Mitigating (and Exploiting) Test Reduction Slippage

Josie Holmes
Department of Geography
Pennsylvania State University

Alex Groce
School of Electrical Engineering and Computer Science
Oregon State University

Mohammad Amin Alipour
School of Electrical Engineering and Computer Science
Oregon State University

ABSTRACT

Reducing the size of tests, typically by delta debugging or a related algorithm, is a critical component of effective automated testing and debugging. Automatically generated or user-submitted tests are often far longer than required, full of unnecessary components that make debugging difficult. Test reduction algorithms automatically remove components of such tests, while preserving the property that the test fails. Unfortunately, reduction can sometimes transform a failing test that detects a subtle, critical, and previously unknown fault into a test that detects a trivial-to-find, unimportant, and already known fault. When reducing a test detecting fault(s) F produces a test that does not detect the same F, this is known as *slippage*. In the case where an interesting fault slips to an uninteresting fault, slippage is a problem, and must be avoided. However, slippage can also be beneficial, when a long test can be reduced to detect a fault that has not otherwise been detected (including by the original test). While traditional delta debugging only produces one reduced test, the concept of slippage suggests an alternative approach, where the output of reduction is a set of reduced tests, in order to avoid problematic slippage and induce beneficial slippage. In this paper, we present preliminary efforts to understand slippage, and compare two approaches to slippage mitigation.

CCS Concepts

•Software and its engineering → Software testing and debugging;

Keywords

delta debugging, slippage, test manipulation and reduction

1. INTRODUCTION

Automated testing often goes hand in hand with test reduction or minimization [18, 11, 12, 6, 14]. Such reduction is now standard practice in industrial testing tools such as Mozilla's `jsfunfuzz` [16]. Many automated test generation methods, especially those based on random testing, produce long tests with many irrelevant components. Such tests are hard to understand for debugging and slow to execute. Delta debugging automatically reduces the size of such tests, producing a smaller test that still fails.

Unfortunately, the consequences of reducing a test are not always desirable. In particular, sometimes the unreduced test no longer detects the same faults it originally detected. This phenomenon is called *slippage* [3].

Formally, we define slippage as occurring whenever a test t detects faults F, and the reduction r of t detects a different set of faults, $F' \neq F$. Informally, most discussion of slippage is concerned with the case where $\exists f. f \in F \wedge f \notin F'$ — that is, when slippage causes loss of a fault. Slippage is usually avoided by using heuristics, such as that tests failing with the same error message or ending in the same step are due to the same fault [6]. In some cases, such as system invariant violations or compiler wrong-code bugs [3], such methods do not work well. The extent to which slippage is a practically important problem in real-world testing is unknown.

However, all slippage is not harmful. Even when a fault is lost, the new fault may be more interesting or harder-to-find. We suspect this is usually not the case, as slippage probably tends to favor easy-to-detect faults, but there is no hard data on the matter. Considering the possibility of beneficial slippage naturally leads to a new approach to mitigating slippage. In place of traditional delta debugging, which, given one failing test, produces one reduced failing test, we propose to modify delta debugging to produce a set of reduced failing tests. Ideally, this set exposes more than one fault, if possible. Throughout the remainder of this paper, we refer to the original, one-test, delta debugging algorithm [18] as `ddmin`.

We propose two novel approaches to slippage mitigation (and exploitation), both making use of any existing `ddmin` implementation, even (we believe) ones that are significantly different than the version of Hildebrandt and Zeller [18]. The contributions of this paper are:

1. A formal definition of slippage, and the notion that slippage may be either harmful or beneficial.
2. A proposal to avoid and make use of slippage by producing more than one reduced test per failing test.
3. Two novel approaches to slippage mitigation.
4. Some preliminary experimental results showing the basic effectiveness of these methods for a very large set

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

of mutant-simulated faults in a Python AVL tree and for a set of real-world tests and faults for a complex JavaScript compiler.

Both of our proposed methods reduce harmful slippage and result in detection of more faults.

