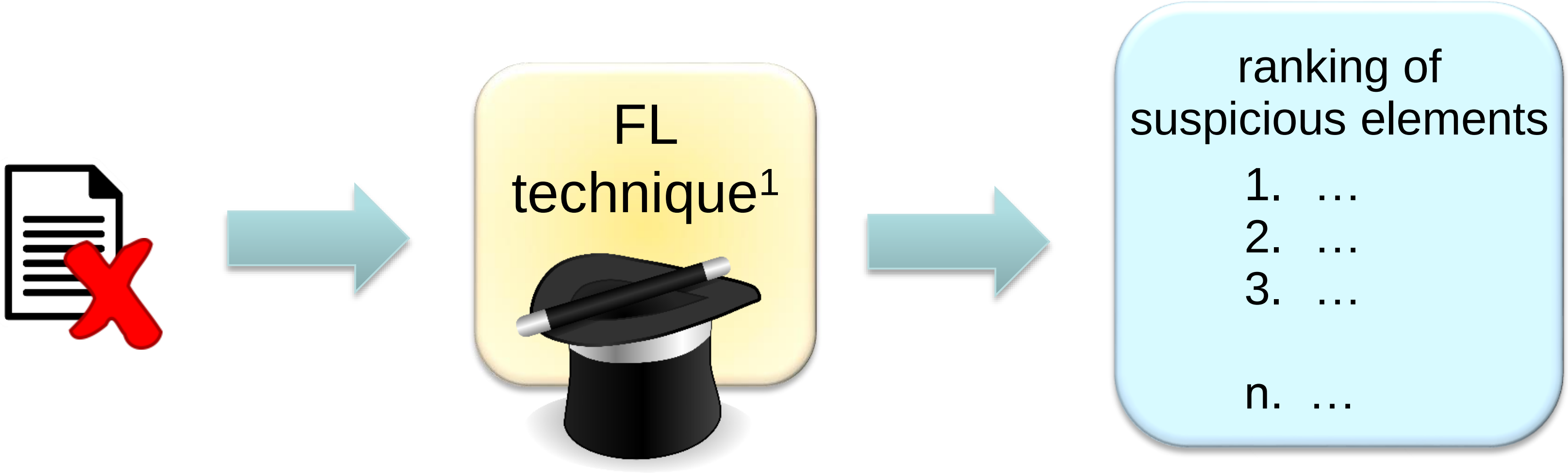
Evaluating Fault Localization Techniques with Bug Signatures and Joined Predicates

Roman Milewski, Simon Heiden, Lars Grunske

Automated Fault Localization

- given:
  - program with (at least) one *known* bug
  - usually indicated by at least one failing test case
- goals:
  - locate the cause of the bug in the source code
  - require as little human effort (time) as possible



FL technique[1]

ranking of suspicious elements

1. …
2. …
3. …

n. …

---

[1] [Won+16]

Bug Signatures

- in this context: sets of *predicates*[1,2], indicative of the occurring bug
  - common: use of data mining techniques to find the most discriminative predicate sets

- predicates:
  - boolean properties that were true at some point during a program run
  - evaluated at multiple instrumentation sites during execution
  - track simple relationships between program variables (e.g., $x < y$, $x == y$, $x >= y$, …), or other properties that can be evaluated to true or false. (e.g., $x > 0$, $x ==$ null, has a branch been taken?, …)

ranking of
suspicious elements

1. …
2. …
3. …

n. …

=

ranking of
bug signatures

1. { 12:[z>0], 14:[i==j] }
2. { 12:[z>0] }
3. { 23:[a<b], 24:[b<c] }

…

[1] [Lib+05], [2] [SK13]

Example

```
1 public boolean isAnyTrue ( boolean a, boolean b ) {
2     int x = 1;
3     if (a)
4             ++x;
5     if (b)
6             ++x;
7     return x == 2; // bug; should be x >= 2
8 }
```

- Tests:
  - isAnyTrue(true, false)
  - result: true (expected: true)

  - isAnyTrue(false, true)
  - result: true (expected: true)

  - isAnyTrue(true, true)
  - result: false (expected: true)

- most discriminative bug signature:

