# Reduce Before You Localize: Delta-Debugging and Spectrum-Based Fault Localization

Arpit Christi,* Matthew Lyle Olson,* Mohammad Amin Alipour,* Alex Groce*

## Abstract

*Spectrum based fault localization is one of the most popular and studied method of automated debugging. Since the original work on this topic, many formulas have been proposed to improve the accuracy of fault localization scores. Most of these improvements are either marginal or context-dependent.This paper proposes that, independent of the scoring method used, the effectiveness of spectrum-based localization can be dramatically improved by, when possible, delta-debugging failing test cases and basing localization only on the reduced test cases. We show that for programs and faults taken from the standard localization literature, a large case study of Mozillas JavaScript engine using 10 real faults, and for mutants of various open source projects, localizing only after reduction often produces much better rankings for faults than localization without reduction, independent of the localization formula used, and the improvement is often even greater than that provided by changing from the worst to the best localization formula for a subject.*

## I. Introduction

Debugging is one of the most time-consuming and difficult aspects of software development [**?**], [**?**]. Recent years have seen a wide variety of research efforts devoted to easing the burden of debugging by *automatically localizing faults*. The most popular approaches, following the seminal work of Jones, Harrold, and Stasko [**?**], [**?**] use statistics of spectra [**?**] of failing and successful executions to score program entities according to how likely they are to be faulty. These spectrum-based approaches are popular in part because they have outperformed competing approaches, and in part because they are highly efficient and easy to use — they typically only require the collection of coverage data and marking of tests as passing and failing, and thus are both computationally cheap and easy to fully automate. Many formulas have been proposed as potentially improving the accuracy of scores [**?**], [**?**], [**?**], [**?**], [**?**] over Tarantula.

Despite this large body of work and continuing interest, there is recent concern about the long-term value of localization research. Parnin and Orso asked the core question: "Are automated debugging techniques actually helping programmers?" [**?**], and did not receive comforting answers. Parnin and Orso studied how actual programmers made use of localization techniques [**?**]. and concluded 1) a proposal to use absolute rank to measure effectiveness, because developers lose interest in localizations after a very few incorrect suggestions, and 2) focus on using richer information (e.g. the actual test cases) rather than just a raw localization in debugging aids. Combined with Yoo et al.'s establishment [**?**] that there is no truly optimal formula for localization, this suggests that the most valuable contributions to localization would be *formula-independent* methods that *potentially result in extremely large improvements in fault rank* rather than small, incremental average improvements in rank. This paper, therefore, argues that in many cases there *is* a simple, easily applied, improvement to localization that works with any formula (or other modification to the method we are aware of), has benefits to developers even if they ignore the localization, and often produces very large improvements in what we consider the most important measure of localization effectiveness, the absolute worst possible ranking of the faulty code.

### A. Reduce Before You Localize

Failing test cases, apart from executing faulty code normally executes large amount of non faulty code also, which makes debugging and fault localization non trivial. It is often impossible to produce a failing test case that executes only faulty code. Due to the way spectrum-based localizations work, reducing the amount of non-faulty code executed in failing test cases will almost always improve localization. Consider the Tarantula [**?**], [**?**] formula. Tarantula, like most spectrum-based approaches, determines how suspicious (likely to be faulty) a coverage entity $e$ (typically a statement) is based on a few values computed over a test suite:

- passed($e$): # of tests covering $e$ that pass
- failed($e$): # of tests covering $e$ that pass

- totalpassed: the # of passing tests
- totalfailed: the # of failing tests

$$\text{suspiciousness}(e) = \frac{\frac{\text{failed}(e)}{\text{totalfailed}}}{\frac{\text{failed}(e)}{\text{totalfailed}} + \frac{\text{passed}(e)}{\text{totalpassed}}}$$

It is easy to see that if we lower failed$(e)$ for all non-faulty statements, while keeping everything else unchanged, the rank (in suspiciousness) of faulty statements will improve. Reducing coverage of non-faulty statements in failing tests, then, is a potentially very effective and formula-independent approach to improving localizations.

Unfortunately, there is no method in the literature for reducing the amount of non-faulty code executed in test cases, to our knowledge.[1] However, there *is* a widely used method for reducing the size of failing test cases, delta-debugging.