2. SLIPPAGE MITIGATION APPROACHES

2.1 The Combinatorial Blocking Algorithm

The first algorithm uses additional unmodified `ddmin` runs, but "blocks" them from producing the same reduced test by modifying the test input to delta debugging. The algorithm is called combinatorial because it uses combinations of components in t to construct new starting tests, and blocking because the purpose of these combinations is to ensure that we do not get the same solution as the first run of `ddmin`, in a way that is analogous to the use of blocking clauses to force new solutions from SAT solvers.

Given test t that reduces to r, we compute all subsets of components of r, C_r. The set of reduced tests is then computed by running delta debugging starting with each $t-c$ that fails, for all $c \in C_r$: c is the blocked components of r. So long as even one component is blocked, it is impossible to reproduce the same r. The intuition is that to find a reduced failing test that exhibits a different fault than r, we want to run delta debugging in such a way as to produce a test as different as possible from r; ideally we would like a reduced test sharing no components with r. However, r will likely contain components that must appear in any failing test: for example, calls to `mount` appear in all useful file system tests, and interesting XML files seldom lack the `<` character. Therefore, rather than only trying to block all components of r, we try delta debugging with different combinations of components blocked.

In practice, iterating through all subsets may be too expensive if r is long. We therefore make a simplifying assumption: if $t - c_1$ does not fail, and $c_1 \subset c_2$, then $t - c_2$ also does not fail. The blocking algorithm begins its search by blocking all single components of r, then proceeds to all combinations of 2 components, etc., at each stage only considering combinations that contain no smaller combination that did not yield a failing test. With this optimization, the expense of blocking becomes low enough to also apply the approach to the new reduced tests found at each stage. To block all previously discovered reductions, it is necessary to compute combinations that include at least one component from each reduced test produced thus far.

Algorithm 1 shows the formal definition of the `comb-block` algorithm. This algorithm depends on a function `block-all (T,s)`, which given a set of tests T and a combination size s, returns all size s combinations of components of tests $t \in T$ such that each combination has at least one element from each $t \in T$. We omit the definition of `block-all` in the interests of space. Our Python implementation [9] simply filters invalid combinations, which works well with the typically small s for reduced tests.

Algorithm 1

Require: failing test t, reduced failing test r, search depth d, max combinations to consider m

```
reductions = {r}; count = 0;
handled, notfailed = ∅;
while d > 0 do
    new = ∅
    for s = 1 to total components in reductions do
        for c ∈ block-all(reductions-handled, s) do
            count = count + 1
            if count > m then return reductions
            end if
            handled = handled ∪ {c}
            if ¬∃c'.c' ∈ c ∧ c' ∈ notfailed then
                if fails(t − c) then
                    new = new ∪ {ddmin(t − c)}
                else
                    notfailed = notfailed ∪ {c}
                end if
            end if
        end for
    end for
    d = d − 1
    reductions = reductions ∪ new
end while
return reductions
```

2.2 Randomized Multiple-Run ddmin

Our second mitigation approach simply notes that slippage in `ddmin` is deterministic only because of the fixed order in which possible reductions of a test are considered. However, if we randomize the order of checks on smaller tests in `ddmin`, it can produce different reduced tests when executed with a different random seed. This means we can try to avoid slippage by simply running `ddmin` multiple times with different seeds. The advantage of this `multi-ddmin` approach is that is can work even when no test removing components of the original reduction fails. The non-existence of such tests does not mean there is no alternative reduction, but that finding it must require a different starting path for `ddmin`. This approach lacks `comb-block`'s directed search for dissimilar reductions and modifies the internals of `ddmin`.

