Using Test Case Reduction and Prioritization to Improve Symbolic Execution

Chaoqiang Zhang, Alex Groce, Mohammad Amin Alipour
School of Electrical Engineering and Computer Science
Oregon State University Corvallis, OR USA
{zhangch, alex, alipour}@eecs.oregonstate.edu

ABSTRACT

Scaling symbolic execution to large programs or programs with complex inputs remains difficult due to path explosion and complex constraints, as well as external method calls. Additionally, creating an effective test structure with symbolic inputs can be difficult. A popular symbolic execution strategy in practice is to perform symbolic execution not "from scratch" but based on existing test cases. This paper proposes that the effectiveness of this approach to symbolic execution can be enhanced by (1) reducing the size of seed test cases and (2) prioritizing seed test cases to maximize exploration efficiency. The proposed test case reduction strategy is based on a recently introduced generalization of delta-debugging, and our prioritization techniques include novel methods that, for this purpose, can outperform some traditional regression testing algorithms. We show that applying these methods can significantly improve the effectiveness of symbolic execution based on existing test cases.

Categories and Subject Descriptors: D.2.5 [Software Engineering]: Testing and Debugging

General Terms: Reliability, Experimentation

Keywords: Symbolic execution, Test case reduction, Test prioritization

1. INTRODUCTION

Symbolic execution has been one of the most promising and exciting areas of automated testing research for many years now [9]. Symbolic execution "runs" a program, replacing concrete inputs with symbolic variables that represent all possible values. When a program branches, the execution takes both paths (if they are feasible under current constraints) and a set of path conditions on symbolic variables is modified for each path to record the new constraints on the symbolic values. Constraint solvers can be used to produce a concrete input from a symbolic path or to check safety of operations (determining if, for example, current path constraints ensure that a division is applied to a non-zero value). Dynamic symbolic execution combines symbolic execution with concrete execution in order to improve scalability [9]. However, the promise of (dynamic) symbolic execution-based testing has been frustrated by scalability problems, including the path explosion problem and the challenge of solving for complex constraints. In terms of program paths generated per unit of testing time, symbolic execution can be a much less efficient approach to automated testing than explicit-state model checking or random testing [13]. For some large programs and generalized symbolic harnesses, symbolic execution basically does not yet work.

One popular method for addressing these problems is to not perform symbolic execution "from scratch" using a highly generalized test harness, but to seed symbolic execution from existing inputs, augmenting an existing suite [18, 19, 29, 30]. For example, the `zesti` (Zero-Effort Symbolic Test Improvement) extension of KLEE is based on the observation that symbolic-execution can take advantage of regression test cases [23]. Other work, motivated by a desire to efficiently reproduce field failures, has shown that symbolic execution can scale better based on a "test case" (a call sequence) than when using only a point-of-failure or call stack [17]. The best-known and most successful large-scale application of symbolic execution to real-world testing, Microsoft's SAGE, is based on seeded exploration from test inputs [2].

Previous investigations have considered the question of how best to symbolically augment a test suite for purposes of dealing with *changes* to the code [24, 30]. However, the literature does not, to our knowledge, address the problem of seeded symbolic execution in general, aimed at improving an existing test suite for a fixed program. Because of its high cost as a pure technique and its improved effectiveness given a base of test cases, dynamic symbolic execution may best fit into a **two-stage** testing process. In the first stage, a much less expensive automated testing approach, such as random testing [14, 25] is used to generate an initial suite of test cases until coverage saturates. Remaining uncovered branches at this point are likely to be difficult enough to cover to justify the cost of symbolic execution. In many cases, the probability of coverage for some reachable branches by randomized methods is essentially zero. The goal of the two-stage approach is to maximize total code coverage as rapidly as possible, under the assumption that this will result in effective and efficient fault detection.

The first stage of this two-stage approach to testing is difficult to generalize: the optimal approach depends heavily on the testing method used and the structure of the test

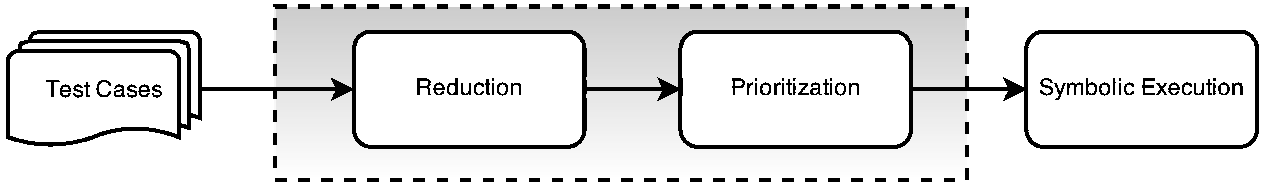


Figure 1: Workflow of this approach

space. This paper considers the more well-scoped problem of improving the effectiveness of the second stage: *given a suite of existing test cases, how can we improve the efficiency of seeded symbolic execution based on those test cases?* The evaluation criteria for proposed methods is simple: given a total testing budget B, how can additional code coverage over the starting suite be maximized?

Two basic approaches are possible. First, the test cases themselves may be modified to improve their suitability as a basis for symbolic exploration. Symbolic techniques are generally more expensive to apply to larger seeds, for the obvious reasons (the set of paths is larger, the program state becomes more complex, and there are more symbolic inputs to solve for). It is therefore reasonable to expect that reducing the size of input test cases should improve the scalability of symbolic exploration. The basic purpose of seeded symbolic augmentation is arguably to explore the "coverage neighborhood" of a test input, which leads to the hypothesis that a test input of smaller size but with the same code coverage will almost always lead to more efficient symbolic exploration than a larger test that does not increase coverage over the smaller test. Delta-debugging [33] is a well-known technique for reducing the size of test cases, but it is only useful for failing test cases, and often reduces test cases so much that they are a poor basis for symbolic exploration. However, recent work has shown that delta-debugging using *preservation of source code coverage* rather than failure as a property to be preserved can produce extremely efficient regression suites based on existing tests [12].

Second, seeded symbolic exploration can be seen as a powerful variation of regression testing, which suggests the use of test case prioritization [32]. For example, an obvious approach is to attempt to run regression tests in such an order that total code coverage is maximized as quickly as possible. The hypothesis is therefore that better ordering of the test cases used as symbolic seeds can improve the efficiency of symbolic execution as well. Because symbolic exploration is much more demanding than simple replay of regression tests, some prioritizations that are not obviously useful for traditional regression testing (in that they do not aim for high code coverage) are considered. The KATCH tool [24] uses a novel prioritization for seeded exploration based on both CFG distance and weakest preconditions to target changed code in patches; our aim is similar, but intended for the general case of increasing overall coverage over an existing test suite base.

The basic workflow for seeded symbolic execution proposed in this paper is therefore (as Figure 1 illustrates):

1. Generate a suite of seed test cases $(T_1, T_2, \ldots, T_n)$ by either a cheap automated method or manual testing.

2. Apply delta-debugging with code coverage as a reduction criteria to produce a new set of test cases $(T'_1, T'_2, \ldots, T'_n)$ with reduced size and execution time but with equivalent code coverage to the original suite.

3. Prioritize $(T'_1, T'_2, \ldots, T'_n)$ and in rank order for each test case T'_i (or for multiple T'_i at once, in parallel to the extent of available computational resources) perform symbolic exploration on T'_i. Symbolic exploration consists of two steps:
   (a) Execute T'_i and collect all possible coverage divergence points (untaken branches).
   (b) Apply a symbolic execution search strategy based on these divergence points using T'_i as a seed.