  { 3:[a==true], 5:[b==true] }

Example (cont.)

```
public boolean isAnyTrue ( boolean a, boolean b ) {
    int x = 1;
    if (a)
          ++x;
    if (b)
          ++x;
    return x == 2; // bug; should be x >= 2
}
```

- Tests:
  - isAnyTrue(true, false) ||
    isAnyTrue(false, true) ||
    isAnyTrue(false, false)
  - result: true (expected: true)
  - isAnyTrue(false, true) ||
    isAnyTrue(true, false) ||
    isAnyTrue(false, false)
  - result: true (expected: true)
  - isAnyTrue(true, true) ||
    isAnyTrue(false, false)
  - result: false (expected: true)

- problem:
  - collected predicate data does not allow to discriminate between test cases.

Joined Predicates

- main idea: utilize information about the *order of predicates*
- *joined predicate:* sequence of satisfied predicates that has been observed during a run of the program
    - in our implementation: *two* predicates that directly follow each other

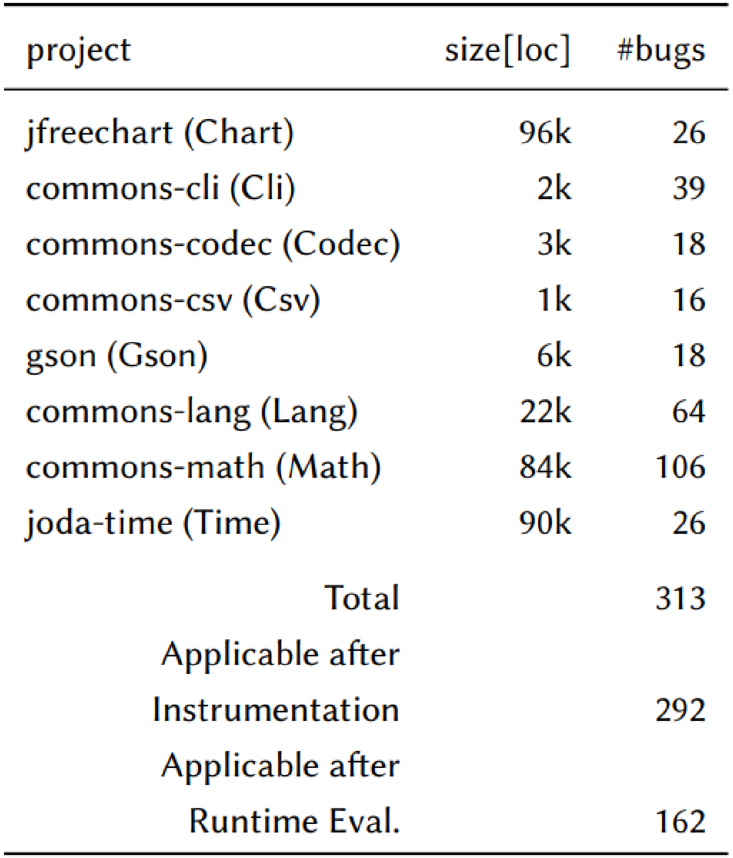
```
1 public boolean isAnyTrue ( boolean a, boolean b ) {
2     int x = 1;
3     if (a)
4             ++x;
5     if (b)
6             ++x;
7     return x == 2; // bug; should be x >= 2
8 }
```

- a discriminative (joined) predicate:

3:[a==true] → 5:[b==true]

Evaluation

- three FL techniques:
  - **joined predicate** bug signature mining (our approach)
  - **singular predicate** bug signature mining[1]
  - **spectrum-based fault localization** (SBFL)[2] using the DStar (D*) metric[3]
- evaluated on a subset of bugs from the **Defects4J benchmark**[4]
  - issues: source code compilation, instrumenter crashes, JVM crashes during test execution, runtime evaluation timeouts, …



project	size[loc]	#bugs
jfreechart (Chart)	96k	26
commons-cli (Cli)	2k	39
commons-codec (Codec)	3k	18
commons-csv (Csv)	1k	16
gson (Gson)	6k	18
commons-lang (Lang)	22k	64
commons-math (Math)	84k	106
joda-time (Time)	90k	26
Total		313
Applicable after Instrumentation		292
Applicable after Runtime Eval.		162

[1] [SK13], [2] [Hei+19], [3] [Won+14], [4] [JJE14]