Delta-debugging [**?**] (DD for short) is an algorithm for reducing the size of failing test cases. Delta-debugging algorithms have retained a common core since early proposals [**?**]: use a variation on binary search to remove individual components of a failing test case $t$ to produce a *new* test case $t_{1min}$ satisfying two properties: (1) $t_{1min}$ fails and (2) removing any component from $t_{1min}$ results in a test case that does not fail. Such a test case is called *1-minimal*. Delta-debugging reduces the size of a test case in terms of its components. Its purpose is to produce small test cases that are easier for humans to read and understand, and thus debug. In our long experience with delta-debugging [**?**], [**?**] and in recent work on variations and applications of delta-debugging [**?**], [**?**], [**?**], we noticed that in addition to reducing the "static" human-readable text of a test case, delta-debugging also almost always *reduces the code covered by a failing test case*, often by hundreds or thousands of lines [**?**]. The core proposal of this paper, therefore, is that *failing test cases should be, when possible, reduced with delta-debugging before they are used in spectrum-based fault localization*: **reduce before you localize**. The original, unreduced, test cases should not be used, as they likely contain much irrelevant, non-faulty code that may mislead localization.

Even if reduction does not improve the localization, we show that under some reasonable assumptions it will not produce a *worse* localization, and at least the developer now has a set of smaller, easier-to-understand test cases to read. In fact, we believe *the only reason not to reduce test cases before localization, as a best practice, is when it is too onerous (or not possible) to set up delta-debugging for test cases.*

In this paper, we show that using delta-debugging to reduce failing test cases, when applicable, does usually produce improvements in fault ranking, using a variety of standard localization formula from the literature, and these improvements are often dramatic. In order to place our results on a firm empirical footing [**?**] we provide results over both SIR/Siemens [**?**] suite subjects studied in previous literature, a set of real faults from an industrial-strength random testing framework for the SpiderMonkey JavaScript engine [**?**], [**?**], and a variety of open source Java programs. Not only does reducing failing tests produce improvements; the improvements produced are often even better than those provided by optimally switching formula.

[1] Cause reduction [**?**], as we note later, could probably be used for this purpose, but might be expensive and has not been applied to this goal.

## II. Related Work

As discussed in the introduction, there is a very large body of work on spectrum-based fault localization (e.g. [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]), all of which informs our work. The most important motivational results for this paper are the investigation of Parnin and Orso [**?**] into the actual use of localizations for programmers, which inspired our evaluation methods, and the claim of Yoo et al. [**?**] that no single formula is best, which directed us to seek formula-independent improvements to localization. Our use of many programs and methods was inspired by the threats identified by Steimann et al. to empirical assessments of fault localizations [**?**].

The most similar actual proposed improvement to localization to ours is the entropy-based approach of Campos et al. [**?**] that uses EvoSuite [**?**] to improve test suites. The underlying approaches are quite different, but both aim to improve the spectra used in localization rather than change their interpretation. The primary advantage of their approach over ours is that it can be of use when test cases cannot be reduced; on the other hand, EvoSuite is probably considerably harder to apply for most developers than off-the-shelf delta-debugging. Another similar approach (sharing the same novel aspect of changing the test cases examined rather than the scoring function) is that of Xuan and Monperrus, who propose a *purification* for test cases [**?**] that executes omitted assertions and uses dynamic slicing [**?**] to remove some code from failing test cases (parameterized by each assertion). Delta-debugging can remove code from unit tests that would be in any dynamic slice, since it does not have to respect any property but that the test case still fails. A core practical difference is that their approach *only* applies to unit tests of method calls (since the slicing is at the test level, not of the program tested), and that we believe delta-debugging tools are more widely used and easily applicable than slicing tools (e.g they are language-independent).

This paper also follows previous work on delta-debugging [**?**], [**?**], [**?**] and its value in debugging tasks. The most relevant recent work is the set of papers proposing that in addition to producing small test cases for humans to read, delta-debugging is a valuable tool in fully

automated software engineering algorithms even if humans do not read the reduced tests: e.g., it is helpful for producing very fast regression suites [**?**], for improving coverage with symbolic execution [**?**], and for clustering/ranking test cases by the underlying fault involved [**?**].