This paper focuses on examining how steps 2 and 3 of this process impact the effectiveness of the symbolic execution effort, measured in terms of improved incremental branch coverage for a time budget and the rate of coverage improvement. The primary contributions of this paper are (1) introducing methods for reducing and prioritizing test cases for seeded symbolic execution and (2) experimentally demonstrating that these methods are effective in improving the efficiency of symbolic execution for 6 real C programs of moderate to large size. A secondary contribution is the proposal that in general a two-stage framework may address some of the difficulties with scaling symbolic execution.

2. TEST CASE REDUCTION

The goal in our framework of using symbolic execution based on test cases is to improve the total program coverage. Seeded symbolic execution takes a test case and attempts to cover new program behavior based on it; for example, if a conditional is false in the original test case, and the symbolic engine finds the true branch to be feasible, it explores to find a test case covering the true branch. The essential quality of the seed test suite is therefore its code coverage. In general, given two test cases similar other than in length, symbolic execution has more difficulty with the longer test case, due to path explosion and increasingly complex constraints as the program executes. Therefore, given two test cases with the same code coverage, it seems likely that symbolic exploration based on the "smaller" test case will be more effective, though it is possible that the change *could* reduce the value of the seed test case (for example constraints may change and branch feasibility in context may change). *Cause reduction* allows us, given a test case T to use delta-debugging [16] to produce a new test case T' such that T and T' have the same code coverage, and removing any component of T' will result in lower code coverage [12]. Cause reduction uses the usual *ddmin* algorithm of delta debugging, but replaces the check that reduced tests fail with a check that a reduced test covers at least all statements covered by the original test case. In experiments with the SpiderMonkey JavaScript engine, a suite of tests where each test was reduced while preserving statement coverage surprisingly showed *better* fault detection than the original suite, and statement coverage-based reduction was shown to preserve other important properties well (test case failure, branch coverage, and mutation kills) [12].

Our test case reduction experiments apply cause reduction to an initial seed test suite $(T_1, T_2, \ldots, T_n)$ to produce $(T'_1, T'_2, \ldots, T'_n)$ such that $\forall i. SC(T_i) = SC(T'_i)$, where SC is statement coverage. It is possible for T_i to be equal to T'_i, but usually T'_i is smaller and executes faster than T_i, as results below show. Cause reduction of a test case often takes a relatively long time to produce a 1-minimal [16] test case. Usually the majority of the reduction achieved is obtained early, and the remaining computation produces little reduction. Unlike traditional delta-debugging, the purpose is not to remove all redundancy to aid human readers, but to make testing more efficient, so total minimality is not important. All experiments therefore use a limited budget for cause reduction, and return the smallest coverage-equivalent test discovered before timeout. The details of test case components and cause reduction implementation differ from program to program. In truth, any mature automated testing infrastructure *should* include a delta-debugging module to enable effective debugging [20], and a delta-debugger is easily modified to implement statement coverage-based cause reduction.

For example, a test case for the `grep` utility includes an input pattern string and an input file:

```
grep 'patternstring' inputfile.inp
```

To apply cause reduction to grep, our implementation first holds `file1.inp` constant, and uses simple character-based delta-debugging to find a (smaller) `patternstring` that results in the same statement coverage. The process is then repeated, holding `patternstring` constant and reducing `inputfile.inp`. The pattern string is reduced first because it is typically shorter and therefore less likely to result in a timeout. For test cases that are sequences of API calls (e.g. the `YAFFS2` file system or a container class), we can simply use each API call as a component and run one reduction. Note that while we count cause reduction against test budgets in our experiments, in practice projects might maintain cause reduced tests as fast regressions ("quick tests") [12]. In all cases, we used statement coverage-based reduction, even though symbolic execution is based on branches. Statement coverage-based reduction in practice is much faster than branch coverage-based reduction, but tends to preserve branch coverage well in the final result [12].

3. TEST CASE PRIORITIZATION

Test case prioritization orders test cases with the goal of finding faults in regression testing as quickly as possible [32]. For seed-based symbolic execution, we can also order the seeds, with the goal of gaining incremental coverage as quickly as possible. A large body of work exists on prioritization methods for regression testing, and it is not our purpose to explore all possible strategies. Instead, we aim to demonstrate first, that ordering is important and second, that some approaches that might not be obviously useful in traditional regression testing are effective for seed-based symbolic execution.

We considered five ranking strategies, some drawn from the simpler regression approaches and two that were devised with symbolic execution in mind:

- **Random Ordering (Random):** Our first prioritization is not in fact a proposed solution but a baseline to measure the importance of ordering. Random ordering simply chooses a random permutation of test cases.
- **Branch Total (BT):** Prioritize test cases by their branch coverage such that the test with the highest branch coverage is used as a seed first. Ties are broken randomly.
- **Branch Additional (BA):** Prioritize by incremental branch coverage over all tests previously ranked. The first test chosen is therefore the same as with the BT method, but further tests are selected by the additional coverage provided, not absolute coverage.
- **Furthest-Point-First (FPF)**: The first test ranked is chosen randomly. Future test cases are selected by computing the minimum distance to all already-ranked tests, and adding the test case with the largest minimum distance to the ranking. The distance function in our case is the Hamming distance between the two test cases' branch coverage vectors; even if a test does not produce incremental branch coverage it can therefore be ranked highly if it executes a very different set of branches than any other test.
- **Shortest Path (SP):** Tests are ranked in order of increasing path length. The number of branches taken during execution is counted, and paths that execute fewer total branches (where the same branch executed multiple times counts each time) in the source of the tested program are chosen first.

While BT and BA are selected as examples of simple coverage-based prioritizations from the regression literature, FPF and SP require more justification, since in theory they might well tend to select test cases with poor total or incremental coverage. In fact, SP seems guaranteed to prioritize test cases that have worse branch coverage, on average!

The motivation for FPF [10] is that while branch coverage is important to symbolic execution based on a seed test case (if a branch is not covered, its divergences will obviously be hard to explore), the context in which branches are executed is also important. We would like to explore from tests that, in some sense, have very different behavior. Unfortunately, defining a generalized, cheap-to-compute, behavioral distance metric for program executions is difficult [11]. The hope is that simply executing different branches together can indicate behavioral difference in test cases well enough to serve as a useful ordering. The use of the FPF algorithm is inspired by efforts in *fuzzer taming*, which seeks to rank a set of failing test cases such that test cases exposing different underlying faults are ranked highly [4]. Basically, FPF aims to maximize the *diversity* (as defined by a distance metric) of test cases for the first N ranked test cases. The chosen distance metric is different than any used in fuzzer taming [4] because the aim is general diversity, not focused on failing test cases (and the inputs do not have such meaningful tokens as program source code). The hypothesis is that despite not necessarily prioritizing high coverage tests, FPF will produce good results by somewhat diversifying the behavior of executions.

The shortest path method is motivated by a simpler concern. Seeded symbolic execution can scale very poorly to long test cases, in our experience. It may be that simply choosing short tests, as measured in terms of potential divergence points, is the best possible ordering, since each symbolic exploration run will have more chance of exploring a large neighborhood of a test case if path explosion is limited and constraints are simpler.

Obviously, there are a great many plausible ranking strategies; our FPF ranking alone is simply one example from a vast family of different rankings based on different metrics. The purpose of this paper is not to exhaustively explore the space, but to establish that ranking is useful, even with possibly sub-optimal ranking techniques, and to show whether prioritizations that are not similar to ones used (to our knowledge) in regression prioritization can be useful due to the nature of symbolic execution.