Quick side note: SBFL

```
1 public boolean isAnyTrue ( boolean a, boolean b ) {
2     int x = 1;
3     if (a)
4             ++x;
5     if (b)
6             ++x;
7     return x == 2; // bug; should be x >= 2
8 }
```

- Tests:
  - isAnyTrue(true, false)
  - result: true (exp.: true)

  - isAnyTrue(false, true)
  - result: true (exp.: true)

  - isAnyTrue(true, true)
  - result: false (exp.: true)

- output is *ranking of executed lines*
- SBFL assigns a score to each line
  - based on the number of successful and failing test cases that execute it (or do not)
  - Dstar (D*): $susp(s) = \frac{n_{ef}^{*}(s)}{n_{ep}(s)+n_{nf}(s)}$
- lines 4 and 6 are seen as most suspicious (executed less by passing test cases)

Evaluation metrics

- results are difficult to compare across FL techniques
    - not all executed lines (i.e., included in SBFL results) are instrumentation sites for the predicate based approaches
    - our evaluation metric „simulates“ the user looking for the bug in the neighborhood of the instrumentation sites, adding a penalty for the performed additional effort (details in the paper)

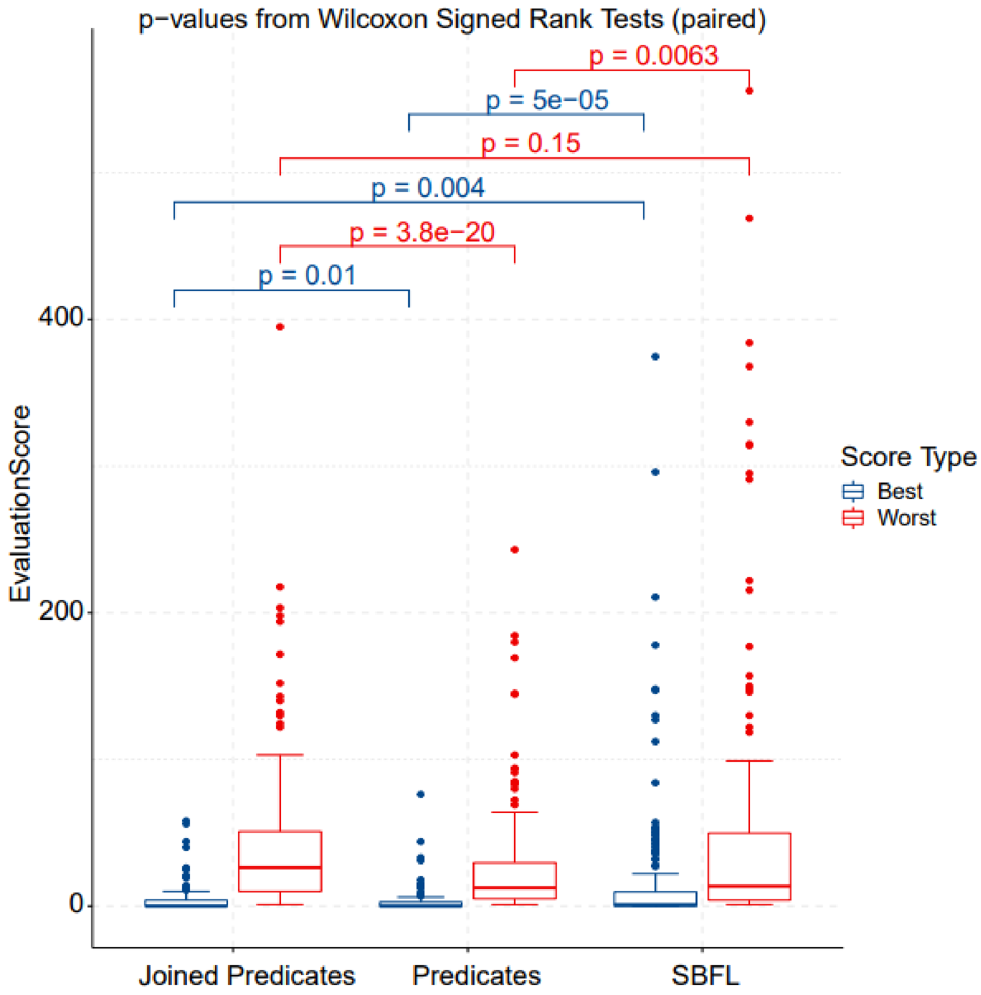
- simplified, results are evaluated with a simple *wasted effort metric*

