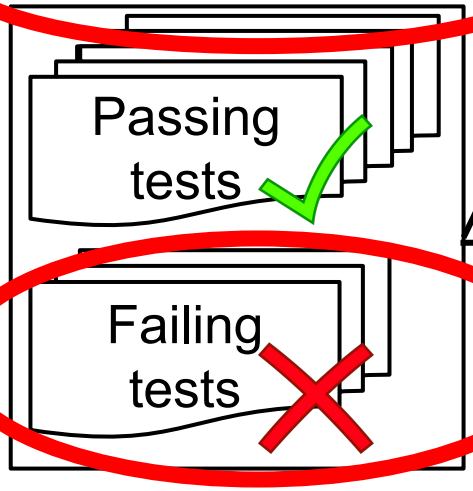
Written before or after the fix?

Evaluating fault localization

Defective program

```
  int n = nums.length;
  double sum = 0;
  for(int i=0; i<n; ++i) {
    sum += nums[i];
  }
  return sum * n;
}
```

Test suite



New standard methodology:
Use **real defects**
from Defects4J (mined from version control repositories)

Defects4J: **real triggering tests**

- Written by developers
- Committed with the fix

Written before or after the fix?

In practice, **fault localization** is run **before the fix**, using triggering **tests from bug reports**.

User-provided test

https://issues.apache.org/jira/browse/LANG-857

```
public void userTest() {
    assertEquals("\uD83D\uDE30", StringEscapeUtils.escapeCsv("\uD83D\uDE30"));
}
```

Developer-provided test

https://issues.apache.org/jira/browse/LANG-857

```
public void testLang857() {
    assertEquals("\uD83D\uDE30", StringEscapeUtils.escapeCsv("\uD83D\uDE30"));
    // Examples from https://en.wikipedia.org/wiki/UTF-16
    assertEquals("\uD800\uDC00", StringEscapeUtils.escapeCsv("\uD800\uDC00"));
    assertEquals("\uD834\uDD1E", StringEscapeUtils.escapeCsv("\uD834\uDD1E"));
    assertEquals("\uDBFF\uDFFD", StringEscapeUtils.escapeCsv("\uDBFF\uDFFD"));
}
```

Developers accept 20% of user-provided tests as is.

Developer-provided tests have:

- **More tests**, more LOC
- More, **stronger assertions** (higher mutation score)
- Less code coverage (**more focused**)

Experimental comparison

Developer-provided tests: from Defects4J

User-provided tests: manually extracted from bug reports

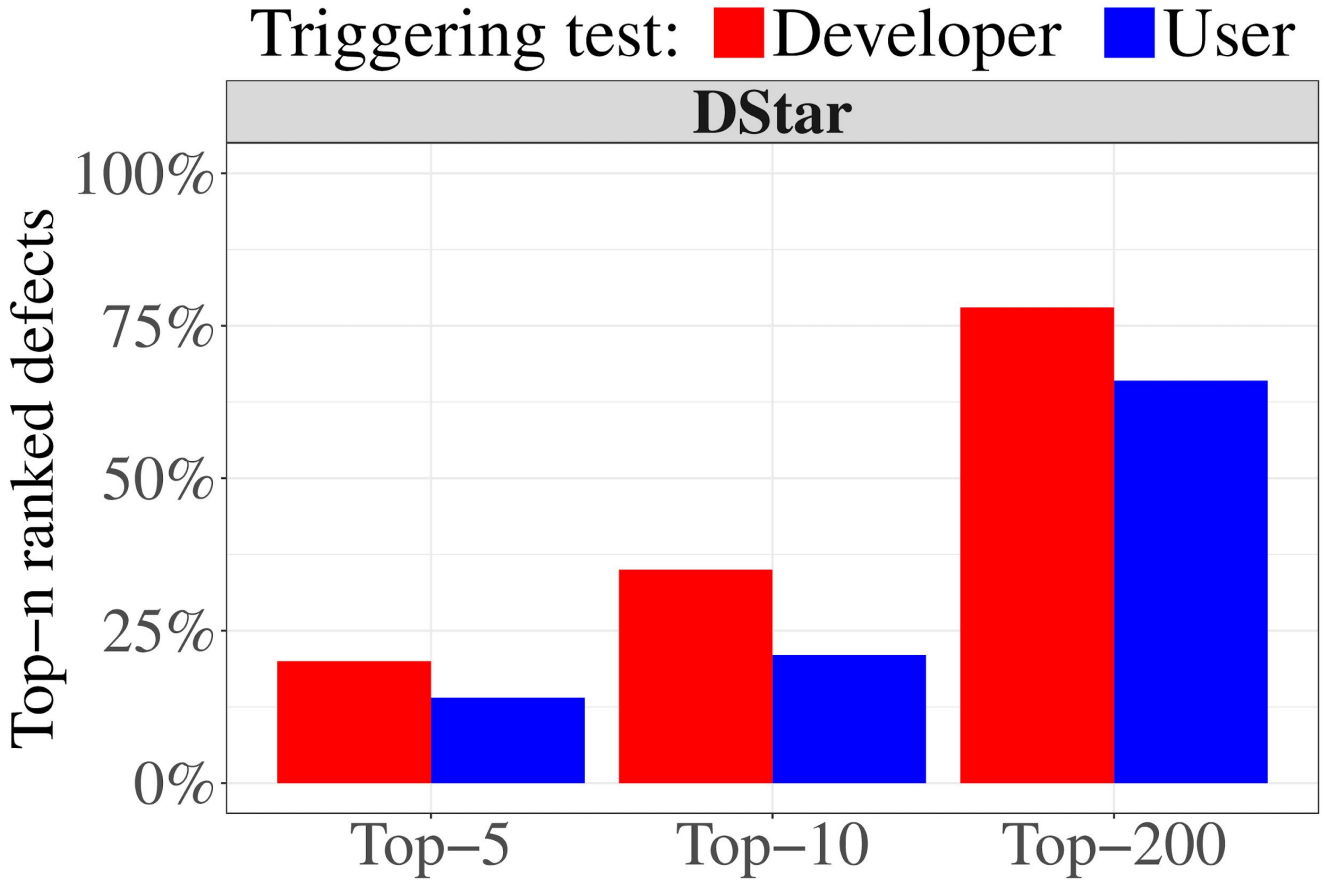
Research question: Is experimental setup (dev-provided tests) characteristic of real-world use (user-provided tests)?