4. EXPERIMENTAL METHODOLOGY

This paper considers two related research questions. First, in seed-based symbolic execution, the effectiveness of exploration depends on the seed test case itself, and some complex/large seeds can perform poorly. Second, each exploration takes significant time, and we would like to quickly improve coverage. The research questions are therefore:

- **RQ1:** *Can test case reduction improve the effectiveness of seeded symbolic execution?*
- **RQ2:** *Given a fixed search time for each seed test case, can ranking seed tests improve the efficiency of symbolic execution?*

Because measuring actual faults detected is highly sensitive to the set of faults used, and makes it difficult to obtain statistical significance for results [1], our basic measure is incremental gain in branch coverage over an initial suite, for a fixed testing budget. There is considerable evidence that when branch coverage for two test suites differs by a significant amount, the suite with higher coverage is usually better at fault detection as well [8]. This approach combines both effectiveness (since the absolute coverage numbers can be compared given the fixed computation effort) and efficiency (in that results are for a given time), and matches the practical needs of testing, where budgets are not infinite and finding bugs sooner rather than later is critical. For prioritization, we follow previous practice and modify this basic evaluation measure to examine the average incremental coverage, thus capturing the climb of the discovery curve for new branches produced by a prioritization strategy (since once all tests have been completed, all prioritizations have the same final coverage results). In all of the experiments, the computation time required for evaluated approaches is counted against the total budget for testing; e.g., if a test case is minimized, the time to minimize is taken away from the time spent in symbolic execution. Because of the large number of test cases involved in each experiment (100) and the large number of experimental treatments run, we limited our time budgets to 10 and 20 minutes per test case (and used a large compute cluster to run symbolic execution instances in parallel).

All experiments are based on the `zesti` version of KLEE, which can combine seeded symbolic execution with KLEE search strategies. For a given test input, this approach executes the test case concretely and symbolically, and for each branch determines feasibility of the negated branch in the symbolic execution. Feasible branches are recorded along with their path constraints for a second-stage pure symbolic search using a chosen search strategy. Because the effectiveness of symbolic execution can vary depending on the search strategy, four KLEE-supported strategies are used in all experiments [3, 21]. These are:

Table 1: Subject programs used in the evaluation

Subject	NBNC	LLVM instructions	# test cases
Sed	13,359	48,684	370
Space	6,200	24,843	500
Grep	10,056	43,325	469
Gzip	5,677	21,307	214
Vim	107,926	339,292	1,950
YAFFS2	10,357	30,319	500

- **Depth-First Search (DFS)**: DFS always continues from the latest execution state when possible. DFS, as in model checking, has the advantage of very low overhead in state selection but can become trapped in parts of the state space, e.g., when there are loops with few statements but that may execute many times.
- **Random State Search (RSS)**: as the name suggests, RSS selects a random state for exploration. The advantages are uniform exploration and avoiding the traps of DFS; however, RSS can repeatedly generate test cases similar to those already produced.
- **Random Path Selection (RPS)**: RPS uses a binary execution tree (where nodes are fork points and leaves are current states) to record explored parts of a program. RPS randomly traverses this structure, which advantages leaves high in the tree, motivated by the probability that these have fewer constraints and may be more likely to hit uncovered behavior. RPS can, like RSS, produce many similar test cases.
- **Minimum Distance to Uncovered (MD2U)**: This approach uses heuristics to prioritize states that are "likely" to cover new code soon, based using on factors including minimum distance to an uncovered instruction and query cost to produce a weight for each state. MD2U may not perform well for every program, like any heuristic strategy.

4.1 Subject Programs

Programs: Table 1 summarizes the programs used in our experiments, showing the name and number of NBNC (non-blank, non-comment) lines of code (measured by CLOC [5]) for each program. We used a total of six C programs, five of which are taken from the SIR repository [6]. The other subject, `YAFFS2` [31], is a widely used open-source flash file system for embedded devices (the default image format for earlier versions of Android). In addition to CLOC, we also show the number of LLVM instructions for each subject program.

Tests Cases: Table 1 also shows the total number of test cases in the test pools from which various test suites are composed. We use the SIR pools for the programs from SIR.

For YAFFS2, we generated random tests using feedback [14]. We did not use swarm testing [15] (our usual approach) due to concern swarming would make tests too easy to reduce and diversify, and thus unfairly advantage our methods. For YAFFS2, we first generated 500 random API-call sequences of length 50, then changed all input parameters into symbolic ones. In our initial experiments, we found that KLEE can finish the seeding stage only for 48 or 55 out of these 500 test cases (for 10 or 20 minutes search respectively). This means that KLEE could not perform its regular search for most of the test cases. We limited the maximum seeding duration for YAFFS2 to half of the total symbolic execution time. For the remainder of the programs, seeding time was unlimited (up to the total symbolic execution budget).

4.2 Experimental Setup

The program inputs for `sed`, `grep`, `space`, and `gzip` are command line parameters. Each parameter is either a command line option or input file. We use `zesti` [23] for our symbolic execution, which can automatically accept command line parameters as symbolic inputs. The symbolic input size is decided by the concrete input size. Take `grep` as an example:

```
klee -zest grep.bc ‘patternstring’ file.inp
```

Given this command, `zesti` will use one symbolic string and a symbolic file as program inputs, and use ‘patternstring’ and `file.inp`’s contents as search seeds. With these original inputs, we use KLEE to do symbolic search around the seeds with the four search strategies noted above. For `vim` seed inputs are 1,950 script files taken from the SIR repository, called as `vim -s scriptfile`. We have `zesti` take the concrete script files as search seeds.

5. EXPERIMENTAL RESULTS: RQ1 (TEST CASE REDUCTION)

5.1 Test Case Reduction Rate

The first question is whether test cases can be reduced significantly while preserving coverage; if there is no redundancy with respect to coverage in test paths, it is unlikely the answer to RQ1 will be affirmative. Previous work on such reduction used only randomly generated tests, which are known to be highly redundant, but SIR subjects include many test cases produced by other methods, including by human authors. Our subject programs also have various test case structures, including method call sequences, pattern search strings, text files, etc., while previous work was essentially based on API call sequences. We use path length as the measurement of the size of each test case. Table 2 shows path lengths before reduction, after reduction, and reduction rates for all the subject programs. We used a 20% of exploration time timeout (2 minutes for 10 minute budget, 4 minutes for 20) for all subjects. We define

$$reduction\ rate = \frac{PL_1 - PL_2}{PL_1}.$$

where PL_1 and PL_2 denote the path length of test case before and after reduction, respectively. For each subject program, we also show the minimum, median, and maximum path lengths and reduction rates over all test cases. Clearly, tests can be considerably reduced. Does this reduction translate into more efficient symbolic exploration?

5.2 Test Case Reduction Effectiveness

Table 3 shows that reduced test cases can in fact improve symbolic execution efficiency compared to the unreduced test cases. For each input test case, we asked KLEE to perform symbolic search for 10 and 20 minutes. From the test case pool, we randomly chose 100 test cases as a test suite 150 times, for each subject. For each such suite, C_0 is the total branch coverage of the original 100 test cases. The set of all branches explored during symbolic execution of the 100 original unreduced test cases is C_1, and $C_1 - C_0$ denotes the set difference of C_1 and C_0 — the new branches discovered during symbolic exploration. The same approach is applied to the reduced test cases and the branches hit are referred to as C_2. To see if reduced test cases improve symbolic execution, we simply compare $|C_1 - C_0|$ and $|C_2 - C_0|$. In Table 3, the columns “before” and “after” under each search strategy are the sizes of $C_1 - C_0$ and $C_2 - C_0$ respectively.