3. EXPERIMENTAL RESULTS

3.1 AVL Tree

For basic experiments, we used a simple Python AVL tree found on the web [17], with 225 lines of code[1]. We used MutPy to generate 82 valid, non-equivalent (measured by applying random testing for one hour, and assuming mutants with no failing tests were equivalent) mutants of the AVL implementation. The test harness and reduction implementations used the TSTL [9, 8, 7] testing language for Python. The AVL tree forms a good simple basis for experimentation with slippage. It has a strong oracle, letting us ignore specification problems, but tests are all very similar, making the construction of pattern-based slippage avoidance mechanisms difficult. AVL is also sufficiently complex to allow for many different faults, some of which are fairly difficult to detect without effective automated testing.

Combining two mutants that cover different lines of code yields a version of AVL tree with two faults that are, we know, independently detectable (that is, we can find each fault in isolation). This results in a program to test with the potential for test slippage. There are 3,274 such com-

[1]All sizes non-comment, non-blank lines, by cloc [5].

Tests	Runtime(s)	$\|f\|_x$ (Faults)	# Tests
Unreduced tests	N/A	1.08	1
`ddmin`	0.41	1.05	1
`comb-block`	2.79	1.14	2.9
`multi-ddmin`	4.24	1.11	1.9

Table 1: Average runtimes, faults detected in isolation, and number of tests for a user to examine for AVL tree mutant pairs.

binations, which we used to explore slippage and perform a simple evaluation of our slippage mitigation algorithms.

For the mutant pair (m_1, m_2), let $m_1 + m_2$ be the program combining m_1 and m_2. We randomly sampled mutant pairs, and performed random testing on $m1 + m2$ for 60 seconds. In all but a very small number of cases, 60 seconds of random testing produces a failing test, t. Executing t on both $m1$ and $m2$ yields a number, $|f|_t$, between 0 and 2, the number of isolated faults t detects (2 if each fault can be detected in isolation by t, 1 if only 1 of the mutants is detected in isolation). In a small number of cases, t only detects the combined fault, and detects neither fault in isolation. We then applied delta debugging to produce a reduced test r, and used the same process to produce $|f|_r$, which could, compared to $|f|_t$, either be smaller (slippage with fault loss), larger (beneficial slippage), or the same (either no slippage or change of fault). Finally, we applied the `comb-block` algorithm to produce a set of tests, T_c and the `multi-ddmin` algorithm to produce a set of tests, T_m. For parameters, we chose to run `ddmin` 10 times in `multi-ddmin`, and constrained `comb-block` to approximately the same runtime by limiting it to a search depth of 5 and a maximum of 1,000 attempted combinations. We computed $|f|_c$ as the count of isolated faults detected by *any* test in T_c; $|f|_m$ was defined in the same way, except using T_m.

We produced 7,500 tests, sampling all 82 mutants numerous times, and sampling 2,959 of the 3,274 mutant pairs. Table 1 shows the results. All differences between means are significant by Wilcoxon test with $p < 10^{-12}$. The `ddmin` results show that using only a single run of classic delta debugging results in fewer faults per test than using un-reduced tests: harmful slippage results in a 2.8% total reduction in fault detection, even with beneficial slippage taken into account. Slightly over 8% of test cases had slippage of some kind when reduced.

Both approaches to slippage reduction produced a significantly higher average number of faults detected than the unreduced tests. Using `comb-block` increased fault detection by about 5.5%, and using `multi-ddmin` increased it by 2.7% (the same margin, interestingly, as a single reduction decreased fault detection). The cost of this improvement was close to a 6x increase in runtime for `comb-block` and over a 9x increase for `multi-ddmin`.

One concern with slippage mitigation is that avoiding slippage may produce a large number of additional tests a developer must examine to see if they expose different faults [3]. However, at least for our AVL example, the number of different tests produced is not particularly larger than the number of faults discovered on average. In most cases, we speculate that `comb-block` and `multi-ddmin` will not produce far more different tests than faults.