## III. Assumptions and Guarantees

In addition to being orthogonal to the spectrum-based localization formula used, test case reduction has a second major advantage independent of empirical results. Namely, under a set of assumptions that hold in many cases, *reduction can only improve, or leave unchanged, the effectiveness of localization*. All formulas for localization have some instances in which they diminish the effectiveness of localization compared to an alternative formula [**?**]. Reducing failing tests before applying a formula, however, at worst leaves the effectiveness of localization unchanged, for most of the formulas in widespread use that we are aware of, under three assumptions:

1) all failing test cases used in the localization involve the same fault,
2) each failing test case reduces to a test case that fails due to the same fault as the original test case, and
3) reducing the input size (in components) also covers less code when the test executes.

The first assumption is probably the least likely to hold in some settings; however, it is also the assumption that is least relied upon. The second assumption is a usual assumption of delta-debugging. Most delta-debugging setups are engineered with this goal in mind, often using some aspect of failure output or test case structure to keep "the same bug." Observed "slippage" rates for faults seem to be fairly low [**?**], even with little mitigation. Note that in the setting where a program has a single fault, assumptions 1 and 2 always hold. For third assumption, it is moderately uncommon though slightly possible for reduced test cases to increase coverage but this can be mitigated easily by cause reduction, a generalized version of delta debugging. [cite]

Given these assumptions, we now show that reduction is, at worst, harmless for most formulas. Recall that spectrum-based localizations rely on only a few values relevant to each entity $e$ to be ranked in a localization: passed($e$), failed($e$), totalpassed, and totalfailed. Given assumptions 1-3 above, *for faulty statements all of these formula elements will be unchanged after delta-debugging*. For non-faulty statements, the only possible change is that failed($e$) may be lower than before failing test cases were reduced. Holding the other values constant, it is trivial to show that most formulas under consideration are (as we would expect), monotonically increasing in failed($e$). Therefore, after reduction, the suspiciousness scores for faulty statements are unchanged and the suspiciousness scores for non-faulty statements are either unchanged or lower. The rank of all faulty statements is therefore either unchanged or improved. There are many spectrum-based fault localization formulas in use. In our evaluation, we have used 3 well known examples, in addition to the basic Tarantula [**?**] formula, as representative:

**Ochiai [?]:**

$$\text{suspiciousness}(e) = \frac{\text{failed}(e)}{\sqrt{(\text{totalfailed})(\text{failed}(e) + \text{passed}(e))}}$$

**Jaccard [?]:**

$$\text{suspiciousness}(e) = \frac{\text{failed}(e)}{\text{failed(e)} + \text{totalfailed}}$$

**SBI: [?], [?]**

$$\text{suspiciousness}(e) = \frac{\text{failed}(e)}{\text{failed(e)} + \text{passed}(e)}$$

All these formulas are monotonically increasing in failed($e$).[2] Reduction can only theoritically improve fault localization or in worst case leave it unchanged if formula under consideration is monotonically increasing failed($e$). We chose these 4 formulas as they were used before to study effect of *test suite reduction on fault localizaiton*,to study *effect of test case purificaiton on fault localization* [**?**] [**?**].

## IV. Experimental Results

### A. Evaluation Measure

Because we aim to take into account the findings of Parnin and Orso [**?**], our evaluation of fault localizations is based on a *pessimistic absolute rank* of the highest ranked faulty statement. That is, for each set of suspiciousness metrics computed, our measure of effectiveness is the *worst possible position* at which the first faulty statement can be reached, when examining the code in suspiciousness-ranked order.[3] For example, if ten statements all receive a suspiciousness score of 1.0 (the highest possible suspiciousness), and one of these is the fault, we assign this localization a rank of 10; an unlucky programmer might examine this statement last of the ten highest-ranked statements. Pessimistic rank nicely distinguishes this result from another localization that also places the bug at score 1.0, but gives twenty statements a 1.0 score. In our view, following Parnin and Orso [**?**], the most important goal of a localization is to direct the developer to a faulty statement as rapidly as possible, ignoring the size of the entire program or even of the faulty execution.

[2] Proofs available on request, and to be posted on our web page as an appendix.