In addition to examining each search strategy, we also combined the search results from all the four search strategies. For each test case, we computed the total branch coverage acquired from all the four search strategies over original and reduced test cases. These comparison results are in the last column of Table 3. In all experiments, the value in the table is the mean of 150 runs, and p-values are based on a Wilcox rank sum test. The few cases where p is not well below 0.00 (rounded) are shown in bold. The best results are highlighted. It is clear that in general, for 10 minute and 20 minute test budgets, spending some portion of the budget reducing test cases usually results in markedly improved incremental branch coverage. The best timeout for reduction is unclear, but the basic validity of using test case reduction to improve symbolic exploration is difficult to ignore. In many cases the difference in median incremental covered branches is also quite large in absolute or relative terms, 50+ branches or from 40-100% more branches, in addition to being statistically significant. Figure 2 shows the improvements (with random ordering of test cases) for `sed`.

6. EXPERIMENTAL RESULTS: RQ2 (TEST CASE PRIORITIZATION)

6.1 Test Case Prioritization Cost

Table 4 shows the ranking cost for 100 randomly chosen test cases for all subject programs. The cost consists of two parts: the first part is from collecting branch coverage information for each test case. We use `gcov` to obtain branch coverage. The second cost is in using the coverage vector to compute the distance between each pair of test cases and rank all test cases using relative distances. As Table 4 shows, ranking test cases is a low cost operation compared to symbolic execution, a neccessary requirement for an affirmative answer to RQ2.

6.2 Test Case Prioritization Effectiveness

To evaluate if prioritizations perform better than random ordering, we adapt the Average Percentage Faults Detected (APFD) [26, 27] measure, which is used extensively in test case prioritization evaluations. Although we examine branch coverage rather than fault detection, the same method works

[1]The reason branch coverage for 10 minutes is higher than for 20 minutes is that KLEE crashed in some 20 minute runs, losing some coverage data.

Table 2: Reduction rates

Subject	Path length before reduction			Path length after reduction			Reduction rate (%)		
	Min	Median	Max	Min	Median	Max	Min	Median	Max
Sed	6	88,723	496,723	6	6,167	80,418	0.00	93.50	98.90
Space	530	4,154	24,147	530	3,798	18,199	0.00	8.18	79.74
Grep	740	103,097	622,223	691	26,466	424,388	0.00	73.57	98.85
Gzip	24	752,257	36,351,281	24	231,629	1,732,247	0.00	59.65	99.99
Vim	201,222	221,219	481,749	201,083	213,957	475,421	0.00	2.43	50.15
YAFFS2	32,632	53,139	91,252	23,719	40,339	71,942	2.00	23.45	50.40

Table 3: Branch coverage increment (mean values over 150 test suites) on 100 random tests with rounded Wilcoxon p-values (Before: Before test case reduction, After: test case reduction)

Subject	Time	DFS			Random path			Random state			MD2U			Combined		
		Before	After	p	Before	After	p	Before	After	p	Before	After	p	Before	After	p
Sed	10m	150.66	211.73	0.00	265.03	274.46	0.00	213.75	248.56	0.00	211.42	249.59	0.00	279.65	289.00	0.00
	20m	163.15	229.53	0.00	298.83	292.77	0.00	229.23	267.65	0.00	239.15	265.14	0.00	311.49	302.33	0.00
Space[1]	10m	3.21	9.18	0.00	5.00	5.33	0.00	3.58	6.77	0.00	3.58	7.24	0.00	5.00	9.26	0.00
	20m	2.24	10.81	0.00	4.23	5.13	0.00	2.75	6.78	0.00	2.75	6.71	0.00	4.23	10.82	0.00
Grep	10m	20.73	160.37	0.00	120.70	139.35	0.00	116.22	183.65	0.00	112.65	189.71	0.00	157.87	212.66	0.00
	20m	21.13	185.68	0.00	177.65	201.23	0.00	114.07	205.58	0.00	110.14	209.36	0.00	194.85	232.85	0.00
Gzip	10m	93.35	103.95	0.00	220.55	226.33	0.00	113.10	129.07	0.00	134.71	158.12	0.00	222.59	228.77	0.00
	20m	153.66	157.25	0.00	233.47	236.50	0.00	176.10	182.41	0.00	193.81	193.89	**0.56**	239.44	242.59	0.00
Vim	10m	312.17	310.36	**0.39**	111.71	116.44	0.00	302.77	308.35	0.00	357.60	365.79	0.00	540.42	542.99	**0.93**
	20m	513.45	558.27	0.00	118.60	123.79	0.00	345.81	358.17	0.00	421.97	442.37	0.00	769.95	821.35	0.00
YAFFS2	10m	78.14	76.28	0.00	98.21	100.18	0.00	93.40	104.80	0.00	93.99	105.27	0.00	115.47	125.35	0.00
	20m	78.54	79.51	0.02	99.15	100.47	0.00	94.58	103.89	0.00	95.09	104.33	0.00	117.98	126.39	0.00
The best results		1	10	-	1	11	-	0	12	-	0	11	-	1	10	-

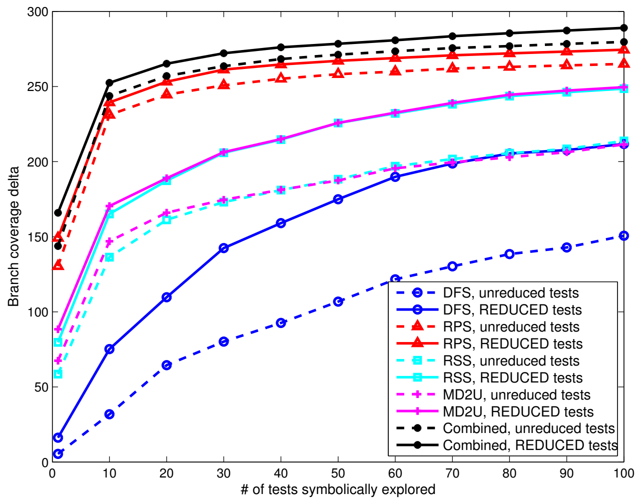


Figure 2: Additional branch coverage on sed program during seeded symbolic execution with unreduced tests and reduced tests (test cases are in random order)

well for comparing two *curves* to see which one converges faster. Average Percentage Branches Discovered (APBD), our measure, is computed as [26]:

Table 4: Ranking cost for 100 test cases for all subject programs in seconds

Subject	Coverage collection	FPF	SP	BA
Sed	17.3	1.3	< 0.1	1.0
Space	12.0	1.1	< 0.1	0.8
Grep	19.5	1.5	< 0.1	1.3
Gzip	43.7	0.7	< 0.1	0.6
Vim	104.8	8.7	< 0.1	7.7
YAFFS2	21.9	1.7	< 0.1	1.5

$$APBD = \frac{\sum_{i=1}^{n-1} BC_i}{nm} + \frac{1}{2n}.$$

Here, n is the number of the test cases, m is the total newly covered branches from symbolic execution, and BC_i is the number of branches newly covered by symbolic exploration of at least one test case in the first i test cases. Note that because ABPD compares curve gains rather than absolute values, it is only useful within orderings of the same total result. For example, the APBD might be better for a 10 minute symbolic execution time limit than for a 20 minute symbolic execution time limit because while the final total branches explored by the 20 minute execution will almost certainly be higher, the *percent* gain might be better