While only 8% of all tests showed slippage, we also ran some experiments with 30 runs for every mutant pair (producing a total of 98,100 failing tests), but only running `comb-block` and `multi-ddmin` if reduction caused loss-of-fault slippage, and checking whether the two approaches managed to restore the lost fault (without losing the other exposed fault). These results showed that 8% of mutant pairs had harmful slippage, and that these pairs had slippage rates up to 83%, making some faults very hard to detect in isolation without slippage mitigation. The `comb-block` approach avoided loss-of-fault slippage for 56% of slipped tests for each mutant, on average, while `multi-ddmin` only avoided it in 33% of slipped tests, on average. The difference was significant with $p < 10^{-96}$.

3.2 SpiderMonkey

We also performed limited experiments with the Mozilla SpiderMonkey JavaScript engine version 1.6 tests produced by `jsfunfuzz`, used in the PLDI 2013 paper introducing the concept of slippage [3]. SpiderMonkey is a large (about 70KLOC), complex, widely used program, and `jsfunfuzz` [15] has been used to discover over 1,700 bugs in JavaScript engines. For 113 randomly sampled tests, using a single reduction produced slippage only about 2.7% of the time (in exactly 3 tests). In these experiments we limited `comb-block` to a depth of 1, with no "recursive" exploration of new tests. For `multi-ddmin` we used 10 reductions, as in the AVL experiments.

The small number of slippage cases allowed us to analyze the results in detail. The `comb-block` approach did not manage to avoid slippage in any of these three cases. For two of the three cases, no blocking combination produced a failing test. In the third case, one combination did produce a failing test, but it reduced to the same fault as the slipped `ddmin` test. The `multi-ddmin` algorithm fared only slightly better, restoring the originally detected fault in one of the three slippage cases.

Interestingly, the overall rate for slippage in the original paper is estimated as 23%, not under 3%. We attribute this to the use of a far more aggressive and powerful reduction algorithm than our line-based `ddmin`. The PLDI 13 reducer is a hand-crafted modification of `ddmin` that performs both character and line based passes, and applies constant propagation and other specialized techniques for JavaScript. Note that both of our mitigation techniques also apply to this method, though `comb-block` is probably easier to apply as it does not require modifying the internals of the more powerful reducer.

Slippage avoidance is not the only goal of our methods. We also computed the total number of discovered faults for `comb-block` and `multi-ddmin`, as with the AVL tree experiments. One-test `ddmin`, in this setting, always produced a single test, detecting one fault. The `comb-block` algorithm discovered 1.34 different faults on average, and the `multi-ddmin` approach discovered 1.09 different faults on average. These differences are significant by Wilcoxon test with $p < 10^{-5}$. Note that the strong results for `comb-block` are in spite of nearly 70% of tests not having any failing blocked versions, and using a search depth of only 1 (no attempt to block based on newly discovered tests). Blocking seems quite effective in producing tests due to a different fault, when this is possible. The total set of distinct faults discovered by each method favored `multi-ddmin`, which produced tests revealing 10 distinct faults, while `comb-block` discovered 8 distinct faults over all 113 tests. The original

set of unreduced tests contains 8 distinct faults, and simple `ddmin` reduced the count to 7 faults.

In part because many tests do not have any combinations that produce a failing test when removed, the runtimes for `comb-block` are generally much lower than for `multi-ddmin`, which takes about 10x or more the time for a single-test `ddmin` run (the average is higher than 10x because of the structure of JavaScript tests, where the first attempt in non-randomized `ddmin` is very likely to fail).

Given the (surprising to us) extreme differences in runtimes observed for SpiderMonkey tests, and the smaller but real differences for AVL tree, in future experiments we plan to exploit the "best effort" nature of both mitigation approaches, and simply use a fixed timeout for each method.

4. RELATED WORK

Chen et al. introduced the idea of slippage in the course of describing efforts to automatically detect different faults in a large set of failing test cases [3]. Hughes et al. [10] proposed a modification of QuickCheck [4] to avoid re-producing known bugs that (in theory) could mitigate the problem of slippage, but is not directly comparable to our approach. The approach of Hughes et al. requires interpretation of test components (e.g. method calls), and analysis of patterns, while our approaches are purely algorithmic, with no additional requirements beyond those of delta debugging itself [18]. It is not clear how best to apply such an approach to cases such as `jsfunfuzz` where each component is not a method call but essentially an arbitrary string, without significant user effort to define abstractions of components.