[3] As usual since at least the work of Reneiris and Reiss[**?**], we consider reaching *any* faulty statement to be sufficient.

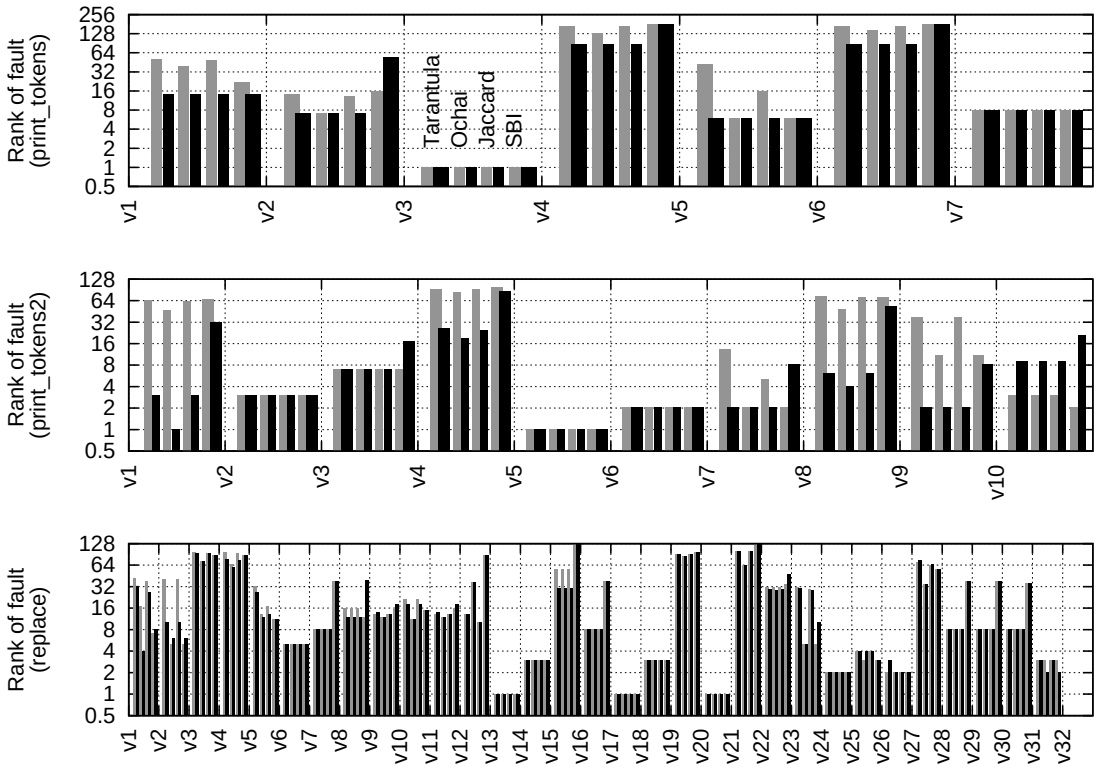


**Fig. 1. First Set of SIR Results**

| Subject | Avg. | Avg. (DD) | #Better | #Same | #Worse |
|---|---|---|---|---|---|
| print_tokens | 59.7 | 29.7 | 18 | 10 | 0 |
| print_tokens2 | 27.0 | 5.8 | 19 | 17 | 4 |
| replace | 24.5 | 21.8 | 37 | 76 | 11 |
| schedule | 7.7 | 14.3 | 18 | 14 | 4 |
| schedule2 | 92.4 | 76.6 | 28 | 12 | 0 |
| tot_info | 29.7 | 17.7 | 75 | 16 | 1 |
| **Total** | | | 195 | 145 | 20 |

**TABLE I. SIR Fault Rank Change Result Frequencies**

## B. SIR Programs

Our initial experiments use the Siemens/SIR [**?**], [**?**] suite programs studied in many previous papers on fault localization, in particular the classic evaluation of the Tarantula technique [**?**]. These subjects provide a large number of faults, reasonable-sized test suites, and have historically been used to evaluate localization methods.