Table 5: Average Percentage Branches Discovered (APBD) values with different prioritization methods on 100 random selected test cases (Rnd: Random, FPF: Furthest Point First, SP: Shortest Path, BA: Branch Additional) (percentage %)

Subject	Time	DFS				Random path				Random state				MD2U				Combined			
		Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA
Before reduction																					
Sed	10m	67.5	80.0	**97.0**	80.7	**94.7**	*94.2*	92.5	*94.3*	85.8	90.9	**91.1**	90.3	87.5	91.3	91.3	**92.3**	94.3	95.5	90.9	**95.6**
	20m	70.5	80.9	**97.1**	80.5	93.9	94.9	90.3	**96.4**	86.3	**92.1**	91.7	90.2	85.8	**93.8**	90.2	91.9	93.9	95.9	88.9	**96.8**
Space	10m	98.2	**99.2**	96.5	99.2	99.3	99.3	97.7	**99.5**	98.8	99.2	97.0	**99.3**	98.8	99.2	97.0	**99.3**	99.3	99.3	97.7	**99.5**
	20m	**99.5**	99.5	99.5	99.5	**99.5**	99.5	99.5	99.5	**99.5**	99.5	99.5	99.5	**99.5**	99.5	99.5	99.5	**99.5**	99.5	99.5	99.5
Grep	10m	72.1	89.9	**91.3**	88.0	96.3	**97.4**	*96.1*	97.3	91.1	**94.7**	*90.1*	93.8	91.5	**95.8**	93.0	94.6	94.8	**96.9**	94.9	96.8
	20m	73.7	**90.6**	90.6	88.0	92.0	*93.0*	**93.2**	*91.8*	88.7	**93.4**	87.6	91.9	89.5	93.4	86.8	**94.0**	92.6	**93.7**	93.3	*93.1*
Gzip	10m	89.9	97.6	93.8	**97.9**	82.1	**95.0**	93.6	94.3	87.9	97.2	93.7	**97.3**	86.7	**98.4**	93.8	98.1	83.0	**95.8**	93.7	95.1
	20m	81.7	96.3	94.0	**96.6**	82.4	**95.5**	93.9	95.2	80.8	**97.5**	93.8	95.9	80.4	**97.4**	93.8	95.3	84.2	**96.9**	94.0	96.5
Vim	10m	72.3	**87.7**	84.2	83.8	89.7	**96.5**	86.6	95.4	79.3	**90.9**	84.4	87.2	78.0	**90.6**	85.3	85.1	75.5	**90.3**	86.0	85.9
	20m	67.4	**86.1**	79.7	81.3	90.1	**97.0**	*88.0*	96.3	80.4	**92.7**	85.6	89.7	81.0	**91.9**	86.1	88.7	74.4	**89.2**	83.7	85.8
YAFFS2	10m	71.1	65.3	*72.0*	*70.5*	88.9	89.5	*88.8*	**90.3**	91.4	91.5	90.5	*92.0*	91.1	91.5	90.6	*91.5*	87.8	86.4	*87.9*	*88.5*
	20m	71.5	65.7	**74.0**	71.3	89.2	*89.8*	**91.0**	90.7	91.3	*91.5*	**92.8**	92.2	91.2	92.1	**92.9**	92.2	87.8	86.7	**90.1**	*88.3*
After reduction																					
Sed	10m	76.9	79.1	**96.6**	90.4	94.8	*95.0*	93.2	**95.5**	86.6	88.9	**92.8**	90.6	86.9	89.8	**92.6**	91.3	94.5	96.1	91.1	**96.6**
	20m	79.4	86.6	**95.7**	95.0	94.8	95.7	92.0	**96.1**	88.1	92.3	93.1	**94.1**	86.9	92.1	**92.9**	92.8	94.8	96.4	91.1	**96.7**
Space	10m	76.6	79.1	**89.2**	84.6	99.0	*99.2*	95.3	**99.4**	85.9	88.1	**94.4**	88.5	83.1	86.1	**91.2**	87.2	83.2	*81.7*	89.2	**90.3**
	20m	70.9	*72.3*	**84.1**	82.6	**99.5**	99.5	99.0	99.5	81.7	85.7	**94.8**	88.3	80.8	84.5	**91.5**	87.1	76.2	*75.4*	84.1	**86.7**
Grep	10m	68.9	84.4	**95.8**	86.2	93.9	95.3	**97.3**	96.0	82.9	89.8	**94.8**	90.3	82.2	89.7	**96.0**	91.5	88.3	92.6	**96.7**	94.1
	20m	69.6	85.8	**95.2**	86.3	86.2	90.4	**96.0**	91.2	79.3	88.9	**94.9**	90.5	80.1	89.4	**95.6**	91.4	86.3	92.7	**96.6**	93.0
Gzip	10m	89.9	96.8	85.3	**97.0**	80.3	**94.5**	73.9	93.7	87.6	96.4	82.5	**96.7**	87.4	96.9	79.9	**97.1**	82.1	**95.5**	74.4	94.7
	20m	81.0	**96.3**	*80.1*	96.3	78.2	**95.0**	73.4	94.4	79.4	**96.8**	76.3	96.7	80.5	**97.2**	74.9	96.8	80.4	**96.4**	73.2	96.2
Vim	10m	71.6	**88.8**	80.5	83.0	88.2	**96.5**	85.5	95.5	77.9	**91.6**	82.2	87.4	77.4	**91.2**	82.1	86.8	76.3	**91.1**	82.1	86.7
	20m	67.4	**84.8**	77.0	82.5	89.7	**96.8**	86.8	96.4	82.5	**93.4**	84.0	90.9	81.2	**91.7**	84.9	88.7	73.8	**88.9**	81.6	86.3
YAFFS2	10m	72.7	64.2	**79.1**	70.2	88.8	89.6	*88.5*	**90.7**	87.0	**89.6**	88.6	88.5	87.3	**90.0**	89.3	*88.1*	85.4	*85.8*	**87.7**	87.1
	20m	72.5	65.1	**82.3**	*71.7*	89.9	*90.3*	90.6	**91.3**	89.2	90.1	90.3	**90.3**	89.2	**90.9**	90.7	90.8	87.2	*87.0*	**89.8**	88.1
The best results		1	7	12	3	3	9	4	8	1	10	7	5	1	11	7	4	1	10	5	7
Better than random		-	18	19	19	-	16	7	20	-	22	17	22	-	23	18	21	-	17	15	20
Worse than random		-	4	2	2	-	0	12	0	-	0	5	0	-	0	5	0	-	2	7	0

for the lower total gain. This is important to keep in mind when reading results below, as often APBD will be "better" for some less effective explorations; however our results only compare prioritizations across the same final total.

For each subject program, we generated 150 test suites, each including 100 randomly chosen test cases from the pool. We computed APBD values for each test suite with all ranking techniques and search strategies and averaged the results (we also performed the same experiment using reduced test cases). Table 5 shows APBD values for most prioritization techniques. Values in *italics* indicate cases where the difference between a technique and random ordering was not statistically significant (using the same test as with reduction results). Because the results for **RQ1** show that reduction is generally a good strategy, the table shows results using reduced test cases, which are generally better. Assuming reduction is desirable, we want to know which prioritizations work best for reduced test cases. Branch Total (BT) prioritization generally performed worst in almost all experiments, and in fact performed worse than random ordering, so is omitted from the tables. We are unclear why in these experiments and in the cause reduction experiments [12] branch total ordering performed so poorly, but we do not suggest its use in prioritization for symbolic execution. The best prioritization varies by subject and strategy, but a few general points are clear. First, all prioritizations outperform random ordering in general. Second, the non-traditional prioritizations are competitive, despite not maximizing branch coverage, and work differently than the regression-prioritization based method. Finally, shortest path (SP) was most effective in two cases: it benefited DFS searches in general, and was often the best prioritization for `YAFFS2`, the most complex and difficult to test of our subjects. We suspect that SP may prove even more valuable as we attempt to apply symbolic execution to increasingly complex programs.