There are also approaches that sidestep slippage by initially producing short test sequences (e.g. recent work by Mao et al. [13]). However, for many generation algorithms longer sequences are essential for good fault detection [1, 2].

5. CONCLUSIONS AND FUTURE WORK

In this paper we describe our first efforts to further investigate the phenomenon of slippage, where reducing the size of a failing test also changes the reason that the test fails. While generally noted as a problem, we show that slippage can also be considered an opportunity to extract more distinct faults from a single failing test. We present two approaches, `comb-block` and `multi-ddmin`, that modify traditional delta debugging to return multiple reduced tests. Preliminary experiments on a Python AVL tree and on Mozilla's SpiderMonkey JavaScript engine show that these algorithms can, at relatively modest cost, significantly improve the number of distinct faults detected based on a single failing test.

As future work, we plan to determine the degree to which harmful slippage is a real problem (not solved by simple heuristics) in real-world testing efforts, and further investigate the causes of such slippage. For example, is slippage overwhelmingly a matter of hard-to-detect faults reducing to easy-to-detect faults? We also plan to further refine and evaluate our slippage mitigation approaches. Most importantly, we want to investigate whether encouraging slippage (via multiple delta debugging runs) is an efficient way to increase fault detection for a testing effort.

6. REFERENCES

[1] J. H. Andrews, A. Groce, M. Weston, and R.-G. Xu. Random test run length and effectiveness. In *Automated Software Engineering*, pages 19–28, 2008.

[2] A. Arcuri. Longer is better: On the role of test sequence length in software testing. In *International Conference on Software Testing, Verification and Validation*, pages 469–478, 2010.

[3] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr. Taming compiler fuzzers. In *Programming Language Design and Implementation*, pages 197–208, 2013.

[4] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of haskell programs. In *ICFP*, pages 268–279, 2000.

[5] A. Danial. CLOC: Count lines of code. https://github.com/AlDanial/cloc.

[6] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *International Conference on Software Engineering*, pages 621–631, 2007.

[7] A. Groce and J. Pinto. A little language for testing. In *NASA Formal Methods Symposium*, pages 204–218, 2015.

[8] A. Groce, J. Pinto, P. Azimi, and P. Mittal. TSTL: a language and tool for testing (demo). In *ACM International Symposium on Software Testing and Analysis*, pages 414–417, 2015.

[9] A. Groce, J. Pinto, P. Azimi, P. Mittal, J. Holmes, and K. Kellar. TSTL: the template scripting testing language. https://github.com/agroce/tstl.

[10] J. Hughes, U. Norell, N. Smallbone, and T. Arts. Find more bugs with QuickCheck! In *Workshop on Automation of Software Test*, pages 71–77, 2016.

[11] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *International Symposium on Software Reliability Engineering*, pages 267–276, 2005.

[12] A. Leitner, M. Oriol, A. Zeller, I. Ciupa, and B. Meyer. Efficient unit test case minimization. In *International Conference on Automated Software Engineering*, pages 417–420, 2007.

[13] K. Mao, M. Harman, and Y. Jia. Sapienz: Multi-objective automated testing for android applications. In *International Symposium on Software Testing and Analysis*, pages 94–105, 2016.

[14] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for C compiler bugs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 335–346, 2012.

[15] J. Ruderman. Introducing jsfunfuzz, 2007. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/.

[16] J. Ruderman. Releasing jsfunfuzz and DOMFuzz. https://www.squarefree.com/2015/07/28/releasing-jsfunfuzz-and-domfuzz/, 2015.

[17] user1689822. python AVL tree insertion. http://stackoverflow.com/questions/12537986/python-avl-tree-insertion.

[18] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *Software Engineering, IEEE Transactions on*, 28(2):183–200, 2002.