Of the seven Siemens programs considered in the empirical evaluation of Tarantula, only one was unsuitable for delta debugging: TCAS takes as input a fixed-size vector of integers, and therefore its inputs cannot be easily decomposed. For the remainder of the programs, the input is easily considered as either 1) a sequence of characters or 2) a sequence of lines, when character-level delta debugging is not efficient (and so unlikely to be chosen by users in practice), which was required for the `tot_info` subject. In all cases, reduction took on average less than three seconds per failing test case, an essentially negligible computational cost.

We evaluated our proposal by 1) first computing the fault ranking for each version of each subject by the five formulas then 2) performing the same computation, but using only reduced (by delta-debugging) versions of the failing tests. Reduction was performed using Zeller's delta-debugging scripts, available on the web, and comparing the output of the original (correct) version of the program and the faulty version as a pass/fail oracle.

Figures 1 and 2 show the results[4]. The lighter shaded bars show the ranking of the fault, without any reduction. The darker bars show the ranking after delta-debugging all failures. The graphs are shown in log-scale due to the range of rankings involved. In many cases, reducing test cases before localizing improved the ranking of the fault by a factor of two or more. Results for individual subject vary: for `print_tokens`, the average ranking for faults, over

[4]Version 32 of `replace` is missing because the fault relies on the library definition of `isalnum`, and is no longer present on modern Linux systems, as we determined after communicating with Gregg Rothermel and the SIR maintainers.

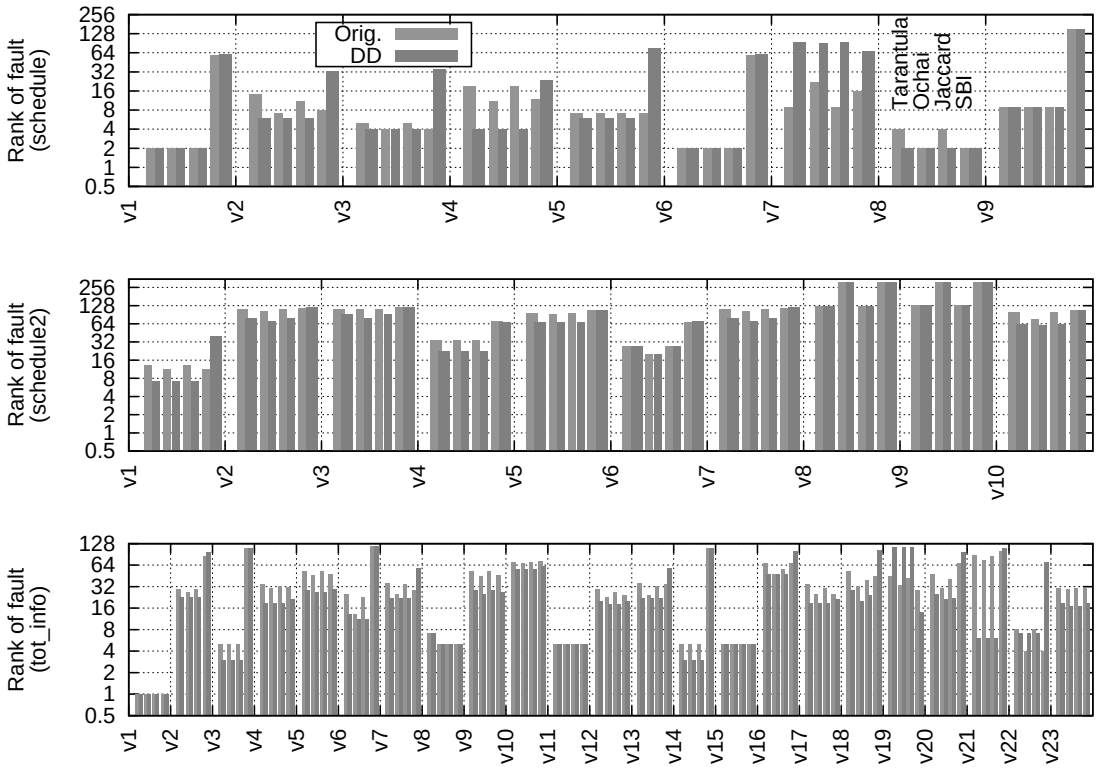


**Fig. 2. Second Set of SIR Results**

all bugs and all formulas, is 59.5 without reduction, and 36.4 with reduction. The result is improved by reduction in 19 cases, remains the same in 15 cases, and is worse in 1 case. These numbers improve to 29.8 average ranking, 18 improvements, and 10 unchanged results. Table I shows similar data for all the SIR subjects.