To visually compare all prioritization methods, we use a box plot to show median and mean APBD values for each prioritization method over all subject programs, search strategies, search times, and reduced and original test cases. In Figure 3, the horizontal lines in the boxes denote median values while the circles are for mean values. "Mix all" is the result putting all search strategies into one data set. Table 6 summarizes the overall performance of prioritizations for each search strategy over all subjects. The effectiveness of SP for DFS searches is clear, as well as the general improvement of all methods on random ordering.

7. OVERALL EFFECTS OF TEST CASE REDUCTION AND PRIORITIZATION

Figure 4 graphically shows the contributions of reduction and prioritization. Each graph shows, for one subject, (1) the average incremental branch discovery curve for the search strategy that performs best (as measured by APBD) for the baseline case of no reduction and random ordering, (2) the curve for that same strategy with test case reduction

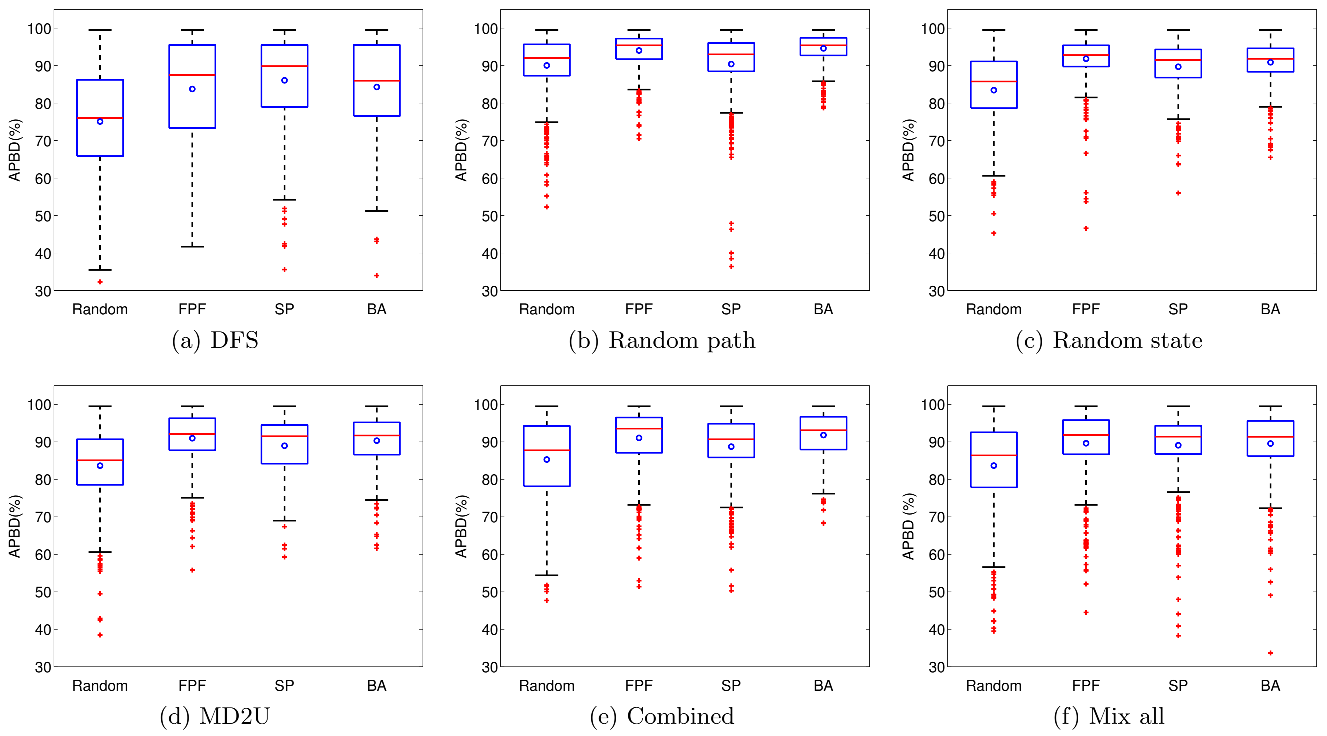


Figure 3: APBD values by search strategy.

Table 6: Average Percentage Branches Discovered (APBD) values by search strategy over all programs, search times, and reduced and original test cases (Rnd: Random, FPF: Furthest Point First, SP: Shortest Path, BA: Branch Additional) (percentage %)

APBD statistics	DFS				Random path				Random state				MD2U				Combined			
	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA	Rnd	FPF	SP	BA
Median	75.0	86.0	90.0	85.6	91.8	95.2	92.6	95.2	86.5	92.8	91.6	91.8	86.2	92.6	91.7	92.0	88.0	93.5	90.8	93.1
Mean	73.9	82.8	86.6	83.8	89.8	93.9	90.4	94.3	84.5	91.8	89.7	90.9	84.2	91.6	89.6	90.8	85.7	91.1	88.5	91.6
Standard deviation	15.3	13.8	11.4	12.3	8.7	5.2	8.2	4.3	10.4	5.9	7.0	5.8	10.7	6.4	7.2	6.1	10.7	7.5	8.6	6.4

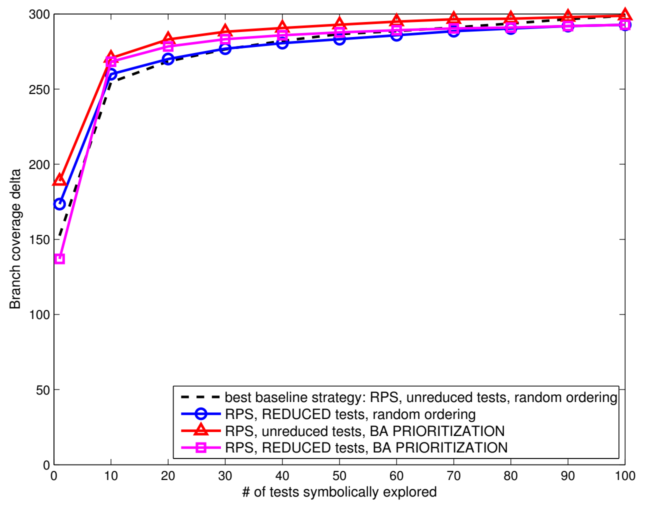
and random ordering, (3) the curve for that strategy with the best prioritization but no reduction, (4) the curve for that strategy with the best prioritization and with test case reduction, (5) the best APBD curve for that subject over all experiments, if this is not already included in 1-4, and (6) the baseline for 5 if it is not already included. These graphs show, first, that the value of reduction vs. prioritization varies with subject and strategy. Second, in three cases using reduction and prioritization not only improves a method, but changes the most effective strategy for search. The `space` and `grep` examples are particularly compelling: the baselines for DFS and MD2U respectively in these examples are strikingly different than the curves for the same search strategy with reduction and prioritization, so much so that these become the optimal strategies for exploring the state space by a large margin over the best strategy without reduction and prioritization. The same thing happens with `YAFFS2`, though the difference in baseline and final curves is less dramatic.