- elements can have the same suspiciousness score, so we consider the following 3 cases:
    - min. wasted effort: $Score_{best}(s) = |\{s' | susp(s') > susp(s)\}|$
    - max. wasted effort: $Score_{worst}(s) = |\{s' | susp(s') \geq susp(s)\}| - 1$
    - avg. wasted effort: $Score_{avg}(s) = \frac{Score_{best}(s) + Score_{worst}(s)}{2}$

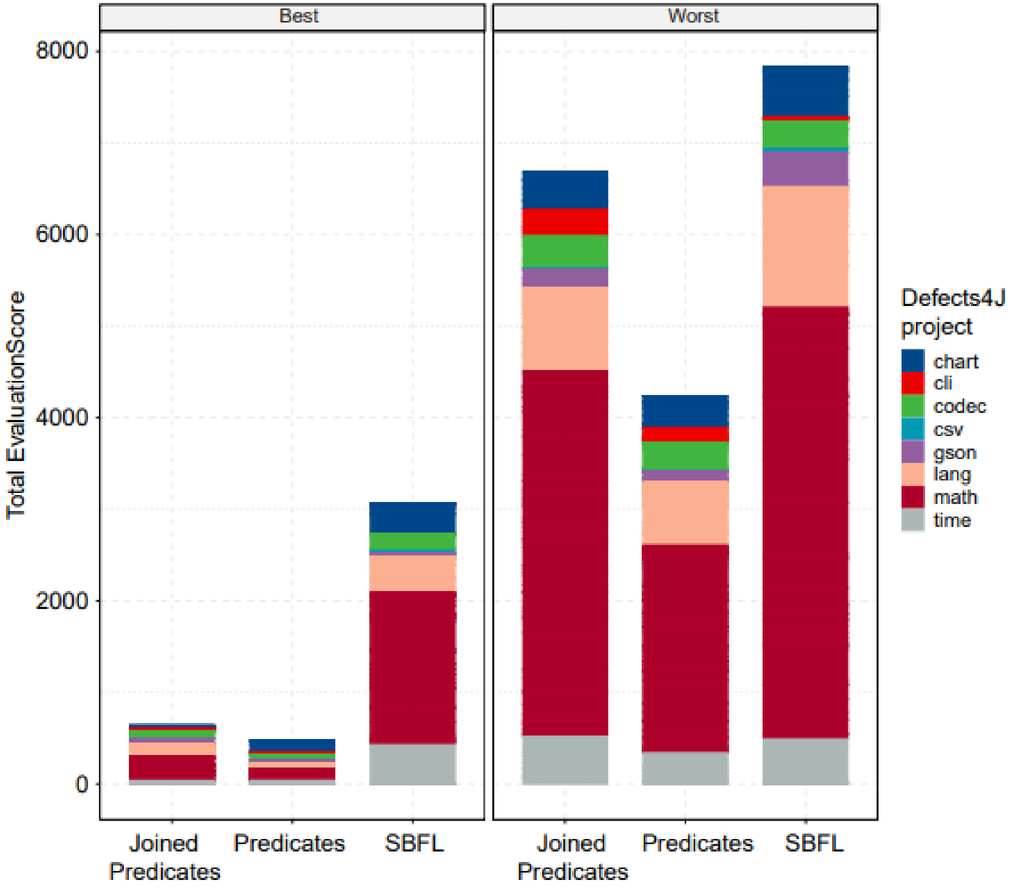
Results

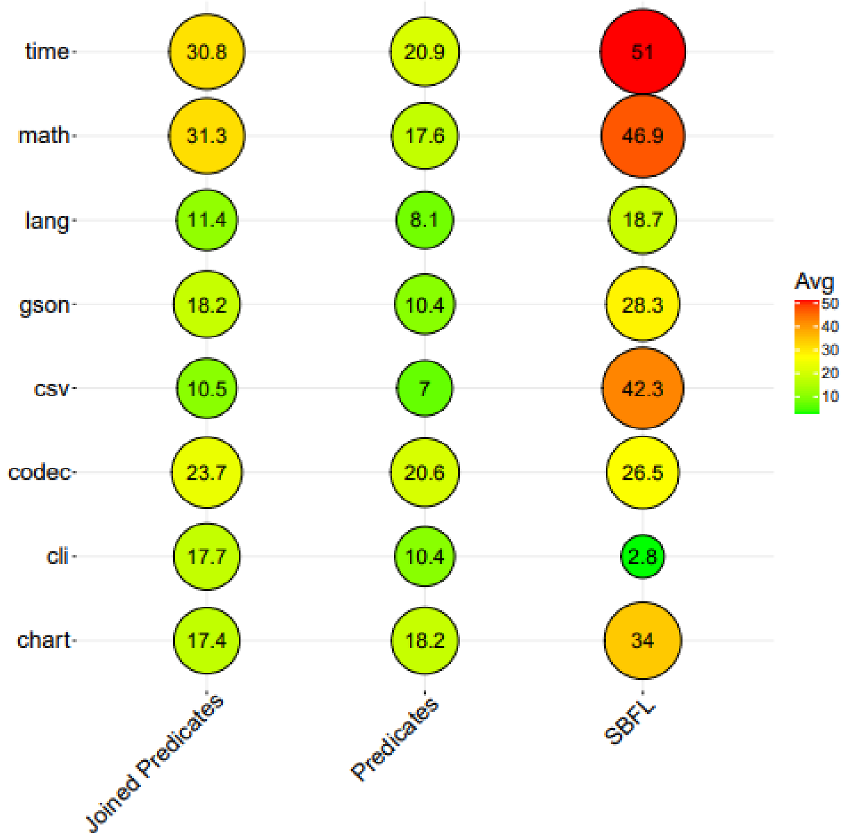
- best case (lucky placement):
  - (singular) predicate approach *is better than*
  - joined predicate approach *is better than*
  - SBFL approach

- worst case (unlucky placement):
  - (singular) predicate approach *is better than*
  - joined predicate approach *and*
  - SBFL approach



Results (cont.)





- both predicate based approaches achieve better average scores across nearly all projects (except cli)

Conclusions

1. the singular predicate based approach is clearly superior to SBFL
    - quantitatively, as shown in our experiments
    - qualitatively, since the user may access additional information beyond the mere execution of a program element

2. using our prototype joined predicate based approach in its current state is not advisable
    - mediocre quantitative results
    - potential qualitative benefits will likely not outweigh the overall decreased performance

Future Work

- potential changes to make the approach more viable:
  - smarter (pre-)selection of predicates to be joined (heuristics, etc.)