Our results exhibits improvement in fault rank was 1.3 times as common as no change in rank, and *nearly 10 times as common as worse rank for the fault.* In addition to the frequency of improvement of fault rank, it is also important to examine the degree of improvement (or the opposite) provided by reduction. Table II shows, the min, max, and average for changes in rank. the effect size when reduction improved rank was usually much larger than the effect size when it gave worse results. For `replace`, the subject with the most instances where reduction made fault ranking worse, we see that the effect size when reduction was harmful was much smaller than when reduction was helpful. Furthermore, when reduction helped, it often improved the ranking of the fault by more than *optimally* switching formula That is, we can ask: if we compare taking the *worst* formula and applying reduction to improve the localization, how often is this better than switching to the *best* localization formula for that subject and fault? Obviously applying reduction is more practical, since we don't know in advance which formula will perform best, until we know the correct result. By this comparison, it was better to apply reduction than switch from *worst* to *best* formula in 36 cases over all SIR subjects. It was better to switch formula in only 22 cases (in the remaining 32 cases these two methods were tied).

| Subject | Better | | | Worse | | |
|---|---|---|---|---|---|---|
| | Min | Max | Avg. | Min | Max | Avg |
| `print_tokens` | 6 | 85 | 44.6 | N/A | N/A | N/A |
| `print_tokens2` | 3 | 67 | 45.8 | 6 | 6 | 6.0 |
| `replace` | 1 | 30 | 9.6 | 1 | 3 | 1.4 |
| `schedule` | 1 | 15 | 4.8 | 85 | 75 | 81.3 |
| `schedule2` | 4 | 35 | 22.6 | N/A | N/A | N/A |
| `tot_info` | 1 | 81 | 14.7 | 3 | 3 | 3.0 |

**TABLE II. SIR Fault Rank Change Effect Sizes**

## C. SpiderMonkey JavaScript Engine

SpiderMonkey is the JavaScript Engine for Mozilla, an extremely widely used, security-critical interpreter/JIT compiler. SpiderMonkey has been the target of aggressive random testing for many years now. A single fuzzing tool, `jsfunfuzz` [**?**], is responsible for identifying more than 1,700 previously unknown bugs in SpiderMonkey [**?**]. SpiderMonkey is (and was) very actively developed, with

| Bug# | Revision Fixed | #Failures | diff size |
|---|---|---|---|
| R60 | 1.16.2.1 | 1 | 115 |
| R95 | 1.3.2.3.8 | 7 | 111 |
| R115 | 1.4.8.1 | 4 | 592 |
| R360 | 3.117.2.6 | 3 | 223 |
| R880 | 3.17.2.14 | 28 | 272 |
| R1172 | 3.208.2.63 | 150 | 214 |
| R1294 | 3.241.2.1 | 405 | 80 |
| R1543 | 3.36.16.1 | 146 | 169 |
| R1561 | 3.37.2.1.4.1.2.2 | 2 | 31 |
| R1873 | 3.50.2.29 | 1,041 | 56 |

**TABLE III. Spidermonkey Bugs**

| | Program Source | | | Test Suite |
|---|---|---|---|---|
| Subject | #Classes. | #Methods | SLOC | #Test cases |
| Apache Commons Validator | 64 | 578 | 6,033 | 434 |
| JExel 1.0.0 beta 13 | 43 | 133 | 1,522 | 344 |
| JAxen | 167 | 1,078 | 12,462 | 2,138 |
| JParser | 115 | 178 | 3,046 | 647 |
| Apache Commons CLI | 23 | 208 | 2,667 | 364 |

**TABLE IV. Open Source Subject Programs**

over 6,000 code commits in the period from 1/06 to 9/11 (nearly 4 commits/day). SpiderMonkey is thus ideal for evaluating how reduction aids localization when using a sophisticated random testing system, using the last public release of the `jsfunfuzz` tool [**?**], modified for swarm testing [**?**]. Using a set of faults in SpiderMonkey version 1.6 found with random testing in previous research [**?**], we show that reduction is essential for localization of bugs found using random testing, and that the use of reduction is even somewhat more important than the choice of localization formula.