8. THREATS TO VALIDITY

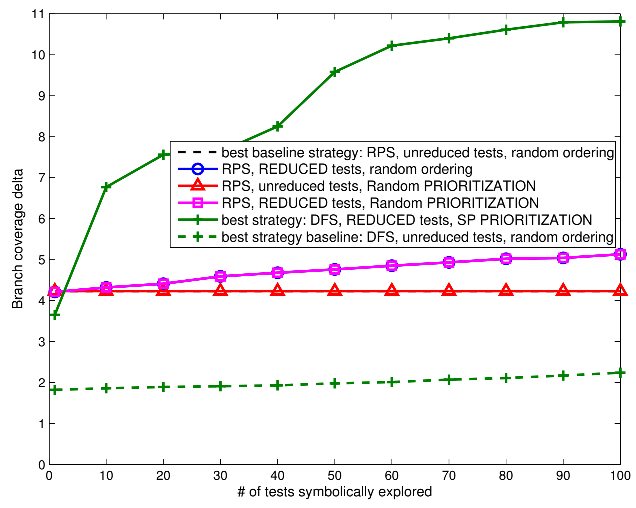
There are two major threats to validity. First, our results cover only six modest-sized C programs, the largest of which has only a little over 100KLOC. These programs may not be representative, though all are frequently studied subjects in the testing literature. Second, there could be errors in our implementation and experimental framework. We have performed a variety of cross-checks to avoid this, but the possibility for error remains. Because our technique does not require modification of the symbolic execution engine, there is no possibility of new bugs introduced into the symbolic execution process, however. We have noted when differences in techniques are not statistically significant, under a statistical test that does not assume normality (and generally is harder to pass than a t-test).

9. RELATED WORK

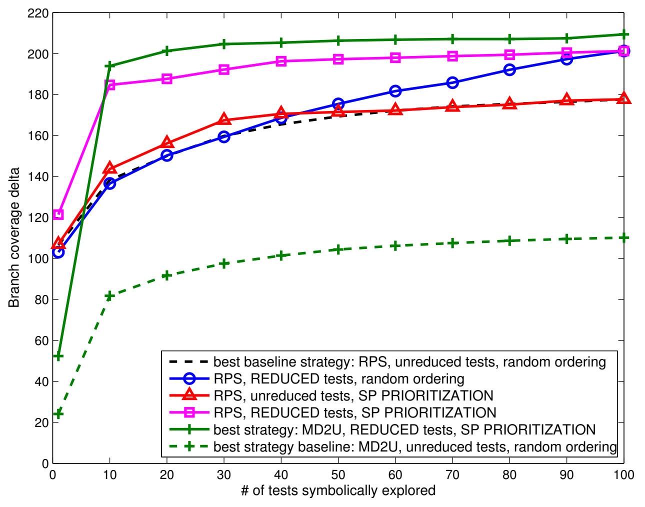
Related works can be divided into three basic categories. **Using Existing Test Cases for Seeding Symbolic Execution.** Seeded dynamic symbolic execution takes an initial test case and tries to cover a branch that test has not covered. This initial test case is called the seed. In the lit-



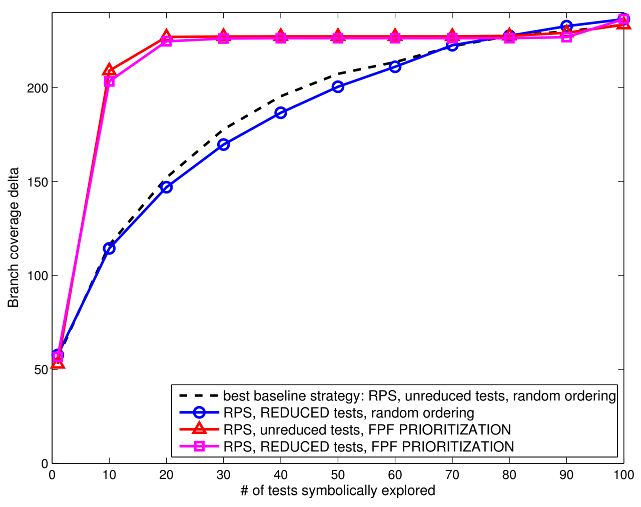
(a) sed



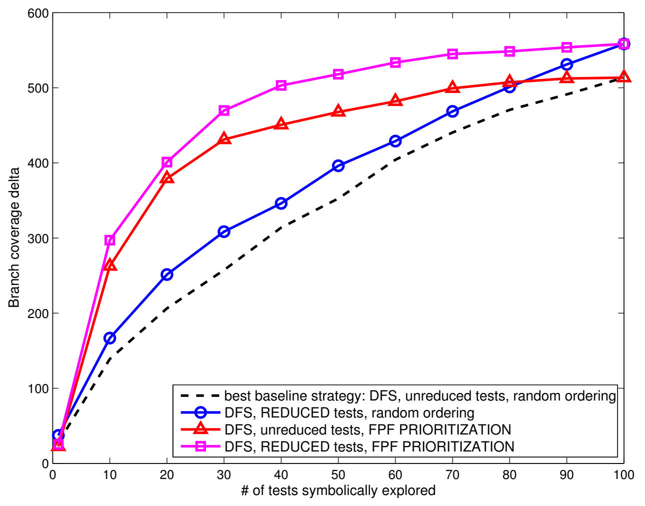
(b) space



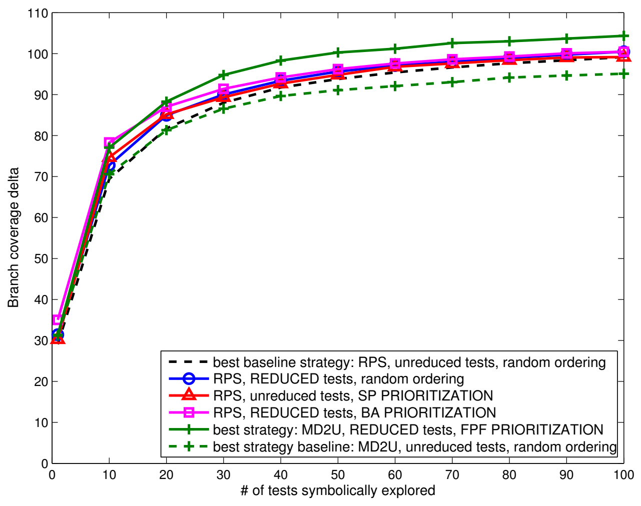
(c) grep



(d) gzip



(e) vim



(f) yaffs2

Figure 4: Ranking methods comparision for all subject programs.

erature, the seed is usually chosen arbitrarily from a pool of test cases [7, 9, 22, 28]. Some methods choose seeds for regression purposes [30]. KATCH [24] chooses the test case closest (by a distance metric based both on static code structure and weakest preconditions) to a target which is a code modification in a patch. Our work differs from previous efforts partly in goal but primarily in that we manipulate the test cases in the pool prior to performing symbolic execution and use novel prioritization methods.

Test Case Reduction. Large test cases, especially those generated by random testing, usually include redundant behaviors. Understanding such long test cases is difficult for developers. Delta debugging [16] is a general technique for reducing the size of a failing test case while holding the fact that the test case fails constant. Test case reduction has been traditionally applied only to failing test cases. Recently, we proposed using delta debugging based on preserving *code coverage* to reduce both failing and successful test cases for purposes of building very fast regression suites [12]. This paper adapts the same technique to the problem of improving the scalability of symbolic execution.