- Fault localization
- Program repair

Fault localization applied to user- vs. dev-tests

Top-N metric:

Does the defective statement appear within the first N reports?



Automated program repair (395 bugs, 2 repair tools)

	Dev-provided tests	User-provided tests
jGenProg/astor		
Correct patches	1	0
ACS		
Correct patches	11	5

Partly due to worse fault localization

Automated program repair (395 bugs, 2 repair tools)

	Dev-provided tests	User-provided tests
jGenProg/astor		
Correct patches	1	0
Generated patches	6	5
ACS		
Correct patches	11	5
Generated patches	12	6

Test separation

Existing developer-written test for Commons Lang #746:

```
@Test
public void testCreateNumber() {
 // a lot of things can go wrong
 …
 assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());

 assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());

 assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
 …
}
```

More than 20 passing assertions in `testCreateNumber`!

Test separation

Augmented developer-written test for Commons Lang #746:

```
@Test
public void testCreateNumber() {
 // a lot of things can go wrong
 …
 assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());
 assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());
 assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());
 assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());
 assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
 …
}
```

Test separation

Augmented developer-written test for Commons Lang #746:

```
@Test
public void testCreateNumber() {
 // a lot of things can go wrong
 ...
 assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());
 assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());
 assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());
 assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());
 assertEquals("11 failed", Double.valueOf("1.1E20"), createNumber("1.1E20"));
 ...
}
```

Many masked passing assertions

Many non-executed passing or failing assertions

Test separation

Alternate formulation of the developer-written test:

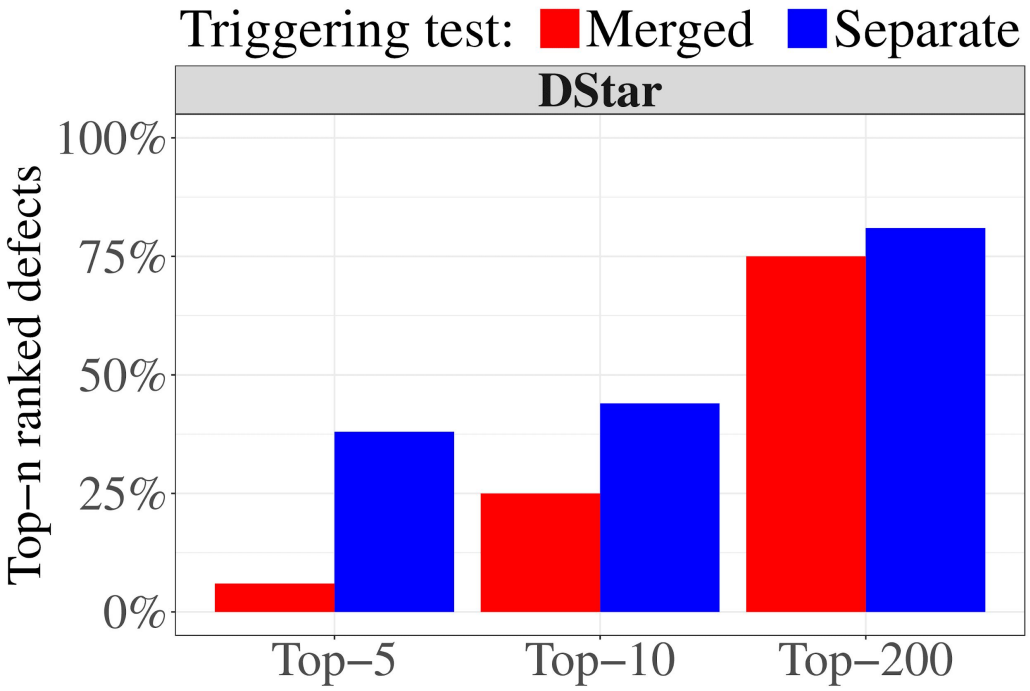
```
…
public void testCreateNumber9() {
 assertTrue("9 failed", 0xFADE == createNumber("0xFADE").intValue());
}

public void testCreateNumber9b() {
 assertTrue("9b failed", 0xFADE == createNumber("0Xfade").intValue());
}

public void testCreateNumber10() {
 assertTrue("10 failed", -0xFADE == createNumber("-0xFADE").intValue());
}

public void testCreateNumber10b() {
 assertTrue("10b failed", -0xFADE == createNumber("-0Xfade").intValue());
}
…
```

Separated tests are better for tools

What if developers never augmented tests, only added new tests?

Developer commits:

- Added only new tests 78% of the time
- Augmented an existing test 22% of the time



Tools should separate tests prior to debugging (see also [Xuan 2014]).

User-provided vs. developer-provided tests

In real-world use, only user-provided tests are available

User-provided tests:

- Smaller; weaker assertions; less focused
- Fault localization: 5-14% worse
- Automated program repair: 54-100% worse

Experiments should use real artifacts in end-user context.