Figure 3 shows the change in rankings of the faulty code for 10 SpiderMonkey bugs (Table III). These bugs were taken from our PLDI 2013 data set [**?**]. Out of the 28 bugs studied in that paper we chose 10 random bugs for which, by hand, we could confirm the true set of faulty lines in the code commit. Each bug is identified by the revision number of the commit in which it was fixed: e.g., R0 maps to revision 1.10.4.1, the first commit of Spidermonkey changes under consideration. Table III shows all bugs studied, the commit version fixing the bug, the number of failing test cases for that bug (# Failures), and the size (in lines) of the fixing commit's `diff`. The faults under consideration here are clearly non-trivial (in fact, most fixes involved changes to multiple source files). For our localization we used the original and reduced test cases from PLDI 13 [**?**] plus a set of 720 randomly generated passing tests generated using the same method as in the original data set.

Across these 10 bugs, the average ranking for the first faulty line encountered was 1,550.7 without reduction, improving to 994.5 with reduction. Reduction improved the localization in 33 cases, with a minimum improvement of 1 ranking and a maximum improvement of 2,137 positions. The average improvement was 674 positions. The results were unchanged in 7 cases. It is important to note that even with such a challenging setting and real bugs, fault localization with reduction performs better then fault localization without reduction irrespective of formula being used, in worst case it performed as good. It was better to use reduction than to optimally (from worst to best) switch formula for 5 of the 10 bugs; it was better to switch formula in 4 cases, and in one case both methods gave the same result. While the results show that reduction was extremely effective in improving localization, it is also true that the localization was *still not very helpful* in many of these cases, considering absolute pessimistic rank as success criteria. Of course, SpiderMonkey 1.6 has over 80KLOC, and even reduced failing tests typically executed over 8,000 lines of code, so a "poor" localization may be useful in such a large fault search space. For 6 of the 10 bugs, all scores after reduction gave the fault a ranking of 128 or better.

In order to expand on how reduction improves localization, we examine in more detail the bug we call "R1543," fixed by commit 3.36.16.1, and its Tarantula localization. In our experiments, there were 146 test cases that failed due to bug R1543. The relevant commit contains three kind of changes: (1) a rename refactoring, (2) a refactoring for readability, and (3) the actual fault fix. The fault is in `jsexn.c`, in the function `Exception`. Comparing diffs of R1542 and R1543 we were able to identify lines 583, 587, 589, 590 and 594 as the actual faulty code. Before reduction, these lines (all with suspiciousness 0.716605) had rankings from 191 on (ranking is arbitrary among buggy lines of same suspiciousness). After reduction, the rankings improved to start at 98. The suspiciousness values of faulty lines, as discussed in Section III, remained unchanged. However, many higher ranked non-faulty lines became less suspicious after being removed from failing tests. The average coverage for an unreduced failing test was 11,461 lines, which decreased to 8,349 (an average decrease of 3,112 lines) after reduction.

## D. Open Source Projects

Next we applied reduction based localization to five open source Java programs (shown in Table IV) and generating mutants for each of the project to simulate bugs as it is commonly done in recent fault localization literature.[**?**] [**?**]

Our strategy was to create mutants in the same way as Xuan and Monperrus [**?**] in their own localization work, using 6 mutant operators. From each set of mutants generated, we selected 5 or 6 mutants at random that met the following criteria: (1) the mutant was killed by at least one test case and (2) the mutant generated no errors in JUnit test cases. A JUnit failure is caused by