Test Case Prioritization. Test case prioritization [32] ranks test cases such that faults can be discovered earlier in the testing process. Rothermel et al. [27] have proposed and experimented with different orderings of test cases based on statement and branch coverage. Statement (branch) total prioritization ranks test cases based on their absolute statement (branch) coverage; the higher the statement (branch) coverage of a test case, the sooner it is executed. Statement (branch) delta prioritization ranks test cases by the *new* coverage over all previously ranked test cases. Our work does not aim to explore the wide variety of sophisticated algorithms for prioritization, a highly active field [34] but only to establish that prioritization can improve the efficiency of symbolic execution, in terms of how quickly additional branches are covered.

10. CONCLUSIONS AND FUTURE WORK

This paper addresses two research questions concerning improving test case seed based symbolic execution, in the context of a proposed two-stage framework for testing. The core idea is that a low-cost testing method can be used to cover the easily-tested branches of a program, and symbolic execution can be used to (at high computational cost) cover additional branches. The hypothesis of this paper is that applying test case reduction and test case prioritization techniques to the initial set of test cases from which symbolic execution proceeds can improve the results of symbolic execution, both in terms of total additional branches covered and in terms of how quickly additional branches are covered.

- **RQ1:** *Can test case reduction improve the effectiveness of seeded symbolic execution?*

- **RQ2:** *Given a fixed search time for each seed test case, can ranking seed tests improve the efficiency of symbolic execution?*

Our experimental results over 6 C programs using the popular `zesti` version of KLEE demonstrate that not only is the answer to both questions affirmative, but that for most of our subjects, for most symbolic execution configurations, both test case reduction and prioritization can increase the effectiveness of symbolic execution in a statistically significant way. In several cases, test reduction leads to an improvement in additional branch coverage of 40-100%. The value of prioritization is a substantial improvement in the discovery curves for new branches as symbolic execution proceeds. For three of the six subjects, applying our methods not only improves the results for the best symbolic search strategy for the program, it improves a previously inferior strategy's results such that it becomes the best way to explore the program's behavior. Given that without taking advantage of parallelism, full symbolic exploration in our 20 minutes-per-test budget experiments requires over 30 hours, improving the speed with which new branches are discovered as well as total additional coverage is critical, especially as in a real two-stage framework test suites will be considerably larger, and test cases likely more varied in quality. In realistic situations, the full test suite will almost certainly never be symbolically explored.

This paper does not aim to devise an optimal strategy for reducing and prioritizing test cases for symbolic execution. Instead, it serves to demonstrate the value of reduction and prioritization even with sub-optimal methods. We speculate that statement-coverage based reduction is an effective strategy that could be easily applied to current symbolic execution workflows, and may be difficult to improve on, given the need to balance (1) preserving the value of the existing test case while (2) obtaining enough reduction to aid symbolic execution. The prioritization strategies, however, are best seen as a starting point for future research, especially investigations of which strategies work best when exploration of all test cases is impossible, and the problem becomes one of selection rather than mere prioritization. We also expect that the problem of how to produce test cases from which to perform symbolic execution deserves further study. In particular, the best approaches to determining when to stop automated testing (e.g., detecting coverage saturation) may be altered in interesting ways if the sequel to automated testing is not an end to testing, but a beginning to a more computationally demanding symbolic exploration to fill in the gaps in a test effort.

Acknowledgments. We thank the anonymous ISSTA '14 reviewers for their comments on the draft of this paper. A portion of this work was funded by NSF grants CCF-1217824 and CCF-1054786. We would also like to thank John Regehr, Xiaoli Fern, Weng-Keen Wong, and Patrice Godefroid for discussions that contributed to our thinking about some ideas considered in this paper.

11. REFERENCES

[1] A. Arcuri and L. C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *International Conference on Software Engineering*, pages 1–10, 2011.

[2] E. Bounimova, P. Godefroid, and D. Molnar. Billions and billions of constraints: whitebox fuzz testing in production. In *International Conference on Software Engineering*, pages 122–131, 2013.

[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

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

[5] Count lines of code. `http://cloc.sourceforge.net/`.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Softw. Engg.*, 10:405–435, 2005.

[7] P. Garg, F. Ivancic, G. Balakrishnan, N. Maeda, and A. Gupta. Feedback-directed unit test generation for c/c++ using concolic execution. In *International Conference on Software Engineering*, pages 132–141, 2013.

[8] M. Gligoric, A. Groce, C. Zhang, R. Sharma, M. A. Alipour, and D. Marinov. Comparing non-adequate test suites using coverage criteria. In *ACM International Symposium on Software Testing and Analysis*, pages 302–313. ACM, 2013.

[9] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

[10] T. F. Gonzalez. Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306, 1985.

[11] A. Groce. Error explanation with distance metrics. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 108–122, March 2004.

[12] A. Groce, M. A. Alipour, C. Zhang, Y. Chen, and J. Regehr. Cause reduction for quick testing. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014. accepted for publication.

[13] A. Groce, K. Havelund, G. Holzmann, R. Joshi, and R.-G. Xu. Establishing flight software reliability: Testing, model checking, constraint-solving, monitoring and learning. *Annals of Mathematics and Artificial Intelligence*. Accepted for publication.

[14] 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.

[15] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *International Symposium on Software Testing and Analysis*, pages 78–88, 2012.

[16] R. Hildebrandt and A. Zeller. Simplifying failure-inducing input. In *International Symposium on Software Testing and Analysis*, pages 135–145, 2000.

[17] W. Jin and A. Orso. Bugredux: Reproducing field failures for in-house debugging. In *International Conference on Software Engineering*, pages 474–484, 2012.

[18] Y. Kim, M. Kim, Y. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using crest-bv and klee. In *International Conference on Software Engineering*, pages 1143–1152, 2012.

[19] Y. Kim, Z. Xu, M. Kim, M. B. Cohen, and G. Rothermel. Hybrid directed test suite augmentation: An interleaving framework. In *Software Testing, Verification and Validation (ICST), 2014 IEEE Seventh International Conference on*. IEEE, 2014. accepted for publication.

[20] Y. Lei and J. H. Andrews. Minimization of randomized unit test cases. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering*, ISSRE '05, pages 267–276, 2005.

[21] Y. Li, Z. Su, L. Wang, and X. Li. Steering symbolic execution to less traveled paths. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 19–32, 2013.

[22] R. Majumdar and K. Sen. Hybrid concolic testing. In *International Conference on Software Engineering*, pages 416–426, 2007.

[23] P. D. Marinescu and C. Cadar. make test-zesti: a symbolic execution solution for improving regression testing. In *International Conference on Software Engineering*, pages 716–726, 2012.

[24] P. D. Marinescu and C. Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 235–245, 2013.

[25] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *International Conference on Software Engineering*, pages 75–84, 2007.

[26] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization. *Trans. Softw. Eng.*, 27:929–948, 2001.

[27] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold. Test case prioritization: An empirical study. In *Proceedings of the IEEE International Conference on Software Maintenance*, ICSM '99, pages 179–188, Washington, DC, USA, 1999. IEEE Computer Society.

[28] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for c. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, 2005.

[29] Z. Xu, Y. Kim, M. Kim, and G. Rothermel. A hybrid directed test suite augmentation technique. In *ISSRE*, pages 150–159, 2011.

[30] Z. Xu, Y. Kim, M. Kim, G. Rothermel, and M. B. Cohen. Directed test suite augmentation: Techniques and tradeoffs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 257–266, 2010.

[31] YAFFS: A flash file system for embedded use. `http://www.yaffs.net`.

[32] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, Mar. 2012.

[33] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, Feb. 2002.

[34] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei. Bridging the gap between the total and additional test-case prioritization strategies. In *International Conference on Software Engineering*, pages 192–201, 2013.