    - reduction of noise
  - optimizing the technical implementation (improved data structures, statistical debugging techniques, …)

thank you!

References

- [Won+16] W. E. Wong, R. Gao, Y. Li, R. Abreu, F. Wotawa, A survey on software fault localization, IEEE Transactions on Software Engineering 42 (2016) 707–740. doi:10.1109/tse.2016. 2521368.
- [Lib+05] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, M. I. Jordan, Scalable statistical bug isolation, ACM SIGPLAN Notices 40 (2005) 15–26. doi:10.1145/1064978.1065014.
- [SK13] C. Sun, S.-C. Khoo, Mining succinct predicated bug signatures, in: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2013, ACM Press, 2013, pp. 576–586. doi:10.1145/2491411.2491449
- [Won+14] W. E. Wong, V. Debroy, R. Gao, Y. Li, The DStar method for effective software fault localization, IEEE Transactions on Reliability 63 (2014) 290–308. doi:10.1109/tr.2013. 2285319.
- [Hei+19] S. Heiden, L. Grunske, T. Kehrer, F. Keller, A. Hoorn, A. Filieri, D. Lo, An evaluation of pure spectrum-based fault localization techniques for large-scale software systems, Software: Practice and Experience 49 (2019) 1197–1224. doi:10.1002/spe.2703
- [JJE14] R. Just, D. Jalali, M. D. Ernst, Defects4j: a database of existing faults to enable controlled testing studies for java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis - ISSTA 2014, ACM Press, 2014, pp. 437–440. doi:10.1145/ 2610384.2628055