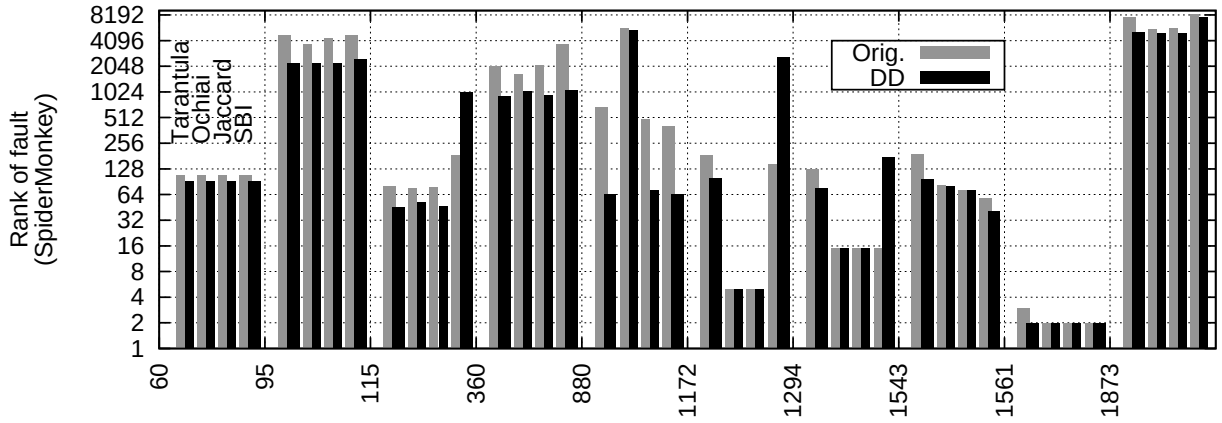


**Fig. 3. SpiderMonkey Results**

an unsatisfied assertion, but an error is caused by another kind of test failure, which may include some test setup or oracle problems. Using assertion failures only assures that we were remaining within the intent of the original tests.

Detailed results of applying reduction based localization to all these bugs are available on our website. Taking all the open source projects and mutants together, we note that reduction improved fault ranking in 51 cases, left it unchanged in 55 cases, and made it worse in 2 cases. The average improvement was 17.62 ranking positions; the average negative effect size was 2 ranking positions. The best improvement was 100 rank positions. The average fault ranking without reduction was 37.64 and with reduction it improved to 29.36.

### E. Threats to Validity

The primary threats to validity here are to external validity [**?**], despite our use of a larger number of faults and subjects than is usual in the literature. Our subjects are all C or Java programs, for example, and for the open source projects we used seeded rather than real faults. The all-too-frequent threats [**?**] of relying on results for a single formula or of making comparisons that are not across exactly duplicated subjects and code-bases, however, are absent from our study by design. To avoid construct threats, we developed independent experimental code-bases for some of the subjects, executed both, and compared results to cross-check the shared code base used for all subjects. Fortunately, most tasks here are straightforward (test execution, coverage collection, delta-debugging, and calculation of scores).

A second point (not strictly a threat) is that this paper focuses on single-fault localization. Even when there are multiple faults, it may be better to use techniques for clustering test cases by likely fault [**?**], [**?**], [**?**], [**?**] and then perform single-fault localization than to try to localize multiple faults at once.

## V. Conclusions

Our primary conclusion is that, when possible, anyone attempting to use spectrum-based fault localization should use delta-debugging to *reduce before localizing*. Across Siemens subjects, real Mozilla SpiderMonkey bugs, and mutants of a set of open source projects, reducing test cases before localizing was seldom harmful and in the cases where it caused harm the effect size was much smaller than in the cases where reduction was helpful. In most cases, reduction was helpful, and it was sometimes extremely effective, improving fault ranking by a factor of 2 (or more) and a very large absolute rank, sometimes hundreds of lines. This makes sense: if failing test cases only contained faulty code, fault localization would be trivial. Delta-debugging, by (usually) reducing the coverage of non-faulty code, approaches this ideal situation as best we know how at present. While delta-debugging is not a panacea for localization, in that it does not apply to some kinds of inputs and is sometimes not helpful, it often produces a very large improvement in localization effectiveness, quite often *more so than can be gained by switching from worst to best formula*.

Our larger take-away message is that the lessons of Parnin and Orso [**?**] should be taken to heart: rather than seek incremental improvements in localization effectiveness, we need large improvements in fault rank, and need to exploit all sources of information, not just coverage vectors. Even when reduction does not assist localization, we believe that the reduced test cases are highly valuable debugging aids. Furthermore, because no single formula is "best" for all faults [**?**], there is much to be gained by devising aids to fault localization that apply to any formula

and any type of spectrum. If automated fault localization is to be adopted in real-world settings, we need more than a competing set of ranking algorithms: we need a complete ecosystem for localization, debugging, and test understanding.